Compiling Distribution Directives in a Fortran 90D Compiler*

Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka
Syracuse University
Northeast Parallel Architectures Center
4-116, Center for Science and Technology
Syracuse, NY, 13244-4100
{zbozkus, choudhar, gcf, haupt, ranka}@npac.syr.edu

SCCS-388

Abstract

Data Partitioning and mapping is one of the most important steps of in writing a parallel program; especially data parallel one. Recently, Fortran D, and subsequently, High Performance Fortran (HPF) have been proposed to allow users to specify data distributions and alignments for arrays in programs. This paper presents the design of a Fortran 90D compiler that takes a Fortran 90D program as input and produces a node program + message passing calls for distributed memory machines. Specifically, we present the design of the Data Partitioning Module that processes the alignment and distribution directives and illustrate what are the important design considerations. We show that our compiler produces portable, yet an efficient code. We also present the performance of the code produced by the compiler and compare it with the performance of the hand written code. We believe, this design can be used by implementors of the HPF compilers.

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

*This work was supported in part by NSF under CCR-9110812 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.

memory counterparts in terms of cost and scalability, though it is a widely accepted that they are difficult to program given the current status of the software technology. One major reason for this difficulty is the absence of a single global address space at the architecture level. 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.

This paper presents the design of a prototype compiler for Fortran 90D. The compiler takes as input a program written in Fortran 90D which is a data parallel language with extensions for specifying data alignments and distributions [1]. Its output is a node program plus calls to a message passing library. Therefore, the user can still program using a data parallel language but is relieved of the responsibility to perform data distribution and communications

The system diagram of the Fortran 90D compiler is shown in Figure 1. Given a syntactically correct Fortran 90D program, the first step of compilation is to generate a parse tree. The partitioning module divides the program into tasks and allocates the tasks to processor elements (PEs) using the compiler directives — decomposition, alignment, and distribution. There are three ways to generate the directives: 1) users can insert them, 2) programming tools can help users to insert them, or 3) automatic compilers can generate them. In the first approach, users write programs with explicit distribution and alignment directives. A programming tool can generate useful analysis to help users decide partitioning styles, and measure performance to help users improve program partitioning interactively [2, 3, 4]. The directives can also be generated automatically by compilers. Promising work has been done along these lines [5, 6, 7, 8, 9].

The focus of this paper is to describe the design and implementation of the data partitioning module. We discuss how to distribute data and manage computations given the data distribution directives. Specifically, we show how the alignment and distribution directives can be systematically processed to produce an efficient code. Details of other modules can be found in[10, 11].

The rest of this paper is organized as follows. Section 2 briefly reviews Fortran D directives. Section 3 presents the overall design used in the data partitioning module (DPM). The design of DPM consists of three stages: preprocesing directives, data mapping and grid mapping. These stages are presented in Sections 4, 5 and 6 respectively. Section 7 summarize our early experince using the initial version of the compiler including a comparison of the performance with hand written parallel code. Section 8 presents a summary of related work. Finally, summary and conclusions are

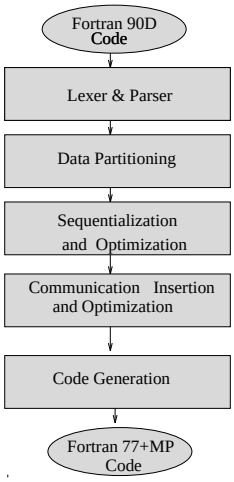

Figure 1: Overview of the Compiler Components.

presented in Section 9.

2 Fortran D Data Distribution Directives

Fortran D is the first language to provide users with explicit control over data partitioning with data *alignment* and *distribution* specifications[1]. The distribution directives can be used with Fortran 77 or Fortran 90. In this paper we consider Fortran 90. Fortran D has three compiler directives.

- DECOMPOSITION
- DISTRIBUTE
- ALIGN

The DECOMPOSITION directive is used to declare the name, dimensionality, and the size of each problem domain. A decomposition is simply an abstract problem or index domain. We call it "template" (the name "template" has been chosen to describe "DECOMPOSITION" in HPF [12]). Arrays in a program are mapped to templates using the ALIGN directive. There may be multiple templates representing different problem mappings, but an array may be aligned only to one template at any point in time. All scalars are replicated. An array not explicitly aligned to any template serves as its own template. The DISTRIBUTE directive specifies the mapping of

the template onto a logical processor grid. Each dimension of the template is distributed in a block, cyclic or irregular manner; the symbol "*" marks dimensions that are not distributed (i.e. collapsed or replicated). The selected distribution can affect the ability of the compiler to minimize communication and load imbalance in the resulting program.

The following example illustrates the Fortran D directives. Consider the data partitioning schema for matrix-vector multiplication proposed by Fox *et al.*[13] and shown in Figure 2. The matrix vector multiplication can be described as

$$y = Ax$$

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

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

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

3 Design Methodology

Fortran 90D compiler maps arrays to physical processors by using a three stage mapping as shown in Figure 3. This three stage mapping has also been proposed in HPF[12].

Stage 1 : ALIGN directives are processed to compute functions that map *array index domain* to the *template index domain* and vice versa. Also, local shape of the arrays it determined.

Stage 2 : Each dimension of the template is mapped onto the logical processor grid based on the distribution directives. Furthermore, mapping functions to generate relationship between global and local indices are computed.

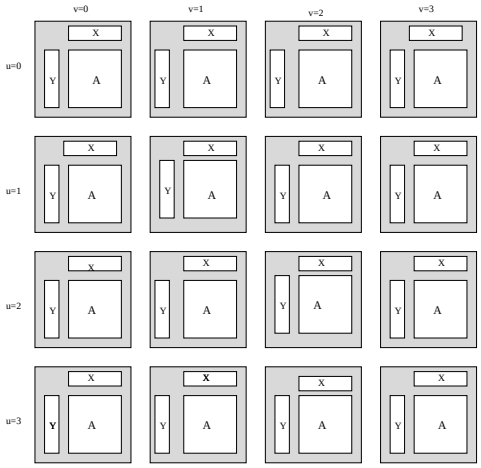

Figure 2: **Matrix-vector decomposition: each processor is assigned array section of A, X, and Y.**

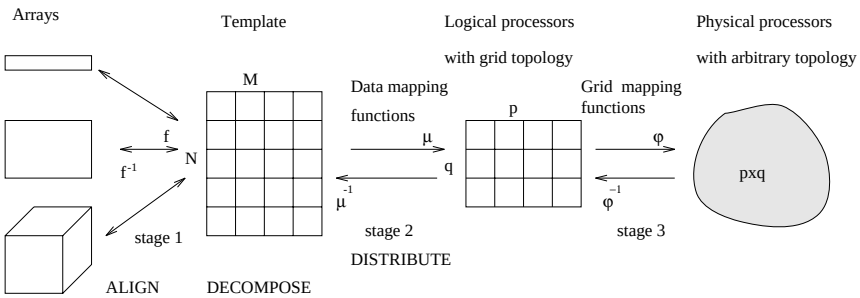

Figure 3: **Three stage array mapping**

Stage 3 : Logical processor grid is mapped onto physical system. This mapping can change from one system to another but the data mapping onto 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.

4 Compiling the ALIGN Directive (Stage 1)

Alignment of data arrays to templates is specified by the ALIGN directives. In this section, we describe how the ALIGN directive is processed in our compiler.

Each array is associated with a template. If an array is not explicitly associated with a template using an ALIGN directive, then it is assumed that it is associated with its own implicit template. In that case, the compiler can choose any distribution it determines is the most appropriate.

Alignment determines which portions of two or more arrays will be in the same processor for a particular data partitioning. Clearly, if arrays involved in the same computation are aligned in such a manner that after distribution their respective sections lie on the same processors then the number of non-local accesses would be reduced.

Alignment is a relation that specifies a one-to-one correspondence between elements of a pair of array objects. The template is defined by a *DECOMPOSITION* directive with its shape and ranks. Let A be an m-dimensional array and $TEMPL$ be an n-dimensional template. The general form of alignment directive is

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

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

Fortran 90D compiler requires that each of A's subscripts i_1, ... ,i_m appears exactly once on the right-hand side of the relation, 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 axis in the array may be different than the order of axis in the template (not necessarily $i_k = i_{a_k}$). This permits transpose style alignments such as aligning $A(I, J)$ with $TEMPL(J, I)$.

Algorithm 1 (Compiling Align directives)

Input: Fortran 90D syntax tree with some alignment functions

Output: Fortran 90D syntax tree with identical alignment functions

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 $f^{-1}(i)$.

The symbol "*" shows the replication or collapse of the corresponding dimension. It may appear in both the array and the template subscripts. The array rank (the number of dimension) m may be different than the rank of template, n. For example, the directive

C$ ALIGN A(i,*) WITH TEMPL($i+1$). requests the second dimension of the array A be collapsed, while the directive

C$ ALIGN A(i) WITH TEMPL(*,$i+1$). forces replication of array A along the first dimension of the template $TEMPL$.

The *alignment function* f_k is required to be a linear function $f_k = s_k * i_{a_k} + o_k$ or $f_k = o_k$. The parameters i_{a_k}, s_k, and o_k correspond to the three components of the alignment function: *axis*, *stride*, and *offset*. Misalignment in the axis or stride components causes *irregular communication*, and misalignment in the offset component causes nearest-neighbor communication [5].

Algorithm 1 gives the steps in the algorithm used by our Fortran 90D 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 code fragment shown in Figure 4. There are three arrays ODD(N/2), EVEN(N/2) and NUM(N). Elements of the array ODD are aligned with odd elements of TEMP. Similarly, elements of the array EVEN are aligned with the even elements of TEMPL. NUM is

```
 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 4: **Example 1: A Fortran 90D program fragment involving directives, forall's and intrinsic function.**

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.

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. 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 *Single Program Multiply Data* (SPMD) code, array declaration is 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 is 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 5 illustrate this transformation on the array ODD. For example, the first forall assignment statement in Figure 4

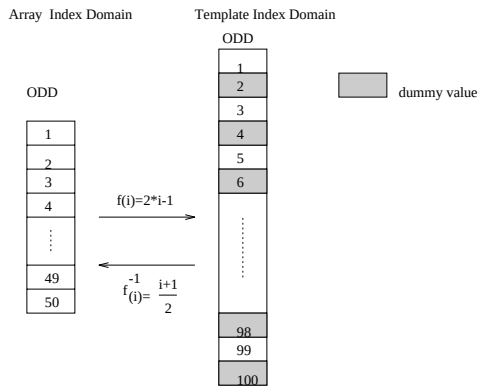

Figure 5: **Transforming array ODD of the example from array index domain to template index domain.**

```
NUM(I)=ODD((I+1)/2)
```

is transformed into

```
NUM(I)=ODD(2*((I+1)/2)-1)  (1)
```

by applying function $f_n(I) = I$ (identical function) and $f_o(I) = 2 * I - 1$ to *lhs* and *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, the statement (1) becomes

```
NUM(I)=ODD(I).
```

The above simplification of indices helps compiler in choosing efficient collective communication routines. Our communication detection algorithm [14, 11] 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 the array reference patterns are given

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

```
 1. PARAMETER(NPROC1=10, N=100)
 2. REAL NUM(10), ODD(10), EVEN(10)   ! local shapes
 3. call set_BOUND(lb,ub,st,1,100,2)  ! compute local lb, ub, st
 4. DO I=lb,ub,st
 5.  NUM(I) = ODD(I)                  ! local computations
 6. END DO
 7. call set_BOUND(lb,ub,st,2,100,2)
 8. DO I=lb,ub,st
 9.  NUM(I) = EVEN(I)                 ! local computations
10. END DO
11. call set_DAD(ODD_DAD,....)        ! put information for ODD into ODD_DAD
12. LOC=MAXLOC(ODD, ODD_DAD)          ! MAXLOC is implemented on f77+MP
```

Figure 6: **The compiler generated Fortran 77+MP code.**

by I and aligned to the same template. However, if *rhs* was ODD(I+2), it will recognize it 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 *Distributed Array Descriptor* (DAD) [15]. 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 4 calls the intrinsic function MAXLOC to find the location of the maximum element in the array ODD. This function must be evaluated using the original array indices. The inverse function for array ODD is $f^{-1}(i) = \frac{i+1}{2}$. MAXLOC will return the location of maximum value in the original *array index domain* by applying f^{-1} function.

Figure 6 shows the compiler generated Fortran 77+MP code for the Fortran 90D code given in Figure 4.

We emphasize that the above transformation from the *array index domain* to the *template index domain* has two advantages.

1-) This allows us to easily detect regular collective communication patterns among arrays aligned to the same template.

2-) We need to keep data distribution functions only for the template and not for all the arrays aligned to the template.

5 Data Distribution (Stage 2)

In this section, we describes how the Fortran 90D compiler distributes the template on the logical processor grid (Figure 3). In this phase, the compiler uses information provided by the DISTRIBUTE directives.

The DISTRIBUTE directives assigns an *attribute* to each dimension of the template. Each attribute describes the mapping of the data in that dimension of the template on the logical processor grid. Attributes in each dimension are independent, and may specify regular or irregular distributions. For example the following directive

```
C$     DISTRIBUTE TEMPL(BLOCK,CYCLIC)
```

distributes the template TEMPL blockwise in the first dimension and cyclicly in the second dimension.

The first version of the compiler supports the following types of distribution.

- **BLOCK** divides the template into contiguous chunks.
- **CYCLIC** specifies a round-robin division of the template.
- **IRREGULAR** specifies to the compiler that the needed data distribution functions are provided by the user.

5.1 Distribution functions

A Fortran 90D program is written in the global name space. Therefore, the arrays and template indices refer to indices in the global name space. Parallelizing the program onto a distributed memory machine requires mapping a global index onto the pair: processor number and local index because on a Distributed Memory Machine, each node has a separate name space. For the above index transformations, we define data-distribution functions (index-conversion functions) as given in Definition 1 below.

Definition 1: *A data-distribution function for each dimension of template μ maps three integers, $\mu(I, P, N) \rightarrow (p, i)$, where I is the global index, $0 \leq I < N$, P is the number of processors, and N is the size of global index. The pair (p, i) represents the processor p, $(0 \leq p < P)$ and i is the local index of p $(0 \leq i < \mu^{\#}(p, P, N))$. $\mu^{\#}(p, P, N)$ gives the cardinality (the number of global*

indices in processor p). The inverse distribution function $\mu^{-1}(p, i, P, N) \rightarrow I$ *transforms the local index* i *in processor* p *back into global index* I.

The term *global index* will be used to refer to the index of a data item within the global array (global name space) while the term *local index* will denote the index of a data item within a logical processor.

The choice of these distribution functions is one of the most important design choices. We use the following criteria:

- calculation of these function at run-time must be efficient, and
- distribution functions should yield a good static load balance.

The **BLOCK** attribute indicates that blocks of global indices are mapped to the same processor. The block size depends on the size of the template dimension, N, and the number of processors, P, on which that dimension is distributed. Our Fortran 90D compiler provides two different block attributes, **BLOCK1** and **BLOCK2**.

The **BLOCK1** attribute assumes that N is divisible by P. This results in a very simple and efficient distribution function as shown in the first column of Table 1. Each processor has N/P global indices which provides on optimum load balance. If N is not divisible by P, the compiler extends N such that N becomes divisible by P. This may cause a load imbalance because the tail processors may have less number of array items than do the beginning processors.

The **BLOCK2** attribute is used to distribute block of N global indices on P processor as evenly as possible. This distribution function is shown in the second column of Table 1. The first $N \ mod \ P$ processors receive $\lceil \frac{N}{P} \rceil$ elements; the rest get $\lfloor \frac{N}{P} \rfloor$ elements. Therefore, the difference between the number of elements assigned to any two processors will be at most one in each dimension. Table 2 shows the distribution of global index for different data sizes on three processors. This distribution functions yields an optimal static load balance. This type of block distribution has been used in the implementation of TOOLBOX [16].

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

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

	Block1-distribution	Block2-distribution	Cyclic-distribution
global to proc $I \rightarrow p$	$p = \frac{I*P}{N}$	$p = max(\lfloor \frac{I}{\lfloor \frac{N}{P} \rfloor + 1} \rfloor, \lfloor \frac{I - N \ mod \ P}{\lfloor \frac{N}{P} \rfloor} \rfloor)$	$p = I \ mod \ P$
global to local $I \rightarrow i$	$i = I - \frac{p*N}{P}$	$i = I - p * \lfloor \frac{N}{P} \rfloor - min(p, N \ mod \ P)$	$i = \lfloor \frac{I}{P} \rfloor$
local to global $(p, i) \rightarrow I$	$I = i + \frac{p*N}{P}$	$I = i + p * \lfloor \frac{N}{P} \rfloor + min(p, N \ mod \ P)$	$I = iP + p$
cardinality	$\frac{N}{P}$	$\lfloor \frac{N+P-1-p}{P} \rfloor$	$\lfloor \frac{N+P-1-p}{P} \rfloor$

Table 2: **BLOCK2 distribution of global index with different data sizes on 3 processors**

Data Size	Proc. 0	Proc. 1	Proc. 2
3	1:1	2:2	3:3
8	1:3	4:6	7:8
100	1:34	35:67	68:100

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

The **IRREGULAR** attribute indicates that the required data distribution function will be provide by the user. The compiler generates in-line codes for the block and cyclic data distribution functions. But for irregular distributions, it calls the user-defined functions from inside the generated code. In generally, these functions are represented as arrays ("MAP" arrays) and return values from those arrays[1]. Note that the MAP arrays themselves can be distributed or replicated. For example, in PARTI developed by Joel Saltz [17], MAP arrays are distributed. In our current version, we had incorporated the schedule, gather and scatter primitives from PARTI, but we are only supporting replication of MAP arrays.

Table 3 shows the performance of computing the data distribution functions of Table 1 on an Intel i860 processor. As expected, the Cyclic distribution functions performs the best and the Block2 the worst.

Table 3: **Computation times for data distribution functions on an Intel i860 processor (time in microseconds).**

	Block1-distribution	Block2-distribution	Cyclic-distribution
global to proc	1.8	7.6	1.9
global to local	1.9	3.6	1.6
local to global	1.8	3.6	0.4

5.2 Usage of the data distribution functions

The following examples illustrate how the data distribution function can be used for various constructs. For these examples, the array A has the following alignment.

```
C$ DECOMPOSITION TEMPL(N,M)
C$ ALIGN A(I,J) WITH TEMPL(I,J)
C$ DISTRIBUTE TEMPL(CYCLIC,BLOCK1)
```

and TEMPL is distributed on a two-dimensional PxQ processor grid.

Example 1 (Masking) Consider the statement

```
A(5,8)=99.0
```

The owner processor of the array element $A(5,8)$ executes the statement. Since the compiler generates SPMD style code, it masks the rest of the processors:

```
if( 5 mod P .eq. my_id(1) .and. 8*Q/M .eq. my_id(2))
     A(5/P, 8-my_id(2)*M/Q) = 99.0
```

Where my_id(1) and my_id(2) describes the processor's position in the two-dimensional logical grid. In this case, the compiler uses the global to processor and global to local functions for cyclic and block1 distributions. The processors are masked according to the coordinate id numbers since the logical processors are arranged in a grid topology.

Example 2 (Grouping) Consider the statement

```
A(:,8)=99.0
```

The group of processors owning the 8th column of array A only need to execute the statement. The rest of the grid must be masked.

```
do i=my_id(1),N,P
   if(8*Q/M .eq. my_id(2))  A(i/P, 8-my_id(2)*M/Q) = 99.0
end do
```

Note that the iterations (indexed by i above) are distributed cyclicly following the owner computes rule.

Example 3 (Forall) Consider the statement

```
forall(i=1:N,j=1:M)  A(i,j)=j
```

In the above computations, all elements of each column of array A are assigned the corresponding column number (in the global index domain).

```
do i=my_id(1),N,P
do j=1,M/Q
  A(i/P,j)=j+my_id(2)*M/P
end do
end do
```

The compiler distributes the iterations i and j in cyclic and block fashion respectively since array A is distributed in that fashion. Iteration index j is localized. The compiler transforms j back to global index by using local to global index conversion in *rhs* expression.

Example 4 (Broadcast) Consider the statement

```
x=A(5,8)
```

where x is a scalar variable (scalars are replicated on all processors). The above statement causes a broadcast communication. The source processor of the broadcast is found by using a global to processor function similar to that in example 1.

Example 5 (Gather) Consider the statement

```
B=A(U,V)
```

where U and V are one-dimensional replicated arrays. B is a two-dimensional array and is distributed in the same way as is array A. This vector-valued assignment causes an unstructured communication (also called *gather*[17] or *random-read*[18] in this case). The owner processors of array B needs some values of array A depending on the contents of arrays U and V at run-time. The compiler makes each owner processor of the array B calculate which processor has the non-local part of the array A by using global to processor function. The compiler also generates code that computes the local index in the array A using the global to local index conversion function for each source processor. After making each processor calculate the local list and the processor list, the compiler generates the statement to call gather collective communication.

Example 6 (Scatter) Consider the statement

```
A(U,V)=B
```

The above statement causes *scatter* or *random-write* kind of communication. Again the compiler generates code such that each owner processor of the array B uses data distribution functions to find the destination of the local array B.

6 Grid Mapping Functions (Stage 3)

So far we have presented techniques used in our compiler that map data onto logical processors. In this section we describe the mapping of logical processors onto physical processors.

There are several advantages of decoupling logical processors from physical system configurations. These advantages include *locality*, *portability* and *grouping*.

Locality: Multiple accesses to consecutive memory locations is called *spatial locality*. *Spatial locality* is very important for Distributed Memory Machines. Arrays representing spatial locations

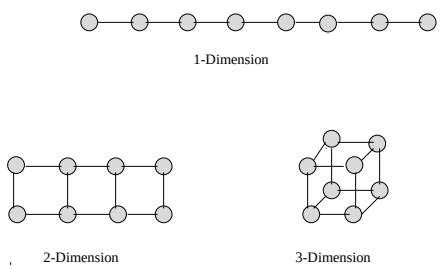

Figure 7: **Logical processor topologies**

are distributed across the parallel computer. For instance, it makes sense to have data distributed in such a way that processors that need to communicate frequently are neighbors in the hardware topology. It has been shown that this is extremely important in the common regular problems in scientific applications such as relaxation [13]. Our template is a d-dimensional mesh. If this template is BLOCK distributed on a d-dimension grid of processors, the neighboring array elements (spatial locality) will be in the neighboring processors. The grid topology is a very good topology for spatial locality. Fortran 90D makes logical processor topology grid according to the number of dimensions of the template as shown in Figure 7.

Portability: The physical topology of the system may be a grid, tree, hypercube etc. The mapping for the best (possible) grid topology changes from one physical topology to another. To enhance portability of our compiler, we separate the physical and logical topologies. Therefore, porting the compiler from one hardware platform to another involves changing the functions that map the logical grid topology to the target hardware.

Grouping: Operations on a subset of dimensions in arrays are very common in scientific programming, e.g., row and column operations on matrices. Fortran 90 provides intrinsic functions such as **SPREAD**, **SUM**, **MAXVAL** and **CSHIFT** that let a user to specify operations along different dimensions by specifying the **DIM** parameter. These dimensional operations conceptually group elements in the same dimension. The dimensional array operations result in "dimensional array communications". We have designed a set of collective communication routines that operate along one or more dimensions (groups of processors) of the grid. For example, we have developed spread (broadcast along dimension), shift along dimensions and concatenate communications. The usage of these primitives is discussed in [11].

```
int gridinit(dim, num(*))
int gridproc(coord(*))
int gridcoord(proc, coord(*))
```

Figure 8: **The prototype of grid-mapping functions**

The performance of the resulting code may be adversely affected if the logical grid to physical system mapping is not efficient. Therefore, one of the goals of these mapping functions is to map nearby processors in the logical grid to physically close processors in the machine architecture.

Definition 2: *A logical processor grid consists of d dimensions,* $(P_0, P_2, ..., P_i, ..., P_{d-1})$*, where* P_i*,* $0 \leq i < d$ *is the size of the* i^{th} *dimension. A processor grid mapping function,* φ*, maps a processor index in the d-dimensional space,* $\varphi(v_0, v_1, ..., v_{d-1}) \rightarrow p$ *where* $0 \leq v_i < P_i$ *(i.e.,* v_i *is the index of the logical processor in the* i^{th} *dimension), and p is the physical processor number, (*$0 \leq p < \prod_{i=0}^{d-1} P_i$*). The inverse mapping function* $\varphi^{-1}(p) \rightarrow (v_0, v_1, ..., v_{d-1})$ *transform the processor number p back into logical grid number.*

The grid mapping function φ and φ^{-1} for hypercube using Gray Code can be found in [13]. The grid mapping onto a fat tree can be found in [19].

Figure 8 gives some of grid mapping function implemented in our compiler. The first routine, **gridinit**, takes the dimensionality of the grid, *dim*, and the number of physical processors in each dimension as an array, *num* and performs the necessary initializations in order to use the other two grid mapping functions φ and φ^{-1}. The routine **gridcoord** implements the function φ to generate the physical processor number corresponding to the logical processor grid specified in the parameter array "coord(*)". Similarly, the routine **gridproc** implements the function φ^{-1}. Its input parameter "proc" specifies the physical processor id and its output is the corresponding index in the logical grid which is stored in the array "coord(*)". The details of these functions can be found in[13].

The usage of these function is to enhance portability. The compiler generates all the communication calls based on the logical coordinates of the processors. The communication routines in turn use the above functions to compute the physical processor ids of involved processors. Another important point to note is that by using the logical grid at compiler level, masking and grouping

are performed by using logical grid coordinates.

7 Discussion

Although this paper is focused on the partitioning module of the compiler, a prototype compiler is complete (it was demonstrated at Supercomputing'92). In this section, we describe our experience on using our compiler.

7.1 Portability of the Fortran 90D Compiler

One of the principal requirements of the users of distributed memory MIMD systems is some "guarantee" of the portability for their code. This was realized early on by the Caltech Concurrent Computation Group led by Geoffrey Fox. Parasoft's Express parallel programming environment represents a commercialization of these ideas. One feature of Express is the portability on various platforms including, Intel iPSC/860, nCUBE/2, networks of workstations etc. We should emphasize that we have implemented a collective communication library which is currently built on the top of Express message passing primitives. Hence, in order to change to any other message passing system such as PVM [20] (which also runs on several platforms), we only need to replace the calls to the communication primitives in our communication library (not the compiler). However, it should be noted that a penalty must be paid to achieve portability because portable routines are normally built on top of the system routines.

As a test application we use is Gaussian Elimination, which is a part of the FortranD/HPF benchmark test suite [21]. Figure 9 shows the execution times obtained to run the same compiler generated code on a 16-node Intel/860 and nCUBE/2 for various problem sizes.

7.2 Performance Evaluation

Table 4 shows a comparison between the performance of the hand-written Fortran 77+MP code with that of the compiler generated code. We can observe that the performance of the compiler generated is within 10% of the hand-written code. This is due to the fact that the compiler generated code produces an extra communication call that can be eliminated using optimizations. However as Figure 10 shows, the gap between the performance of the two codes increases as the number of processors increases. This is because the extra communication step is a broadcast which

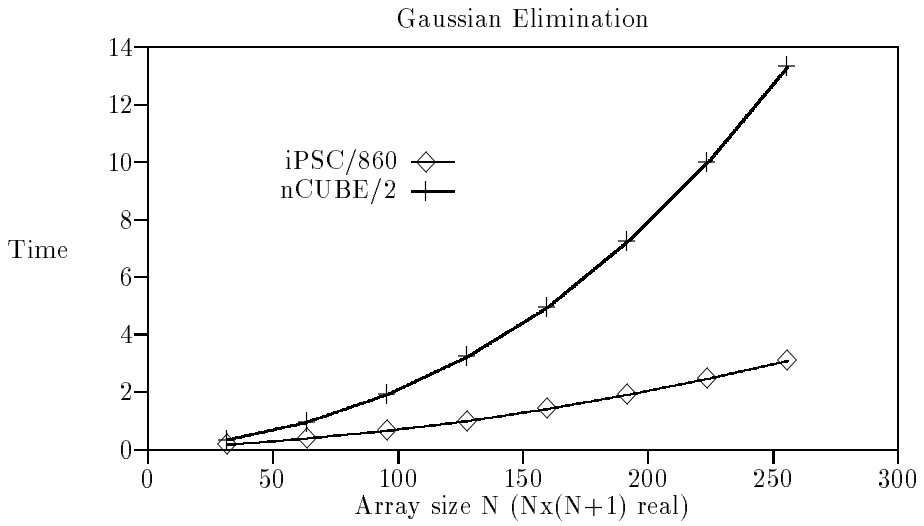

Figure 9: **Execution time of Fortran 90D compiler generated code for Gaussian Elimination on a 16-node Intel iPSC/860 and nCUBE/2 (time in seconds).**

is almost $O(log(P))$ for a P processor hypercube system. We are currently incorporating several optimizations in the compiler.

7.3 An Experiment with Distributions

A data distribution that minimizes the communication requirements for one application does not necessarily do the same for another applications since different applications may have different reference patterns.

An advantage of being able to specify different distribution directives is the ability to experiment with various distributions without extensive recoding. Hence, with a few experiments a user can choose one of the best distributions for his/her application. We illustrate the above using an example of 1-D FFT.

FFT is a difficult algorithm to compile efficiently because of the butterfly communication requirements for an efficient implementation. Automatic recognition of such patterns is difficult. Our compiler uses unstructured communication to implement such communication patterns.

We use FFT to illustrate the difference in performance when data distribution is changed.

Table 4: **Comparison of the execution times of the hand-written code and Fortran 90D compiler generated code for Gaussian Elimination. Matrix size is 1023x1024 and it is column distributed.(Intel iPSC/860, time in seconds).**

Processors	Hand Written	Fortran 90D Compiler
1	623.16	618.79
2	446.60	451.93
4	235.37	261.87
8	134.89	147.25
16	79.48	87.44

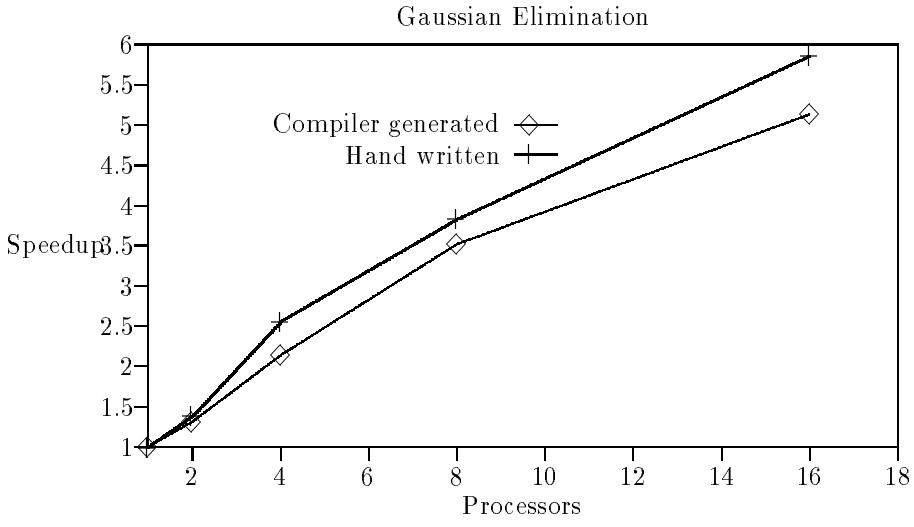

Figure 10: **Speed-up comparison (corresponds to Table 4 of the hand-written code and Fortran 90D compiler generated code for Gaussian Elimination.**

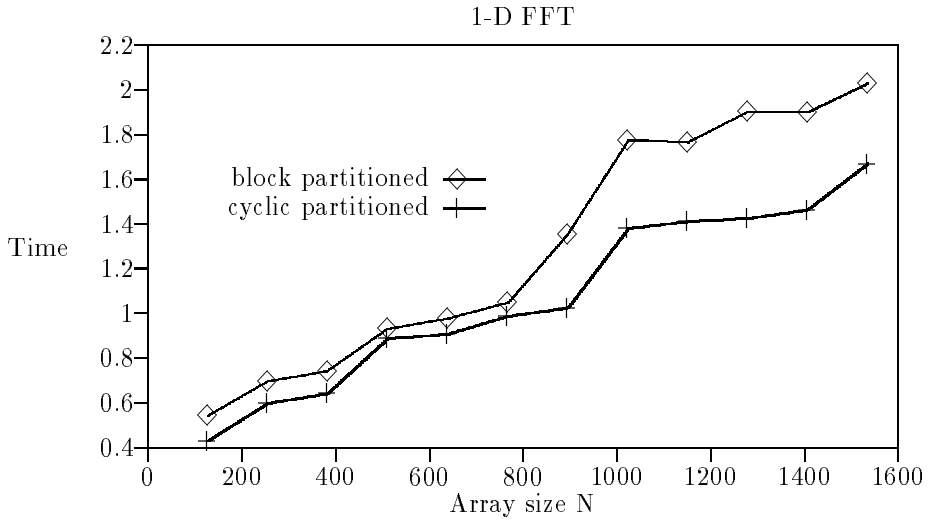

Figure 11: **Comparison of Fortran 90D compiler generated BLOCK and CYCLIC distribution of 1-Dimensional FFT codes on a 16 node Intel iPSC/860(time in seconds).**

Figure 11 shows the performance of the FFT for block (block1) and cyclic distributions. Cyclic distribution performs better due to two main reasons. One, it cyclicly distributes data, and hence, far off elements are stored in closer processors. This reduces the communication requirements compared to that when block distribution is used[22]. Second, for unstructured communication, the destination processors and locations must be calculated using the distribution functions (see Section 5). The overhead of computing these functions is the least for cyclic distribution (see Table 3) resulting in less overhead in the generated FFT code.

8 Summary of Related Work

The compilation technique of Fortran 77 for distributed memory systems has been addressed by Callahan and Kennedy [23]. Currently, a Fortran 77D compiler is being developed at Rice [24]. Superb[25] compiles a Fortran 77 program into a semantically equivalent parallel SUPRENUM multiprocessor. Kali[26] implementation puts a great deal of effort on run-time analysis for optimizing message passing. Quinn *et al.*[18] uses a data parallel approach for compiling C* to hypercube machines. The ADAPT system[27] compiles Fortran 90 for execution on MIMD distributed mem-

ory architectures. The ADAPTOR[28] is a tool that transform data parallel programs written in Fortran with array extension and layout directives to explicit message passing. Chen[29] describes a general compiler optimization techniques that reduces communication overhead for Fortran-90 implementation on massivelly parallel machines.

Due to space limitations, we do not elaborate on various related projects.

9 Conclusions

In this paper we presented a design of a prototype Fortran 90D compiler. Specifically, we presented in detail how the data distribution directives can be processed and what are the design choices for the data partitioning modules. The compiler prototype is working although new features are being added continuously. We also presented the performance of the code generated by the compiler for two applications. We believe that the performance will improve as optimizations are incorporated in the compiler. We showed that our design produces portable, yet an efficient code.

This design can be used by compiler developers for High-Performance Fortran (HPF). The only change required to process the data distribution directives is to change the syntax of the directives to that of HPF syntax. We are currently implementing that change. We believe that our design of the data partitioning module (DPM) can be used by implementors of HPF compiler to implement the DPM in their compiler.

We are currently investigating various optimization techniques for computation partitioning and communication detection. Furthermore, we are investigating automatic data partitioning and alignment techniques. Note that it is important to implement these techniques for temporary arrays even if the distributions are specified for the source arrays. Finally, we are currently involved in implementing dynamic alignment and redistribution of arrays in our compiler.

Acknowledgments

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 Min-You Wu of SUNY, Buffalo for his contributions.

References

[1] 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.

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

[3] V. Balasundaram, G. C. Fox, K. Kennedy, and U. Kremer. An Interactive Environment for Data Partitioning and Distribution. In *Fifth Distributed Memory Compu. Conf.*, Apr 1990.

[4] B. Chapman, H. Herbeck, and H. Zima. Automatic Support for Data Distribution. *IEEE: Transaction on Computers*, pages 51–57, 1991.

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

[6] 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.

[7] 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.

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

[9] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE: Transaction on Parallel and Distributed Systems*, pages 472–482, October 1991.

[10] M. Wu, G. Fox, Z. Bozkus, A. Choudhary, and S. Ranka. Compiling Fortran 90 programs for distributed memory MIMD paralelel computers. Technical Report SCCS-88, Northeast Parallel Architectures Center, May 1991.

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

[12] HIGH PERFORMANCE FORTRAN FORM. High performance fortran language specification version 0.4. *Draft, Also available as technical report CRPC-TR92225 from the Center for Research on Parallel Computation, Rice University.*, Nov. 1992.

[13] 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.

[14] 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.

[15] 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.

[16] A. Skjellum and C. H. Baldwin. The Multicomputer Toolbox: Scalable Parallel Libraries for Large-Scale Concurrent Applications. Technical Report UCRL-JC-109251, Lawrence Livermore National Laboratory, December 1991.

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

[18] 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.

[19] Z. Bozkus, S. Ranka, and G. C. Fox. Benchmarking the CM-5 multicomputer . *The Fourth Symposium on the Frointiers of Massively Parallel Computation*, 1992.

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

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

[22] S. L. Johnsson. Performance Modeling of Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, pages 300–312, August 1991.

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

[24] 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.

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

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

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

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

[29] 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.