Compiling Distribution Directives in a Fortran 90D Compiler*

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

Abstract

Data Partitioning and mapping is one of the most important steps of writing a parallel program; especially a 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 the arrays in their programs. This paper presents the design of the Data Partitioning Module of Fortran 90D compiler that processes the alignment and distribution directives.

1 Introduction

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

The focus of this paper is to describe the design and implementation of the data partitioning module in Fortran 90D compiler. 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[1, 2].

2 Fortran D Data Distribution Directives

Fortran D provides users with explicit control over data partitioning with data alignment and distribution specifications[3]. 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, and 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[4]). 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.

3 Design Methodology

Fortran 90D compiler maps arrays to physical processors by using a three stage mapping as shown in Figure 1. This three stage mapping has also been proposed in HPF[4].

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 is 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.

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.

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

$$\text{CS ALIGN } A(i_1[*], \ldots, i_m[*]) \text{ WITH } TEMPL(f_1(i_{a1}[*]), \ldots, f_n(i_{an}[*])[*]).$$

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 $i_1, \ldots, i_m$ 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 $TEMPL(k,l)$.

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

$$\text{CS ALIGN } A(i,*) \text{ WITH } TEMPL(i+1).$$

requests the second dimension of the array $A$ be collapsed, while the directive

$$\text{CS ALIGN } A(i.*) \text{ WITH } TEMPL(*,i+1).$$

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 $f_k = s_k \times i_{a_k} + o_k$ or $f_k = o_k$. The parameters $i_{a_k}, s_k, o_k$ 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].

5 Data Distribution (Stage 2)

In this section, we describe how the Fortran 90D compiler distributes the template on the logical processor grid (Figure 1). 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. For example the following directive

$$\text{CS DISTRIBUTE TEMPL(BLOCK,CYCLIC)}$$

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

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 \( \mu \) maps three integers, \( \mu(I, P, N) \rightarrow (p, i) \), where \( I \) is the global index, \( 0 \leq I \leq N \), \( P \) 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 **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. This results in a very simple and efficient distribution function as shown in the first column of Table 1.

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

|                | Block-distribution | Cyclic-distribution |
|----------------|--------------------|---------------------|
| global to proc | \( p = \frac{IL}{P} \) | \( p = I \mod P \)  |
| global to local| \( i = I - \frac{IP}{P} \) | \( i = \left\lfloor \frac{i}{P} \right\rfloor \) |
| local to global| \( I = i + \frac{IP}{P} \) | \( I = iP + p \)    |
| cardinality    | \( \frac{N}{P} \)   | \( \frac{N-IP}{P-1-x} \) |

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 \( \left\lfloor \frac{N}{P} \right\rfloor \) elements; the rest get \( \frac{N}{P} \) elements. In addition, these distribution functions are efficient and simple to compute. Although cyclic distribution functions provided a good static load balance, the locality is worse than that using block distributions because cyclic distributions scatter data.

### 6 Grid Mapping Functions (Stage 3)

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 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 [6]. 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.

**Portability:** The physical topology of the system may be a grid, a tree, a 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 90D 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 [2].
7 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.

Figure 2 shows the performance of the FFT for block 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[7]. Second, for unstructured communication, the destination processors and locations must be calculated using the distribution functions. The overhead of computing these functions is the least for cyclic distribution resulting in less overhead in the generated FFT code.

Acknowledgments

We would like to thank the other members of our compiler research group R. Bordawekar, R. Ponnumary, 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] Z. Bozkus et al. Compiling Distribution Directives in a Fortran 90D Compiler. Technical Report SCCS-388, Northeast Parallel Architectures Center, July 1992.

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

[3] 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.

[4] HIGH PERFORMANCE FORTRAN FORM. High performance fortran language specification version 0.A. Draft. Also available as technical report CRPC-TR92225 from the Center for Research on Parallel Computation, Rice University., Nov. 1992.

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

[7] S. L. Johnsson. Performance Modeling of Distributed Memory Architectures. Journal of Paral-allel and Distributed Computing, pages 300–312, August 1991.