Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results*

Zeki Bozkus, Alok Choudhary†, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka‡
Northeast Parallel Architectures Center, Syracuse University

Abstract

Fortran 90D/HPF is a data parallel language with special directives to enable users to specify data alignment and distributions. This paper describes the design and implementation of a Fortran90D/HPF compiler. Techniques for data and computation partitioning, communication detection and generation, and the run-time support for the compiler are discussed. Finally, initial performance results for the compiler are presented. We believe that the methodology to process data distribution, computation partitioning, communication system design and the overall compiler design can be used by the implementors of HPF compilers.

1 Introduction

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 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 a number of languages such as Fortran D, CM Fortran [4] and Vienna Fortran [5]. HPF essentially adds extensions to Fortran 90 similar to the Fortran D directives. Hence, Fortran 90D and HPF are very similar except a few syntactic 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], but it is also a natural language for specifying parallelism in a class of problems called *loosely synchronous* problems. 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.

*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. Alok Choudhary is also supported by an NSF Young Investigator Award CCR-9357840.

†Also with ECE Dept.

‡Also with CIS Dept.

© 1993 ACM 0-8186-4340-4/93/0011 $1.50

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.

The rest of this paper is organized as follows. The compiler architecture is described in Section 2. Data partitioning, and computation partitioning are discussed in Sections 3, and 4. Section 5 presents the communication primitives and communication generation for Fortran 90D/HPF programs. In Section 6, we present the runtime support system including the intrinsic functions. Some optimization techniques are given in Section 7. Section 8 summarizes our initial experience using the current version of the compiler. It also presents a comparison of the performance with hand written parallel code. Section 9 presents a summary of related work. Finally, summary and conclusions are presented in Section 10.

2 Compiler System Diagram

Our Fortran 90D/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. 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.

Figure 1 shows the components of the basic Fortran 90D/HPF compiler. 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 an equivalent *forall* statement with no loss of information [6]. 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.

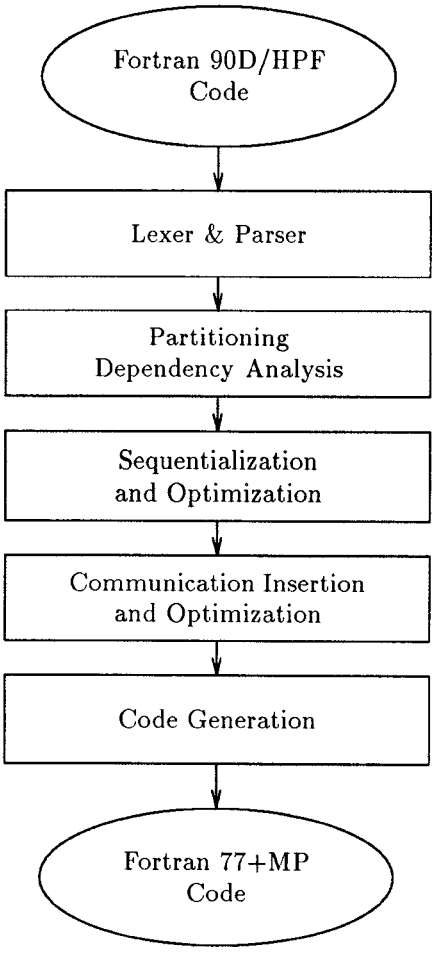

Figure 1: Diagram of the compiler.

After partitioning, the parallel constructs in the node program are sequentialized since they will 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* [7] 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

The distributed memory system solves 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 efficiency 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 DISTRIBUTE 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 Fortran 90D/HPF compiler maps arrays to physical processors by using a three stage mapping as shown in Figure 2 which is guided by the user-specified Fortran D directives.

Stage 1 : The alignment of arrays to template is determined by their subscript expressions in the ALIGN directive. The compiler computes f and f^{-1} function from the directive and applies f functions for the corresponding array indices to bring them onto common template index domain. The original indices can be calculated by f^{-1} if they are required. The algorithm to compile align directive can be found in [8].

Stage 2 : Each dimension of the template is mapped onto the logical processor grid, based on the DIS-

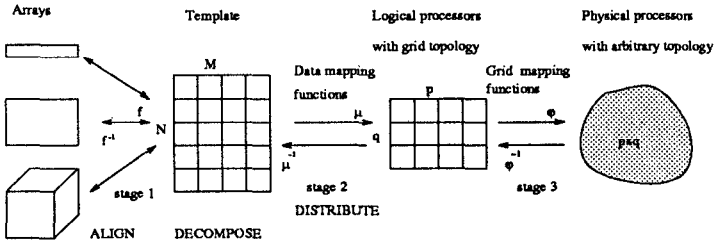

Figure 2: Three stage array mapping

TRIBUTE directive attributes. *Block* divides the template into contiguous chunks. *Cyclic* specifies a round-robin division of the template. The mapping functions μ and μ^{-1} to generate relationship between global and local indices are computed.

Stage 3 : The logical processor grid is mapped onto the physical system. The mapping functions φ and φ^{-1} can change from one system to another but the data mapping onto the logical processor grid does not need to change. This enhances portability across a large number of architectures.

By performing the above three stage mapping, the compiler is decoupled from the specifics of a given machine or configuration. Compilation of distribution directives is discussed in detail in [8].

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 [5, 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.

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 total number of iterations will still be the same as the number of *lhs* array elements being assigned.

5 Communication

Our Fortran 90D/HPF compiler produces calls to collective communication routines [10] instead of generating individual processor send and receive calls inside the compiled code. There are three main reasons for using collective communication to support interprocessor communication in the Fortran 90D/HPF compiler.

1. *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 [11]. 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 automatically.

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

3. *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.

5.1 Communication Primitives

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 preprocessing 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 preprocessing. We classify our communication primitives into *unstructured* and *structured* communication.

Our structured communication primitives are based on a logical grid configuration of the 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.
- **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 [12].
- **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.
- **concatenation:** This primitive concatenates a distributed array and the resultant array ends up in all the processors participating in this primitive.

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 preprocessing that involves local computations

[13], and 2) where to determine the send and receive lists preprocessing itself requires communication among the processors [14]. The primitives are as follows.

- **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 preprocessing loop.
- **gather:** This is similar to *precomp_read* except that preprocessing loop itself may require communication.
- **scatter:** This is similar to *postcomp_write* except that preprocessing loop itself may require communication.

5.2 Communication Detection

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 relationships among the 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 Li and Chen [15]. Further, we extend the above tests to include unstructured communication.

Consider the following forall statement to illustrate the steps involved in communication detection.

FORALL (i1=l1:u1:s1, i2= ..., ...)
LHS(f_1,f_2,...,f_n) = RHS1(g_1,g_2,...,g_m) + ...

where g_i and f_j, $1 \leq i \leq m$, $1 \leq j \leq n$, are functions of index variables or are indirection arrays.

The algorithm first attempts to detect structured communication if the arrays are aligned to the same template. For each array on the RHS, the following processing is performed. Each subscript of the array is coupled with the corresponding subscript on the LHS array such that both subscripts are aligned with the same dimension of the template. For each such pair, the algorithm attempts to find a structured communication pattern in that dimension according to Table 1. If a structured communication pattern is found then the subscript on the RHS from this pair is tagged with indicating the appropriate communication primitive.

Table 1: Communication primitives based on the relationship between *lhs* and *rhs* array subscript reference patterns 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	$(unknown, i)$	scatter

If any distributed dimension of an array on the RHS is left untagged then the array is marked with one of the unstructured communication primitives depending on the reference pattern. Note that any pattern that can not be classified according to Tables 1, is marked as *unknown* (such subscripts involving more than one forall index, e.g $I + J$) so that scatter and gather can be used to parallelize any forall statement.

5.3 Communication Generation

Having recognized the type of communication in each dimension of an array for structured communication or each array for unstructured communication in a forall statement, the compiler needs to perform the appropriate program transformations. We now illustrate these transformations with the aid of some examples.

5.3.1 Structured Communication

All the examples discussed below have the following mapping directives.

```
C$ PROCESSORS(P,Q)
C$ DISTRIBUTE TEMPL(BLOCK,BLOCK)
C$ ALIGN A(I,J), B(I,J) WITH TEMPL(I,J)
```

Example 1 (transfer) Consider the statement

```
FORALL(I=1 N) A(I,8)=B(I,3)
```

The first subscript of B is marked as *no_communication* because A and B are aligned in the first dimension and have identical indices. The second dimension is marked as *transfer*.

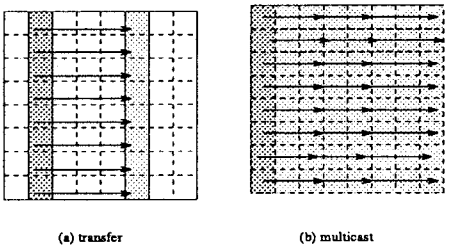

Figure 3: Structured communication on logical grid processors.

```
1.    call set_BOUND(lb,ub,st,1,N,1)
2.    call set_DAD(B_DAD,.....)
3.    call transfer(B, B_DAD, TMP,
        source=global_to_proc(3),
        dest=global_to_proc(8))
4.    DO I=lb,ub,st
5.         if(grid(2).EQ.dest)
            A(I,global_to_local(8)) = TMP(I)
6.    END DO
```

In the above code, the *set_BOUND* primitive (line 1) computes the local bounds for computation assignment based on the iteration distribution (Section 4). In line 2, the primitive set_DAD is used to fill the Distributed Array Descriptor (DAD) associated with array B so that it can be passed to the *transfer* communication primitive at run-time. The DAD has sufficient information for the communication primitives to compute all the necessary information including local bounds, distributions, global shape etc. Note that transfer performs one-to-one send-receive communication based on the logical grid. In this example, one column of grid processors communicate with another column of the grid processors as shown in Figure 3 (a).

Example 2 (multicast) Consider the statement

```
FORALL(I=1:N,J=1:M) A(I,J)=B(I,3)
```

The second subscript of B marked as *multicast* and the first one as no_communication.

```
1.    call set_BOUND(lb,ub,st,1,N,1)
2.    call set_BOUND(lb1,ub1,st1,1,M,1)
3.    call set_DAD(B_DAD,.....)
4.    call multicast(B, B_DAD, TMP,
       source_proc=global_to_proc(3), dim=2)
5.    DO I=lb,ub,st
6.    DO J=lb1,ub1,st1
7.       A(I,J) = TMP(I)
8.    END DO
```

Line 4 shows a broadcast along dimension 2 of the logical processor grid by the processors owning elements $B(I,3)$ where $1 \leq I \leq N$ (Figure 3 (b).)

5.3.2 Unstructured Communication

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 preprocessing loop before communication so that the schedule routine can combine the messages to the maximum extent. The preprocessing loop is also called the "inspector" loop [16, 13].

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)
3     DO I=1, N/P
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=1, N/P
13       if((I.ge.lb).and.(I.le.ub)
                .and.(mod(I,st).eq.0))  ! mask
14                A(I) = tmp(count)
15       count= count+1
16    END DO
```

The pre-processing loop is given in lines 1-9. Note that this pre-processing loop executes concurrently in each processor. The loop covers entire local array bounds since each processor has to calculate 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 *schedule1* routine does not need to communicate but only constructs the scheduling data structure *isch*. 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.

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. The *precomp_read* primitive brings an element into *temp* for each local array element since preprocessing loops coves entire local array. The *if* statement masks the assignment to preserve the semantic of original loop.

Example 2 (gather) Consider the statement

```
FORALL(I=1 N) A(I)=B(V(I))
```

The array B is marked as requiring *gather* communication since the subscript is only known at runtime. The receiving processors can know what nonlocal data they need from other processors, but a processor may not know what local data it needs to send to other processors. For simplicity, in this example, we assume that the indirection array V is replicated. If it is not replicated, the indirection array must also be communicated to compute the receive list on each processor.

```
1   count=1
2   call set_BOUND(lb,ub,st,1,N,1)
3   DO I=lb,ub,st
4      receive_list(count)=global_to_proc(V(i))
6      local_list(count) = global_to_local(V(i))
7      count=count+1
8   END DO
9   isch = schedule2(receive_list,
          local_list, count)
10  call gather(isch, tmp,B)
11  count=1
12  DO I=lb,ub,st
13     A(I) = tmp(count)
14     count= count+1
15  END DO
```

Once the scheduling is completed, every processor knows exactly which non-local data elements it needs to send to and receive from other processors. Recall that the task of *scheduler2* is to determine exactly which send and receive communications must be carried out by each processor. The scheduler first figures out how many messages each processor will have to send and receive during the data exchange. Each processor computes the number of elements (*receive_list*) and the local index of each element it needs from all other processors. In *schedule2* routine, processors communicate to combine these lists (a fan-in type of communication). At the end of this processing, each processor contains the send and receive list. After this point, each processor transmits a list of required array elements (*local_list*) to the appropriate processors. Each processor now has the information required to set up the send and receive messages that are needed to carry out the scheduled communication. This is done by the gather primitives.

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) [16] designed to efficiently support irregular patterns of distributed array accesses.

6 Run-time Support System

The Fortran 90D 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 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, reshape, and matrix multiplication.

Some of the intrinsic functions can be further optimized for the underlying hardware architecture. Our Fortran 90D/HPF compiler has more than 500 parallel run-time support routines and the implementation details can be found in [17].

Arrays may be redistributed across subroutine boundaries. A dummy argument which is distributed differently from 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) [17].

7 Optimizations

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 optimizations 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, to use parallelism within the single node (e.g. using attached vector units), our compiler propagates information to the node compiler using node directives. Since there is no data dependency between different loop iteration in the original data parallel constructs such as *forall* statement, vectorization can be performed easily by the node compiler.

Our compiler performs several optimizations to reduce the total cost of communication. Some of *communication* optimizations [15, 18, 14] are as follows.

1. *Vectorized communication.* Vectorization combines messages for the same source and destination into a single message to reduce communication overhead. Since we are only parallelizing array assignments and forall statements in Fortran 90D/HPF, there is no data dependency between different loop iterations. Thus, all the required communication can be performed before or after the execution of the loop on each of the processors involved.

2. *Eliminate unnecessary communications.* In many cases, communication required for two different operands can be replaced by their union. For example, the following code may require two *overlapping_shifts*. However, with a simple analysis, the compiler can eliminate the shift of size 2.

```
FORALL(I=1:N) A(I)=B(I+2)+B(I+3)
```

3. *Reuse of scheduling information. Unstructured* communication primitives are required by computations which require the use of a preprocessor. As discussed in Section 5.3.2, the schedules can be reused with appropriate analysis.

4. *Code movement.* The compiler can utilize the information that the run-time support routines do not have procedural side effects. For example, the preprocessing loop or communication routines can be moved up as much as possible by analyzing *definition-use chains*. This may lead to moving of the scheduling code out of one or more nested loops which may reduce the amount of communication required significantly. We are incrementally incorporating many more optimizations in the compiler.

Table 2: Comparison of the execution times of the hand-written code and Fortran 90D compiler generated code for several applications. (Intel iPSC/860, time is in seconds).

Program	Size	Number of PEs				
		1	2	4	8	16
Gauss Hand	1Kx1K	623 1	446.6	235.3	134 8	79.4
Gauss F90D	1Kx1K	618.7	451 9	261.8	147.2	87 4
Nbody Hand	1Kx1K	6 8	1.7	1.2	0.7	0.4
Nbody F90D	1Kx1K	13.8	5 9	2.4	1.3	0.8
Option Hand	8K	4 2	3 1	1 6	0.8	0.4
Option F90D	8K	4 3	3 1	1.6	0.8	0.4
Pi Hand	64K	0 398	0 200	0.101	0 053	0.030
Pi F90D	64K	0 411	0 207	0 104	0 054	0 032

8 Experimental Results

To illustrate the performance of our compiler, we present benchmark results from four programs and the first 10 Livermore loop kernels. *Gauss* solves a system of linear equations with partial pivoting. *Nbody* program simulates the universe using the algorithm in [10]. *Option* program predicts the stock option pricing using stochastic volatility European model. *Pi* program calculates the value of pi, using numerical integration. The Livermore kernels are 24 loops abstracted from actual production codes that have been widely used to evaluate the performance of various computer systems. Data for all programs were block distributed and were written outside of the compiler group at NPAC by experienced message passing programmers.

Tables 2 and 3 show the performance of compiler generated codes ($F90D/HPF$) and hand-written f77+ MP code. The tables contain data from running these programs with varying number of processors an Intel iPSC/860. The compiler generated codes and hand-written codes use Express as a message passing library. Timings were taken using *extime()* function having an accuracy of one microsecond. The programs were compiled by using Parasoft Express Fortran compiler which calls Portland Group if77 release 4.0 compiler with all optimizations turned on (-O4).

Table 3: Comparison of the execution times of the hand-written code and Fortran 90D compiler generated code for the first 10 Livermore loop kernels. Data size is 16K real. (a 16 node Intel iPSC/860, time is in milliseconds).

Loop #	Type of Application	F90D	Hand	Ratio
1	Hydrodynamics	2.545	2.550	0 99
2.	Incomplete Cholesky	11 783	10.440	1.12
3	Inner product	3 253	3.249	1.00
4	Banded linear equations	5 139	3 212	1 60
5	Tridiagonal elimination	30928	30897	1 00
6.	Linear recurrence relations	1849 1	1886 5	0.98
7	Equation of state	11 346	3 704	3 06
8	A D I	38 656	20 038	1 92
9	Numerical Integration	2 255	2 441	0 92
10	Numerical Differentiation	9 814	4 589	2 13

We observe that the performance of the compiler generated codes are usually within a factor of 2 of the hand-written codes. This is due to the fact that experienced programmer can incorporate more optimization than our compiler currently does. For example, a programmer can combine or eliminate some of the communication or some of intra-processor temporary copying. The compiler uses a more generic packing routine, whereas a programmer can combine communication for the same source and destination for different arrays. Another observation is that our run-time system shift routine is slower than the programmer's shift routines. We are planing to rewrite some part of our run-time shifts using assembly language.

9 Summary of Related Work

Callahan and Kennedy [9] proposed distributed-memory compilation techniques based on data-dependence driven program transformations. These techniques were implemented in a prototype compiler in the ParaScope programming environment. Currently, a Fortran 77D compiler is being developed at Rice [18]. The Fortran 77D compiler introduces and classifies a number of advanced optimizations needed to achieve acceptable performance; they are analyzed and empirically evaluated for stencil computations. SUPERB [5] is a semi-automatic parallelization tool designed for MIMD distributed-memory machines. It supports arbitrary user-specified contiguous rectangular distributions, and performs dependence analysis to guide interactive program transformations. KALI [13] is the first compiler system that supports both regular and irregular computations on MIMD machines. KALI requires that the programmer explicitly partition loop iterations onto the processor grid. An inspector/executor strategy is used for run-time preprocessing of the communication for irregularly distributed arrays. Dataparallel C [19] is a variant of the original C* programming language, designed by Thinking Machines Corporation for its Connection Machines processor array. Data parallel C extends C to provide the programmer access to a parallel virtual machine. ARF is a compiler for irregular computations [14]. Saltz *et al.* describe and experimentally characterize ARF compiler and runtime support procedures which embody methods that are capable of handling a wide range of irregular problems in scientific computing. Many techniques especially unstructured communication of Fortran 90D compiler are adapted from ARF compiler. The ADAPT system [20] compiles Fortran 90 for execution on MIMD distributed memory architectures. The ADAPTOR [21] is a tool that transform data parallel programs written in Fortran with array extensions and layout directives to explicit message passing. Li and Chen [22] describe general compiler optimization techniques that reduce communication overhead for Fortran-90 implementation on massivelly parallel machines. Our compiler uses pattern matching techniques to detect communication similar to Li and Chen's. Sabot [23] describes the techniques that are used by the CM compiler to map the fine-grained array parallelism of languages such as Fortran 90 and C* onto the Connection Machine architectures.

10 Conclusions

In this paper, we presented design, implementation and performance results of our Fortran 90D/HPF compiler for distributed memory machines. Specifically, techniques for processing distribution directives, computation partitioning, communication detection and generation were presented. We also showed that our design is portable, yet efficient.

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

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

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

[8] Z. Bozkus et al. Compiling Distribution Directives in a Fortran 90D Compiler. Technical Report SCCS-388, Northeast Parallel Architectures Center, July 1992.

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

[10] 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.

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

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

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

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

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

[17] 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.

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

[19] Philip Hatcher, Anthony Lapadula, Robert Jones, Michael Quinn, and Ray Anderson. A Production-Quality C* Compiler for Hypercube Multicomputers. *Third ACM SIGPLAN symposium on PPOPP*, 26:73–82, July 1991.

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

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

[22] 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.

[23] G. Sabot. A Compiler for a Massively Parallel Distributed Memory MIMD Computer. *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992.