1994

# Compiling Fortran 90D/HPF for 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

# Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers*

Zeki Bozkus, Alok Choudhary†, Geoffrey Fox, Tomasz Haupt, Sanjay Ranka‡ and Min-You Wu§

Syracuse University

Northeast Parallel Architectures Center

3-201, Center for Science and Technology

Syracuse University

Syracuse, NY, 13244-4100

{zbozkus, choudhar, gcf, haupt, ranka, wu}@npac.syr.edu

**Abstract**

This paper describes the design of the Fortran90D/HPF compiler, a source-to-source parallel compiler for distributed memory systems being developed at Syracuse University. Fortran 90D/HPF is a data parallel language with special directives to specify data alignment and distributions. A systematic methodology to process distribution directives of Fortran 90D/HPF is presented. Furthermore, 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 compilers for HPF.

*This work was supported by ARPA 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

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

‡Sanjay Ranka, 4-116 CST, CIS Dept, Syracuse University, Syracuse, NY, 13244

§Min-You Wu, CS Dept. State University of New York, Buffalo, NY, 14260

# 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 state of the 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 accesses.

There has been significant research in developing parallelizing compilers. In this approach, the compiler takes a sequential 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 difficult to detect by a parallelizing compiler. Thus, in our opinion, 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 include parallel loop, such as a *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 a 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 for 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, 6], 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. Output is a SPMD (Single Program Multiple Data) program with appropriate data and computation partitioning and communication calls for distributed memory 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 rest of this paper is organized as follows. The compiler system overview 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. Section 7 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 8 presents a summary of related work. Finally, a summary and conclusions are presented in Section 9.

## 2 Compiler System Overview

Our Fortran 90D/HPF 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 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.

The basic structure of our Fortran 90D/HPF compiler is organized around four major modules: parsing, partitioning, communication detection and insertion, and code generation. 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 Corp. This module parses the input program into an abstract syntax tree, performs semantic analysis to annotate the tree with type information, and builds up a symbol table; it also performs error checking.

Our compiler transforms each array assignment statement and *where* statement into an equivalent *forall* statement with no loss of information [7]. In this way, subsequent steps need only deal with *forall* statements. Currently, our compiler does not handle module, pointer, and allocatable array statements of Fortran 90.

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

Dependence analysis is carried out to obtain dependence information for use in sequentialization of the data parallel constructs and insertion of communication primitives. Standard techniques of data dependence analysis for Fortran programs can be applied here [8].

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 a *loosely synchronous* 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. Collective 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 requirehigh 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 data *alignment* and *distribution* specifications. It has three main compiler directives.

- DECOMPOSITION

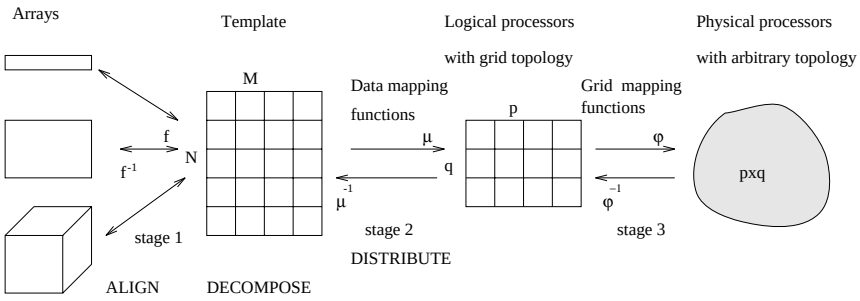



Figure 1: Three stage array mapping

- ALIGN
- DISTRIBUTE

The DECOMPOSITION directive is used to declare the name, dimensionality, and the size of each problem domain. A decomposition is simply an abstract representation of a problem or index domain. We call it "template" (a name 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. There may be multiple templates representing different problem mappings, but an array may be aligned to only one template at any time. All scalars are replicated. An array not explicitly aligned to any template serves as its own template.

The DISTRIBUTE directive specifies coarse-grain parallelism, grouping template elements and mapping them (and aligned array elements) to the finite resources of the machine. Each dimension of the template is distributed in either 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 the physical processors using a three stage mapping as shown in Figure 1. This three stage mapping has also been proposed in HPF [3].

**Stage 1:** ALIGN directives are processed to compute functions that map the *array index domain* to the *template index domain* and vice versa.

**Stage 2:** Each dimension of a template is mapped onto the *logical processor grid* based on the DISTRIBUTE directives. *Block* divides the template into contiguous chunks. *Cyclic* specifies a round-robin division of the template. The mapping functions $\mu$ and $\mu^{-1}$ to generate relationship between global and local indices are computed. These function have been studied extensively by

Koelbel[9].

**Stage 3:** The logical processor grid is mapped onto the physical system. This mapping 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. We now give the compilation techniques of Stage 1. The compilation of distribution directives and Stages 2 and 3 are described in more detail [10].

## Compiling the ALIGN Directive (Stage 1)

Alignment determines which portions of two or more arrays will be mapped to the same processor. Clearly, if arrays involved in the same computation are aligned in such a manner that after distribution their respective sections lie on the same processor then the number of non-local accesses will be reduced. The *DECOMPOSITION* directive defines the shape and rank of a given template. Let $A$ be an $m$-dimensional array and $TEMPL$ be an $n$-dimensional template. The general form of the alignment directive is

**C$ ALIGN A**($i_1$[*], ... ,$i_m$[*]) **WITH TEMPL**($f_1(i_{a_1})$[*], ... ,$f_n(i_{a_m})$[*]).

The specified elements of $A$ are aligned to those of $TEMPL$. The template is eventually distributed on a set of processors and the compiler guarantees that the array elements aligned to the same element of the template will be mapped to the same processor.

An *alignment function*, $f_k$, is required to be an affine function. That is, $f_k = s_k * i_{a_k} + o_k$ or $f_k = o_k$. The parameters $i_{a_k}$, $s_k$, and $o_k$ correspond to the three components of the alignment: *axis*, *stride*, and *offset*. Misalignment in the stride components causes *unstructured communication*, and misalignment in the offset component causes nearest-neighbor communication [11].

The Fortran 90D/HPF compiler requires that each of $A$'s subscripts $i_1$, ... ,$i_m$ appear exactly once on the $TEMPL$'s subscripts, so that a one-to-one correspondence with a section of $TEMPL$ is established. This restriction does not permit skew alignments such as aligning $A(I)$ with $TEMPL(I, I)$ or $A(I, J)$ with $TEMPL(I + J)$. The order of axes in the array may be different from the order of axes in the template. This permits transpose style alignments such as aligning $A(I, J)$ with $TEMPL(J, I)$.

The symbol "*" indicates that the corresponding dimension is replicated or collapsed. It may

**Algorithm 1 (Compiling Align directives)**

*Input:* Fortran 90D/HPF syntax tree with some alignment functions to template

*Output:* Fortran 90D/HPF syntax tree with identical alignment functions to template

*Method:* For each aligned array, and for each dimension of that array,
carry out the following steps:

**Step 1.** Extend aligned arrays to match template size.

**Step 2.** Determine local shape of arrays.

**Step 3.** Apply alignment functions to the aligned arrays.

**Step 4.** Transform into canonical form.

**Step 5.** Compute $f^{-1}(i)$.

appear in both the array and the template subscripts. The array rank (number of dimensions) $m$ may be different from the rank of the template, $n$. For example, the directive

**C$ ALIGN A**($i$,*) **WITH TEMPL**($i+1$)

requires the second dimension of the array $A$ be collapsed (not distributed), while the directive

**C$ ALIGN A**($i$) **WITH TEMPL**(*,$i+1$)

forces replication of array $A$ along the first dimension of the template $TEMPL$.

Algorithm 1 gives the steps in the algorithm used by our Fortran 90D/HPF compiler to process the align directives. The following example illustrates the steps and all the transformations performed to transform array indices from the *array index domain* to *template index domain* and vice versa.

Consider the Fortran 90D/HPF code fragment shown in Figure 2. There are three arrays ODD(N/2), EVEN(N/2) and NUM(N). Elements of the array ODD are aligned with odd elements of TEMPL. Similarly, elements of the array EVEN are aligned with the even elements of TEMPL. NUM is aligned identically with TEMPL. Hence, ODD and EVEN are aligned with odd and even indices of NUM respectively, because they are aligned to the same template.

**Step 1.** *Extend aligned arrays to match template size.* Note that we assume that the array size is equal to or smaller than the template size in the distributed dimension(s). If an array size is smaller than the template size in the distributed dimension, the compiler extends the array size to match the template size. For example, ODD and EVEN arrays are extended to size $N$ to match the template TEMPL's size, which is $N$. Note that an array is only extended in the distributed dimension of the array. An alternative approach such as proposed by Chatterjee et al. [12] is to

```
 1. PARAMETER(NPROC1=10, N=100)
 2. REAL NUM(N), ODD(N/2), EVEN(N/2)
 3. C$ DECOMPOSITION TEMPL(N)
 4. C$  DISTRIBUTE TEMPL(BLOCK)
 5. C$  ALIGN NUM(I) WITH TEMPL(I)
 6. C$  ALIGN ODD(I) WITH TEMPL(2*I-1)
 7. C$  ALIGN EVEN(I) WITH TEMPL(2*I)
 8.     FORALL(I=1:N:2) NUM(I) = ODD((I+1)/2)
 9.     FORALL(I=2:N:2) NUM(I) = EVEN(I/2)
10.     LOC=MAXLOC(ODD)
```

Figure 2: Example 1: A Fortran 90D/HPF program fragment involving directives, forall's and an intrinsic function.

compute and store the local index in a table. However, this introduces a level of indirection for each access. Furthermore, storing an index table also requires memory space that can be potentially as large as the distributed array itself. Many of the commercial compilers (e.g., Dec MPP Fortran [6], CM-Fortran [4], and Cray MPP Fortran [13]) extend arrays to the nearest power of two, whereas we extend in the distributed dimension to match the template size.

**Step 2.** *Determine local shape of arrays.* In this step, the compiler determines the local shape and size of the distributed arrays based on the processor grid information associated with the corresponding template. In the above example, the template TEMPL is distributed on $P$ processors. P is a compile-time parameter for each dimension of DISTRIBUTION directive. Hence, the compiler determines the size of the distributed dimension of arrays as ODD($\lceil N/P \rceil$), EVEN($\lceil N/P \rceil$) and NUM($\lceil N/P \rceil$). Since our compiler produces SPMD code, array declarations are the same in every processor.

**Step 3.** *Apply alignment functions to the aligned arrays.* In this step, all indices of each occurrence of an array (all the statements) in the input program are transformed into the *template index domain* using the alignment function $f$(I). Arrays ODD, EVEN, and NUM are associated with $f_o(I) = 2 * I - 1$, $f_e(I) = 2 * I$, $f_n(I) = I$ functions, respectively. Figure 3 illustrates this transformation on the array ODD. For example, the first forall assignment statement in Figure 2

NUM(I)=ODD((I+1)/2) is transformed into NUM(I)=ODD(2*((I+1)/2)-1) (1)

by applying the functions $f_n(I) = I$ (identity function) and $f_o(I) = 2 * I - 1$ to the *lhs* and the

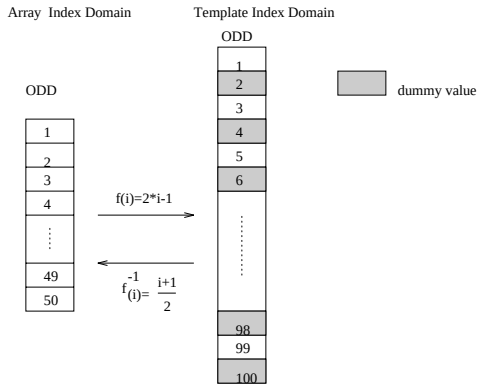



Figure 3: Transforming array ODD of the example from array index domain to template index domain.

*rhs* respectively.

**Step 4.** *Transform into canonical form*[1]. In this step, the compiler simplifies all functions applied in Step 3 by performing symbolic manipulation and partial evaluation of constants. For example, statement (1) becomes **NUM(I)=ODD(I).**

The above simplification of indices helps the compiler to choose efficient collective communication routines. Our communication detection algorithm is based on symbolically comparing the *lhs* and *rhs* reference patterns and determining if the pattern is associated with one of the collective communication routines. In the above statement the compiler compares *lhs* and *rhs* indices and determines that no communication is required, because both of the array reference patterns are given by $I$ and are aligned to the same template. However, if the *rhs* are ODD(I+2), it will be recognized as a shift communication.

**Step 5.** *Compute* $f^{-1}(i)$. For each array, we compute the inverse alignment function $f^{-1}(i)$ corresponding to each $f(i)$; $f^{-1}(i)$ is stored in the *Distributed Array Descriptor* (DAD) [14]. In DAD, for each alignment function (affine) we store constants $a$ and $b$, where $f(i) = ai + b$. This function is needed when any computation needs to be performed using the original index of an array. For example, the last statement in Figure 2 calls the intrinsic function MAXLOC to find the location of the maximum element in the array ODD. This function must be evaluated using

[1] A canonical form is a syntactic form in which variables appear in a predefined order and constants are partially evaluated.

the original array indices. The inverse function for array ODD is $f^{-1}(i) = \frac{i+1}{2}$. MAXLOC will return the location of the maximum value in the original *array index domain* by applying the $f^{-1}$ function.

# 4 Computation Partitioning

Once the data is distributed, there are several alternatives for assigning computations to processing elements (PEs) for each instance of a *forall* statement. Note that we internally transform all array statements into equivalent *forall* representations. One of the most common methods for computation assignment 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 many cases. Most of the current implementations of parallelizing compilers use the owner computes rule [5, 15]. However, it may not be possible to apply the owner computes rule for every case. The following examples describe how our compiler performs computation partitioning.

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

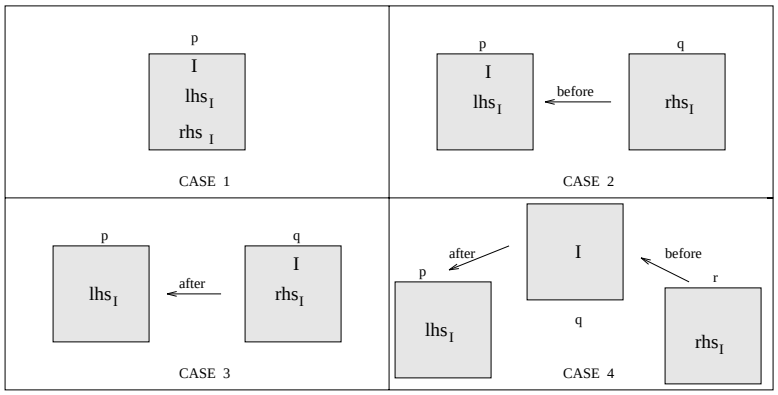
```
  forall (i=1:N, j=1:N)
&   B(i,j) = 0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
```

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.

Figure 4 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 off-processor data required to execute an iteration instance are performed before the computation is performed.

**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)
```




CASE 1: No communications

CASE 2: Communication before computation to fetch non-local rhs

CASE 3: Communication after computation to store non-local data lhs

CASE 4: Communication before and after computation to fetch and store non-locals

Figure 4: 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.

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. 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
```

# 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 [16]. 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 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.

2. *Increased portability of the Fortran 90D/HPF compiler.* Separating the communication library from the basic compiler design enhances portability because only the machine specific low-level communication calls in the library need to be changed. Note that the compiler communication library can be optimized for each machine and portability is derived from a common interface so that the code generation by the compiler does not have to change.

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 [17, 18, 11]. 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 than generating point to point communication for each pair of communicating processors (e.g., see [19]), thereby enabling the compiler to generate better distributions.

### 5.1 Communication Primitives

In order to perform a collective communication on array elements, the communication primitive needs the following information: send processors list, receive processors list, local index list of the source array, and 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, respectively.

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 [20].
- **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, [9] and 2) where to determine the send and receive lists preprocessing itself requires communication among the processors [21].

- **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 require 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.

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 [22]. Further, we extend the above tests to include unstructured communication. Table 1 shows the patterns of communication primitives used in our compiler. The details of our communication detection algorithm can be found in [7].

| 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 |

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).

## 5.2 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 appropriate program transformations. We now illustrate these transformations with the aid of two examples.

**Example 1 (multicast)** Consider the statement where $A(I, J)$, and $B(I, J)$ are aligned with $TEMPL(I, J)$.

```
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) ! compute local lb, ub, and st
2.    call set_BOUND(lb1,ub1,st1,1,M,1) ! compute local lb, ub, and st
3.    call set_DAD(B_DAD,.....)        ! put information for B into 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
```

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

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 messages. The preprocessing loop is also called the "inspector" loop [23, 9].

**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 can only be known at runtime. The receiving processors can know what non-local 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) ! compute local lb, ub, and st
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. 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 to the appropriate processors a list of required array elements (*local_list*). Each processor now has the information required to set up the communication schedule.

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 repeatedly to 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 executing it only 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 passes a pointer to the already existing schedule.

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) [23] designed to efficiently support irregular patterns of distributed array accesses. PARTI and other communication primitives and

intrinsic functions form the run-time support system of our Fortran 90D compiler.

# 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 that 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. The intrinsic functions that may induce communication can be divided into five categories as shown in Table 2.

Table 2: Fortran90D/HPF Intrinsic Functions

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

The first category requires data to be transferred using fewer overhead structured shift communications operations. The second category of intrinsic functions require computations based on local data followed by the 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 [16]. Some intrinsic functions can be further optimized for the underlying hardware architecture.

Table 3 presents a sample of performance numbers for a subset of the intrinsic functions on iPSC/860. A detailed performance study is presented in [14]. 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 the TRANSPOSE function, going from one processor to two or four actually results in an increase in the time due to communication requirements. However, for larger size multiprocessors the times decrease, as expected.

Table 3: Performance of some Fortran 90D Intrinsic Functions (time is in 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 that 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 called *distributed array descriptor* (DAD) [14].

# 7 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 [16]. *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

Table 4: 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 | Problem Size | Number of PEs | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 4 | 8 | 16 |
| Gauss Hand | 1023x1024 | 623.16 | 446.60 | 235.37 | 134.89 | 79.48 |
| Gauss F90D | 1023x1024 | 618.79 | 451.93 | 261.87 | 147.25 | 87.44 |
| Nbody Hand | 1024x1024 | 6.82 | 1.74 | 1.29. | 0.76 | 0.42 |
| Nbody F90D | 1024x1024 | 13.82 | 5.95 | 2.40 | 1.31 | 0.86 |
| Option Hand | 8192 | 4.20 | 3.14 | 1.60 | 0.83 | 0.43 |
| Option F90D | 8192 | 4.30 | 3.19 | 1.64 | 0.84 | 0.44 |
| Pi Hand | 65536 | 0.398 | 0.200 | 0.101 | 0.053 | 0.030 |
| Pi F90D | 65536 | 0.411 | 0.207 | 0.104 | 0.054 | 0.032 |

systems. Data for all programs were block distributed and were written outside of the compiler group at NPAC by experienced message passing programmers.

Tables 4 and 5 show the performance of compiler generated codes ($F90D/HPF$) and hand-written f77+ MP code. The tables contain data from running these programs with a varying number of processors on Intel iPSC/860. The compiler generated codes and hand-written codes use the Express 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).

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 an experienced programmer can incorporate more optimizations than our compiler currently does. For example, a programmer can combine or eliminate some of the communication or some of the 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.

Table 5: 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 number | Type of Application | F90D/HPF | Hand | Ratio |
|---|---|---|---|---|
| 1. | Hydrodynamics | 2.545 | 2.550 | 0.9980 |
| 2. | Incomplete Cholesky | 11.783 | 10.440 | 1.1286 |
| 3. | Inner product | 3.253 | 3.249 | 1.0012 |
| 4. | Banded linear equations | 5.139 | 3.212 | 1.600 |
| 5. | Tridiagonal elimination | 30928.6 | 30897.7 | 1.001 |
| 6. | Linear recurrence relations | 1849.1 | 1886.5 | 0.9801 |
| 7. | Equation of state | 11.346 | 3.704 | 3.0632 |
| 8. | A.D.I | 38.656 | 20.038 | 1.9291 |
| 9. | Numerical Integration | 2.255 | 2.441 | 0.9238 |
| 10. | Numerical Differentiation | 9.814 | 4.589 | 2.1386 |

# 8 Summary of Related Work

Callahan and Kennedy [24] 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 [25, 26]. 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 [27, 9] 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 [28, 29] 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 [30, 31, 21]. Saltz *et al.* describe and experimentally characterize ARF compiler and runtime support procedures that embody methods

capable of handling a wide range of irregular problems in scientific computing. Many techniques, especially the unstructured communication of the Fortran 90D/HPF compiler are adapted from the ARF compiler. The ADAPT system [32] compiles Fortran 90 for execution on MIMD distributed memory architectures. The ADAPTOR [33] is a tool that transforms data parallel programs written in Fortran with array extensions and layout directives to explicit message passing. Li and Chen [22, 34] 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 [35] describes the techniques 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.

# 9 Summary and Conclusions

Fortran 90D/HPF is a language that incorporates parallel constructs and allows users to specify data distributions. In this paper, we have presented a design for a Fortran 90D/HPF compiler for distributed memory machines. Specifically, techniques for the processing of distribution directives, computation partitioning, communication detection, and generation were presented. Our design is both efficient and portable. We presented preliminary performance results from our compiler.

We believe that the methodology presented in this paper to compile Fortran 90D/HPF can be used by the designers and implementors of the HPF compiler.


### 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. Finally, special thanks J. Saltz and his group at Maryland University for many inspiring discussions and inputs that have greatly influenced this work as well as for providing us with PARTI runtime system.

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

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

[8] 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.

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

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

[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] F. Long R. Schreiber S. Chatterjee, J.R. Gilbert and S.H Tseng. Generating Local Address and Communication Sets for Data-Parallel Programs. *The Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[13] T. MacDonald D. M. Pase and Andrew Meltzer. MPP Fortran Programming Model. Technical report, Cray Research, Inc., 1992.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] 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.

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

[24] M. Y. Wu and D. D. Gajski. A programming aid for message-passing systems. *Parallel Processing for Scientific Computing*, pages 328–332, 1989.

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

[26] 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.

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

[28] Michael Quinn, Philip Hatcher, and Karen Jourdenais. Compiling C* Programs for a Hypercube Multicomputer. *Parallel Computing Laboratory, University of New Hampshire*, PCL-87-12, December 1987.

[29] 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.

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

[31] 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.

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

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

[34] 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.

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