Fortran D Language Specification

Geoffrey Fox
Seema Hiranandani
Ken Kennedy
Charles Koelbel
Ulrich Kremer
Chau-Wen Tseng
Min-You Wu

CRPC-TR 90079
December 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

FORTRAN D Language Specification[‡]

Geoffrey Fox[*]
Seema Hiranandani[†]
Ken Kennedy[†]
Charles Koelbel[†]
Ulrich Kremer[†]
Chau-Wen Tseng[†]
Min-You Wu[*]

October 17, 1993

Abstract

This paper presents FORTRAN D, a version of FORTRAN enhanced with data decomposition specifications. It is designed to support two fundamental stages of writing a data-parallel program: *problem mapping* using sophisticated array alignments, and *machine mapping* through a rich set of data distribution functions. We believe that FORTRAN D provides a simple machine-independent programming model for most numerical computations. We intend to evaluate its usefulness for both programmers and advanced compilers on a variety of parallel architectures.

1 Introduction

Recent advances in large-scale parallel processing technology has made much computing power available for today's computation scientists and engineers. However, the usefulness of parallel processing has been limited by the difficulty of programming the great variety of rapid evolving parallel architectures. A major goal for the Center for Research on Parallel Computation (CRPC) is to develop a machine-independent parallel programming model usable for both shared and distributed-memory SIMD/MIMD architectures.

High-level parallel languages such as Linda [CG89], Strand [FT90, FO90], and Delirium [LS91] are valuable when used to *coordinate* coarse-grained functional parallelism. However, these languages do not meet the needs of computational scientists because they are inefficient for capturing fine-grain data parallelism (of the type described by Hillis and Steele [HS86] and Karp [Kar87]). This is mainly due to the fact that existing parallel languages lack both language and compiler support to assist in efficient data placement [PB90]. Parallelism must also be explicitly specified because these languages do not provide compilers that can automatically detect and exploit parallelism.

[*]NPAC, Syracuse University, Syracuse NY 13244

[†]Department of Computer Science, Rice University, Houston TX 77251

[‡]This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center.

To overcome this deficiency, we have designed Fortran D, a version of Fortran enhanced with a rich set of data decomposition specifications. Fortran D is targeted at data-parallel numeric applications that are not supported by existing parallel languages. The extensions proposed in Fortran D are compatible with both Fortran 77 and Fortran 90, a version of Fortran with explicit manipulation of high-level array structures. Fortran 90D can be viewed as a refinement of CM Fortran [TMC89] consistent with a parallel Fortran 77.

We believe that Fortran D is powerful enough to express most fine-grain parallel computations, but also simple enough that a sophisticated compiler can produce efficient programs for different parallel architectures. In particular, Fortran D is well suited for supporting compiler techniques for automatic data decomposition and communication generation, two crucial problems in programming distributed-memory machines. Fortran D programs also have the advantage of being deterministic, unlike programs written in most explicitly parallel languages.

We are in the process of implementing a Fortran 77D compiler [HKT91b] in the context of the ParaScope programming environment [CCH$^+$88, BKK$^+$89]. We chose as our first target the Intel iPSC/860, a MIMD distributed-memory machine. A later project will produce a Fortran 77D compiler for a SIMD distributed-memory machine. We have also started a Fortran 90D compiler aimed at the iPSC/860 and NCUBE-2 hypercube [WF91]. We plan to compare how closely Fortran D compilers can approach the performance of hand-coded programs, and use our experiences to evaluate the usefulness of Fortran D for data-parallel programming.

The rest of this paper presents the design of Fortran D, especially its strategy for expressing data parallelism and mapping it to the underlying parallel architecture. We also compare Fortran D with related research in parallel languages, data decompositions, and compiler techniques. We start by describing our view of data parallelism.

2 Data Parallelism

The data decomposition problem can be approached by noting that there are two levels of parallelism in data-parallel applications. First, there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. We call this the *problem mapping* induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program, and is largely independent of any machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays

Second, there is the question of how arrays should be *distributed* onto the actual parallel machine. We call this the *machine mapping* caused by translating the problem onto the finite resources of the machine. It is dependent on the topology, communication mechanisms, size of local memory, and number of processors in the underlying machine. Data distribution provides opportunities to reduce data movement, but must also maintain load balance. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

Fortran D requires the user to specify data decompositions in terms of these two levels of data parallelism. First, the `ALIGN` statement is used to describe a problem mapping. Second, the `DISTRIBUTE` statement is used to map the problem and its associated arrays to the physical machine. We believe that our two phase strategy for specifying data decomposition is natural for the computational scientist, and is also conducive to modular, portable code. Previous projects also include a third intermediate level of parallelism representing a coarse-grain "virtual machine". We do not think this is necessary for our work, although it may be helpful for explicit message-passing programs.

3 Problem Mapping

In FORTRAN D, the `DECOMPOSITION` statement is used to declare a name for each problem mapping. Arrays in the program are mapped to the decomposition with the `ALIGN` statement. The result represents an abstract high level specification of the fine-grain parallelism of a problem. There may be multiple decompositions representing different problem mappings, but an array may be mapped to only one decomposition at a time. All scalars and arrays not mapped to a decomposition are allocated locally.

3.1 DECOMPOSITION Statement

Declaring a Decomposition

The decomposition statement declares the name, dimensionality, and size of a decomposition for later use. A decomposition is simply an abstract problem or index domain. No storage is allocated for a decomposition.

```
DECOMPOSITION A(N)
DECOMPOSITION B(N,N)
```

In this example, A is declared as an one-dimensional decomposition of size N, with elements indexed from 1 to N. B is a two-dimensional N by N decomposition.

3.2 ALIGN Statement

The `ALIGN` statement is used to map arrays with respect to a decomposition. Arrays mapped to the same decomposition are automatically aligned with each other. Alignment can take place either within or between dimensions.

The alignment of arrays to decompositions is specified by placeholders in the subscript expressions of both the array and decomposition. I, J, K, etc... are canonical placeholders indicating the location of dimensions in a decomposition. Array subscripts are fixed; they always consist of the placeholders in alphabetical order beginning with I. The decomposition subscripts can be functions of the placeholders; they specify the alignment of the array with respect to the decomposition.

Exact Match

The simplest alignment occurs when the array is exactly mapped onto the decomposition. In the following example, the arrays X1 and X2 are mapped exactly onto the equivalent dimensions in the decompositions A and B.

```
REAL X1(N), X2(N,N), X3(N,N)
DECOMPOSITION A(N), B(N,N)
ALIGN X1(I) with A(I)
ALIGN X2(I,J) with B(I,J)
ALIGN X3(I,J) with B(I,J)
```

For convenience, placeholders are not required where the mapping is exact. Multiple arrays may also be aligned with the same statement. For instance, the alignments in the previous example could also have been specified with the following syntax.

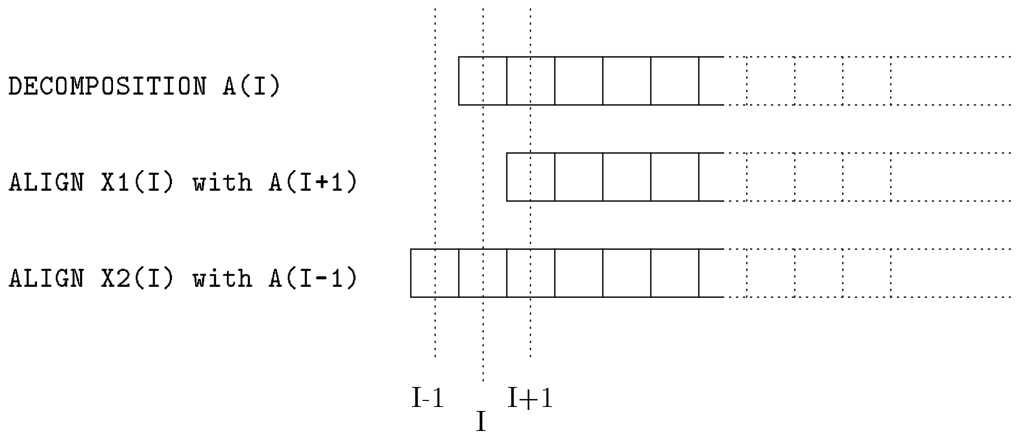


Figure 1 1-D Alignment Offsets

```
REAL X1(N), X2(N,N), X3(N,N)
DECOMPOSITION A(N), B(N,N)
ALIGN X1 with A
ALIGN X2, X3 with B
```

3.2.1 Intra-dimension Alignment

Intra-dimension alignment determines the data decomposition within each dimension. This section describes how offset and stride may be specified.

Alignment Offsets

In FORTRAN D, the user can specify an alignment offset for any dimension of an array. The simplest case occurs when the array and decomposition have the same number of dimensions. Constants are added to the placeholders in the decomposition to indicate the offset in that dimension.

```
REAL X1(N), X2(N)
DECOMPOSITION A(N)
ALIGN X1(I) with A(I+1)
ALIGN X2(I) with A(I-1)
```

In this example, X1 and X2 are aligned with respect to decomposition A by 1 and -1. X1(I) is thus always mapped to the same element of the decomposition as X2(I+2); *e.g.,* X1(1) is mapped together with X2(3).

```
REAL X3(N,N), X4(N,N)
DECOMPOSITION B(N,N)
ALIGN X3(I,J) with B(I,J-1)
ALIGN X4(I,J) with B(I-1,J+2)
```

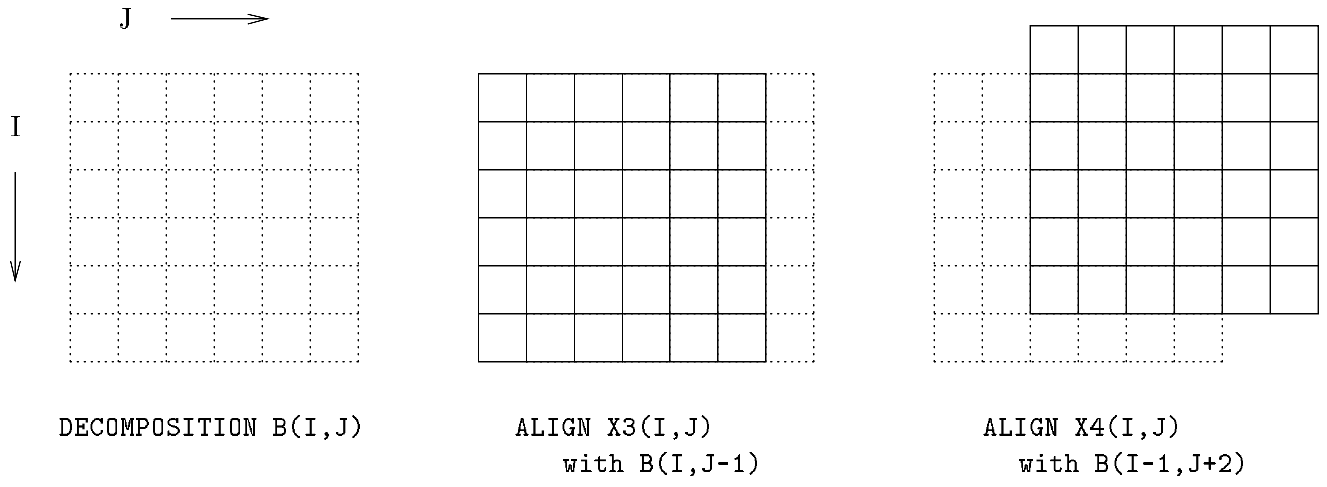


Figure 2 2-D Alignment Offsets

Similarly, in this example the alignment of X3 and X4 with respect to decomposition B means that X3(I,J) is mapped to the same element of B as X4(I+1, J-3).

Alignment Strides

FORTRAN D also allows a stride to be specified when performing intra-dimensional alignment. Alignment strides are used to determine the density of an array mapped to a dimension. They are introduced as coefficients of placeholders in the subscript expressions of decompositions in an `ALIGN` statement. Strides may be also used in combination with offsets.

```
REAL X1(N), X2(N)
DECOMPOSITION A(N)
ALIGN X1(I) with A(2*I)
ALIGN X2(I) with A(2*I-1)
```

In this example, array X1 has a stride of 2 with respect to decomposition A. It is thus mapped to the even elements of A. Array X2 also has a stride of 2, but the alignment offset of -1 causes it to be mapped to every odd element of A. Alignment strides are easily extended to higher order arrays and decompositions, as in the following example.

```
REAL X1(N,N), X2(N,N+N)
DECOMPOSITION B(N,N)
ALIGN X1(I,J) with B(2*I,2*J)
ALIGN X2(I,J) with B(2*I-1,J)
```

Alignment strides with negative values are also allowed; they correspond mapping the reflection of the array dimension onto the decomposition.

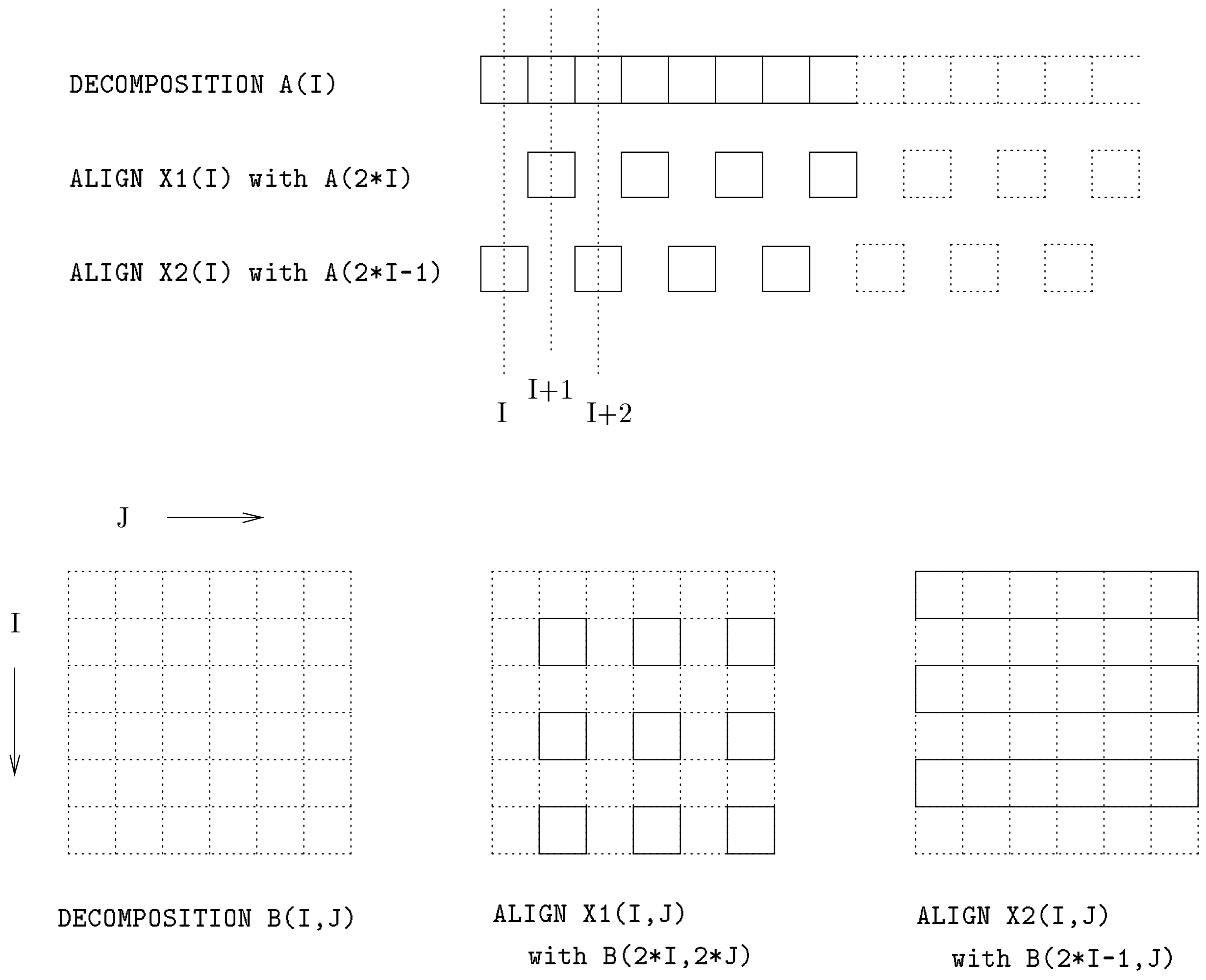
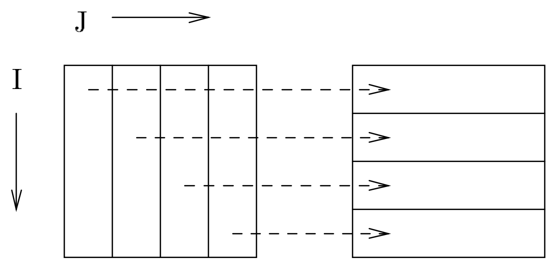

DECOMPOSITION A(I)
ALIGN X1(I) with A(2*I)
ALIGN X2(I) with A(2*I-1)
I
I+1
I+2
J
I
DECOMPOSITION B(I,J)
ALIGN X1(I,J) with B(2*I,2*J)
ALIGN X2(I,J) with B(2*I-1,J)

Figure 3 Alignment Stride



`ALIGN X(I,J) with B(J,I)`

Figure 4 Alignment Permutation

3.2.2 Inter-dimension Alignment

Inter-dimension alignment determines the data decomposition between dimensions. This section describes how permutation, collapse, and embedding may be specified.

Permutation

In FORTRAN D, the user can arbitrarily permute the dimensional alignment between arrays and decompositions. A common application would be to perform array transpositions. Canonical placeholders must be used to mark the aligned dimensions.

```
REAL X1(N,N), X2(N,N,N)
DECOMPOSITION B(N,N), C(N,N,N)
ALIGN X1(I,J), with B(J,I)
ALIGN X2(I,J,K), with C(K,I,J)
```

In this example, the transpose of X1 is mapped to the decomposition B, as indicated by the reversed placeholders I and J. Similarly, the third and first dimensions of X2 are mapped to the first and second dimensions of decomposition B.

Collapse

It is sometimes convenient to ignore certain dimensions of the array when mapping an array to a decomposition. All data elements in the unassigned dimensions are collapsed and mapped to the same location in the decomposition. An array dimension may be collapsed in the `ALIGN` statement simply by excluding its placeholder from the decomposition subscripts, as this demonstrates that the dimension has no effect on the actual alignment.

```
REAL X1(N,N), X2(N,N), X3(N,N,N), X4(N,N,4)
ALIGN X1(I,J) with A(I)
ALIGN X2(I,J), X3(I,J,K) with A(J)
ALIGN X4(I,J,K) with B(I,J)
```

In this example, the first dimension of array X1 is mapped onto the decomposition A. The second dimension of X1 is collapsed and stored on the same processor. In other words, each row of X1

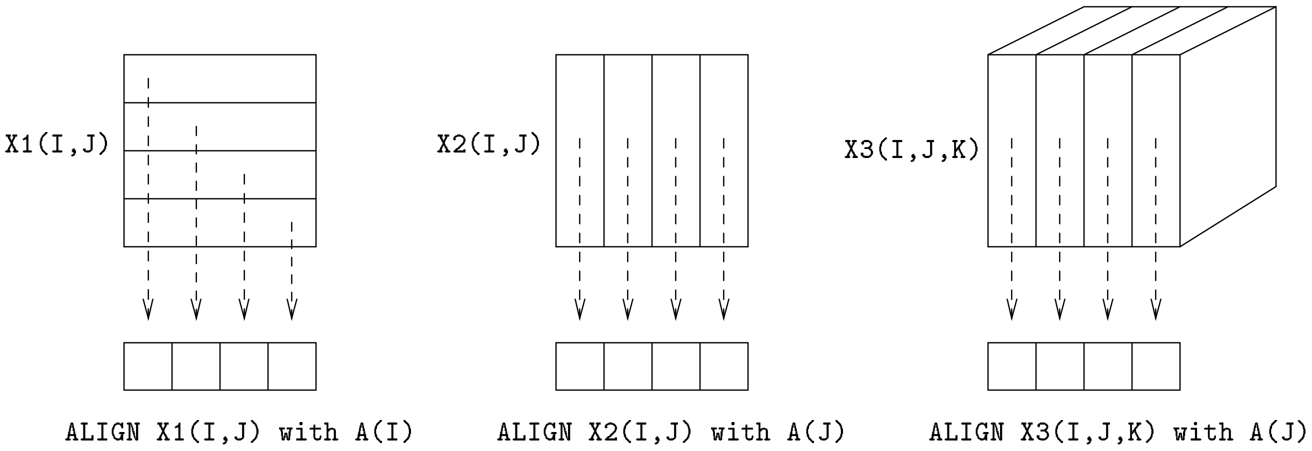


Figure 5 Array Collapse

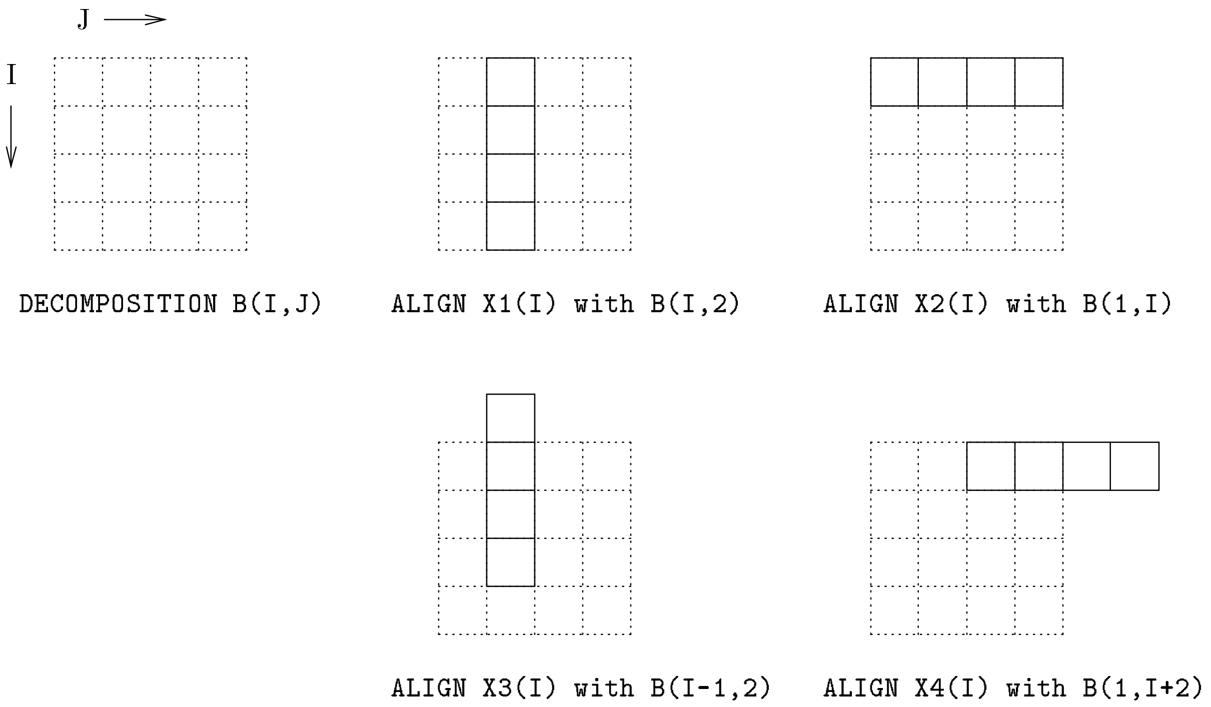


Figure 6 Array Embedding

is mapped to an individual element in decomposition A. Similarly, each column of array X2 is mapped to A. For array X3, the second dimension is mapped onto the decomposition A, with the first and third dimensions local. Array collapse frequently occurs when an array dimension is used to store multiple data fields per problem element, such as for array X4 in the example.

Embedding

Conversely, it may be necessary to map arrays with fewer dimensions onto the decomposition. In these cases it is necessary to specify both the mapping for each dimension of the array and the actual position of the array in the unmapped dimensions of the decomposition. This determines the embedding of the array in the decomposition.

```
REAL X1(N), X2(N), X3(N), X4(N)
ALIGN X1(I) with B(I,2)
ALIGN X2(I) with B(1,I)
ALIGN X3(I) with B(I-1,2)
ALIGN X4(I) with B(1,I+2)
```

In this example, array X1 is mapped to the first dimension of decomposition B, a column. It is necessary to specify the actual column position with a constant remaining unmapped dimension. In this case the constant "2" in the second dimension indicates that X1 should be mapped to the second column of decomposition B. Similarly, array X2 is mapped to the first row of decomposition B. In a more complex example, arrays X3 and X4 are both aligned and mapped to decomposition B.

This scheme can be extended to higher order arrays and decompositions. Note that when arrays are mapped only to part of a decomposition, the array may not be mapped to all the processors in the machine, depending on the actual distribution.

Combinations

The user can apply any combination of intra-dimensional and inter-dimensional alignments when mapping arrays to decompositions.

```
REAL X1(N,N), X2(N,N)
ALIGN X1(I,J) with B(J+2,I-1)
ALIGN X2(I,*) with B(4,I-2)
```

In this example, array X1 is both aligned and transposed with respect to decomposition B. Array X2 is collapsed into its first dimension (forming a single column), mapped to the fourth row of decomposition B, and aligned by -2.

3.2.3 Alignment Options

The `ALIGN` statement also supports options to specify actions for overflows, mapping parts of arrays to a decomposition, and either totally or partially replicating arrays. These options are discussed in this section.

Array Overflow

It is possible that the array to be aligned does not fit completely within the decomposition, causing an *overflow*. In these cases, an *optional* `overflow` clause may be used to select one of three options, `ERROR`, `TRUNC`, and `WRAP`, described below.

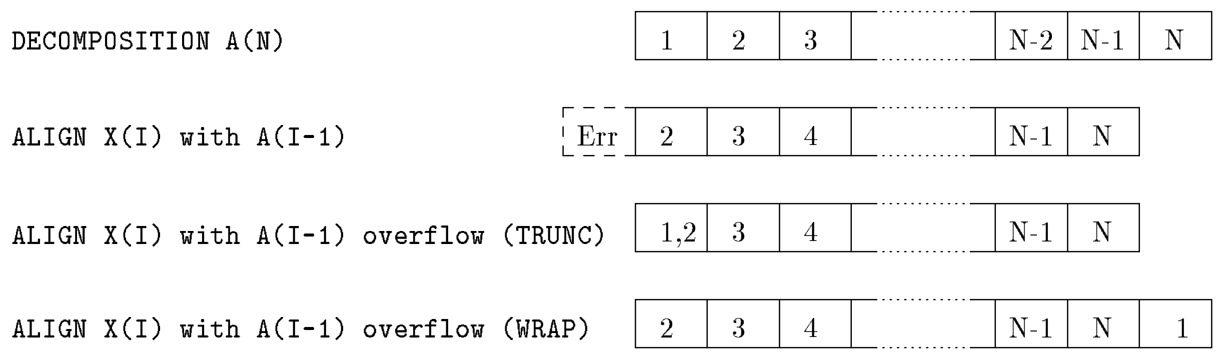

DECOMPOSITION A(N)
1
2
3
N-2
N-1
N
ALIGN X(I) with A(I-1)
Err
2
3
4
N-1
N
ALIGN X(I) with A(I-1) overflow (TRUNC)
1,2
3
4
N-1
N
ALIGN X(I) with A(I-1) overflow (WRAP)
2
3
4
N-1
N
1

Figure 7 Array Overflow

The default choice, `ERROR`, considers elements overflowing the decomposition to be unmapped. Any attempt to access such elements will be considered to be an error. Alternatively, the user may choose to truncate the array with the `TRUNC` option. All elements overflowing the decomposition are then mapped to the element on the edge of the decomposition in that dimension. `WRAP`, the last choice, wraps overflowing array elements back to the opposite end of the decomposition. Systolic algorithms in particular may benefit from this feature.

```
REAL X1(N), X2(N), X3(N), X4(N,N,N)
DECOMPOSITION A(N), C(N,N,N)
ALIGN X1(I) with A(I-1)
ALIGN X2(I) with A(I-1) overflow (TRUNC)
ALIGN X3(I) with A(I-1) overflow (WRAP)
ALIGN X4(I,J,K) with C(I-1,J-1,K-1) overflow (ERROR,TRUNC,WRAP)
```

In the previous example, attempting to reference X1(1) would be illegal since it maps to the undeclared decomposition element A(0), which by default is defined as type `ERROR`. Because X2 is truncated, the array elements X2(1) and X2(2) map to the same decomposition element A(1). Wrapping X3 causes the array element X3(1) to map to the decomposition element A(N). The alignment statement for X4 shows how overflow options may be specified for multidimensional decompositions.

Array Range

By default, the `ALIGN` statement maps the entire array to the decomposition. However, FORTRAN D also allows just part of an array to be mapped onto a decomposition. This may be done by specifying a section of the array to be mapped using the `range` clause. The `range` clause specifies a range for each dimension of the form <from>:<to>. The `*` symbol indicates that the entire array dimension should be mapped. A subarray can thus be selected and aligned with a decomposition. This partial alignment feature is useful when one large work array is subdivided into several smaller logical arrays at run-time.

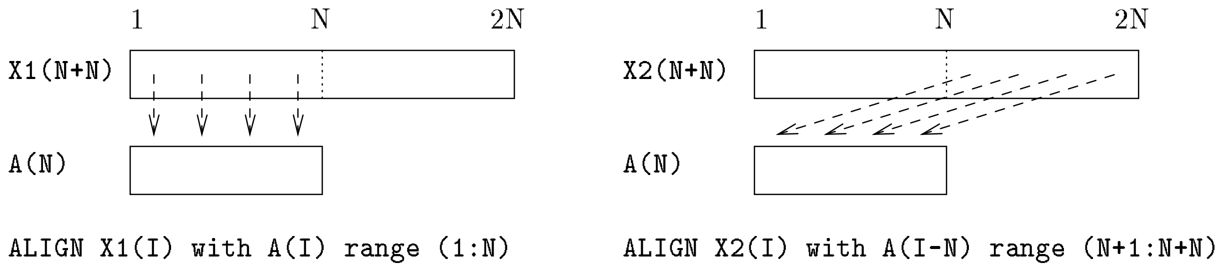

1
N
2N
X1(N+N)
A(N)
ALIGN X1(I) with A(I) range (1:N)
1
N
2N
X2(N+N)
A(N)
ALIGN X2(I) with A(I-N) range (N+1:N+N)

Figure 8 Array Range

```
REAL X1(N+N), X2(N+N), X3(N+N,N+N), X4(N,N+N)
DECOMPOSITION A(N), B(N,N)
ALIGN X1(I) with A(I) range (1:N)
ALIGN X2(I) with A(I-N) range (N+1:N+N)
ALIGN X3(I,J) with B(I-N,J-N) range (N+1:N+N,N+1:N+N)
ALIGN X4(I,J) with B(I,J-N) range (*,N+1:N+N)
```

In the previous example, the `range` clause is used to map elements 1 to N of array X1 to decomposition A and elements N+1 to 2N of array X2 to decomposition A, starting at decomposition element 1. Similarly, the subarray of X3 beginning at (N+1,N+1) is aligned with decomposition B. Finally, half of array X4 is aligned with decomposition B, with the * symbol indicating that the entire first dimension of X4 is mapped to the decomposition.

Replication

The `ALIGN` statement may also be used a means to replicate distributed variables in Fortran D. This can be done by assigning a range for a dimension rather than a position or placeholder. Ranges may be specified as <from>:<to>, or simply as * if the entire dimension is desired. If an assignment is made to a replicated value, all replicated values would be updated. Note that all variables not aligned to a decomposition are considered to be totally replicated on all processors. The compiler will label scalar and array variables as local, distributed, or replicated.

```
REAL X1(N), X2(N), X3(N)
DECOMPOSITION B(N,N)
ALIGN X1(I) with B(I,1:2)
ALIGN X2(I) with B(I,*)
ALIGN X3(I) with B(I-1:I,*)
```

In the first `ALIGN` statement in this example, a range from 1 to 2 is specified in the second dimension of B. This causes each of the first two columns of decomposition B to each get a copy of array X1, in effect replicating every element of X1 among the first five elements of each row of B. In the second `ALIGN` statement, the * symbol in the second dimension of decomposition B specifies that each element of array X2 is replicated across all elements of B in the same row. The two modes may also be combined, as in the third statement, where each row of B gets a copy of the element of array X3 in that row, as well the element of X3 from the previous row.

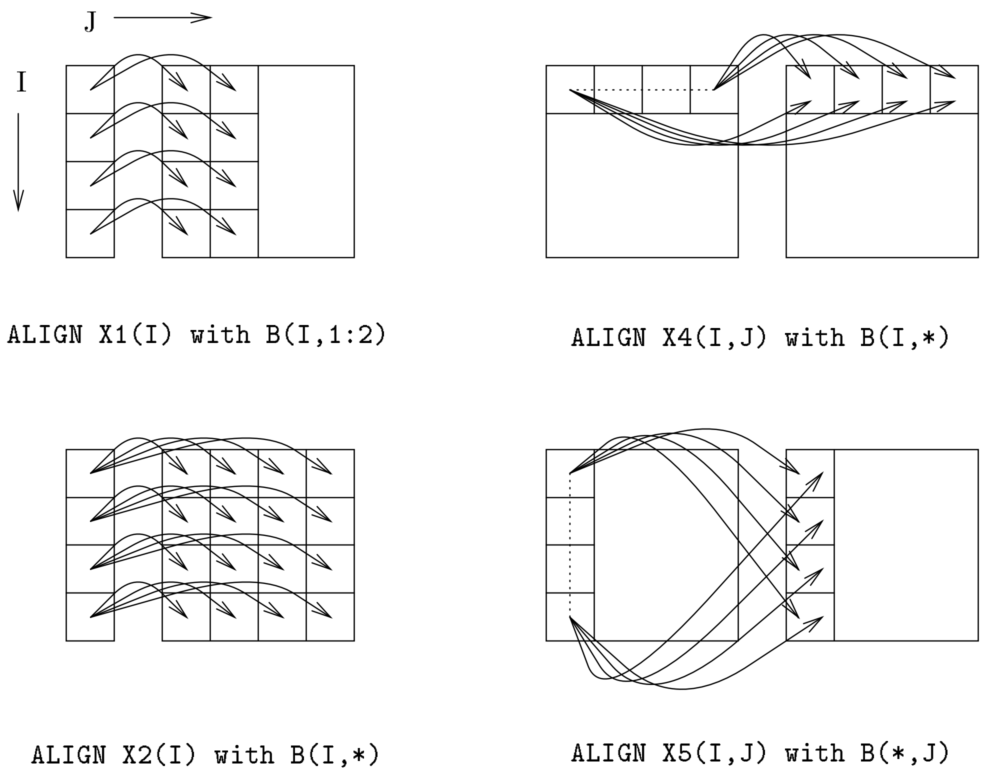

J
I
ALIGN X1(I) with B(I,1:2)
ALIGN X4(I,J) with B(I,*)
ALIGN X2(I) with B(I,*)
ALIGN X5(I,J) with B(*,J)

Figure 9 Array Replication

```
        REAL X4(N,N), X5(N,N), X6(N,N)
        DECOMPOSITION B(N,N)
        ALIGN X4(I,J) with B(I,*)
        ALIGN X5(I,J) with B(*,J)
        ALIGN X6(I,J) with B(*,*)
```

Replication can also be extended to higher dimension arrays. In this example, the first `ALIGN` statement causes each row of array X4 to be mapped to each element in the corresponding row of decomposition B. Similarly, the second `ALIGN` statement causes each column of X5 to be mapped to each element in the corresponding column of B. Finally, each element of X6 is totally replicated for each element of decomposition B; *i.e.*, each processor is guaranteed to have a copy of X6. This is exactly the default case for unaligned arrays.

4 Machine Mapping

In FORTRAN D, we use the DISTRIBUTE statement to specify the mapping of the decomposition to the physical parallel machine. The distribution selected will affect the ability of the compiler to minimize communications and load imbalance for the resulting program. Physical machine characteristics such as the number of processors, amount of memory per processor, and communication costs between processors must all be taken into account since they affect which distributions are feasible and efficient. Program characteristics such as the size of distributed arrays and computation structure may also be crucial in determining a good distribution.

In addition, data parallelism may either be regular or irregular. Regular parallelism can be effectively exploited through relatively simple data distributions. Irregular data parallelism, on the other hand, may require irregular data distributions and run-time preprocessing to manage the parallelism.

In FORTRAN D, a distribution specifies the machine mapping for exactly one decomposition. The compiler then applies the distribution to all the arrays mapped to the decomposition. The user does not need to specify a distribution for each array. It is illegal to access any element of a distributed array before it has been mapped to the machine with a DISTRIBUTE statement.

4.1 n$proc

FORTRAN D reserves the variable n$proc to indicate the number of processors available. It may be evaluated at run-time or passed as a compile-time option to the compiler.

4.2 DISTRIBUTE Statement

The DISTRIBUTE statement takes the name of a decomposition and assigns an *attribute* to each dimension of the decomposition. Each attribute describes the mapping of the data in that dimension of the decomposition. Attributes in each dimension are independent, and may specify regular or irregular distributions, as described in later sections. The symbol * is used to denote dimensions which are assigned locally; *i.e.,* these dimensions are not distributed.

```
DISTRIBUTE A(attribute)
DISTRIBUTE B(attribute, attribute)
```

In this example, the decompositions A and B are assigned an attribute for each dimension of the decomposition. Distributions in effect describe the assignment of data to an underlying processor array.

4.3 Regular Distributions

The three types of attributes for regular distributions in FORTRAN D are BLOCK, CYCLIC, and BLOCK_CYCLIC. Suppose there are P processors and N elements in a decomposition. We assume for simplicity that P divides N evenly. If this is not the case, the resulting distribution will be slightly unbalanced. The FORTRAN D distributions can then be described as follows:

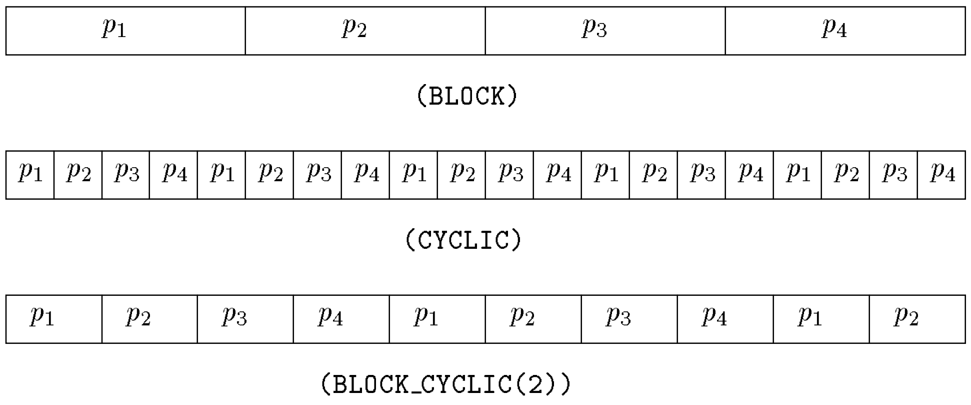

p_1 p_2 p_3 p_4
(BLOCK)
p_1 p_2 p_3 p_4 p_1 p_2 p_3 p_4 p_1 p_2 p_3 p_4 p_1 p_2 p_3 p_4 p_1 p_2 p_3 p_4
(CYCLIC)
p_1 p_2 p_3 p_4 p_1 p_2 p_3 p_4 p_1 p_2
(BLOCK_CYCLIC(2))

Figure 10 1-D Distributions

- `BLOCK` divides the decomposition into contiguous chunks of size N/P, assigning one block to each processor.
- `CYCLIC` specifies a round-robin division of the decomposition, assigning every P^{th} element to the same processor. `CYCLIC` distributions are useful for load balancing.
- `BLOCK_CYCLIC` is similar to `CYCLIC` but takes a parameter M. It first divides the dimension into contiguous chunks of size M, then assigns these chunks in the same fashion as `CYCLIC`.

Only one attribute may be assigned for each dimension of the decomposition. However, multi-dimensional decompositions may have different combinations of distribution patterns. For these decompositions, processors are allocated as evenly as possible to each distributed dimension. This creates an implicit processor array. The following examples show some common Fortran D distributions.

```
DISTRIBUTE A(BLOCK), B(CYCLIC), C(BLOCK_CYCLIC(2))
DISTRIBUTE A(BLOCK,*), B(*,BLOCK), C(BLOCK,BLOCK)
DISTRIBUTE A(CYCLIC,*), B(*,CYCLIC), C(CYCLIC,CYCLIC)
DISTRIBUTE A(BLOCK_CYCLIC(2),*), B(*,BLOCK_CYCLIC(3))
DISTRIBUTE C(BLOCK_CYCLIC(2),BLOCK_CYCLIC(4))
DISTRIBUTE A(CYCLIC,BLOCK), B(BLOCK,CYCLIC), C(BLOCK,BLOCK_CYCLIC(2))
```

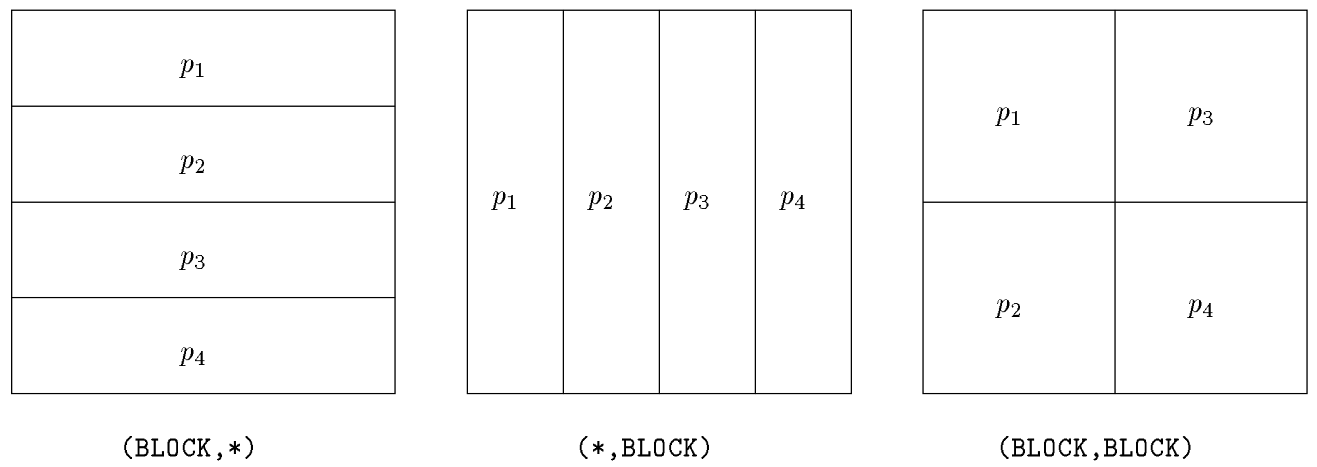


Figure 11 2-D Block Distributions

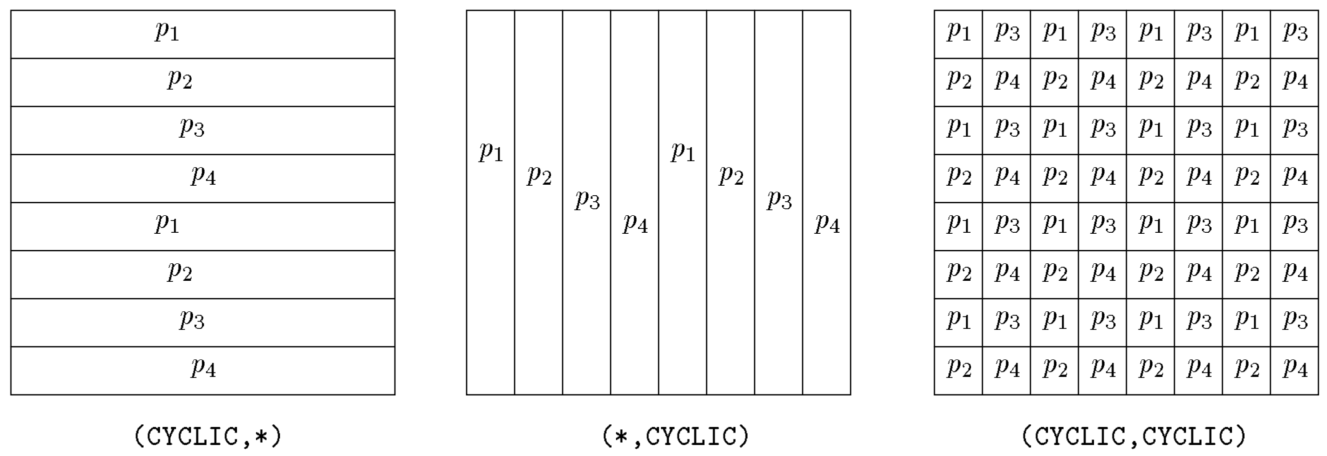
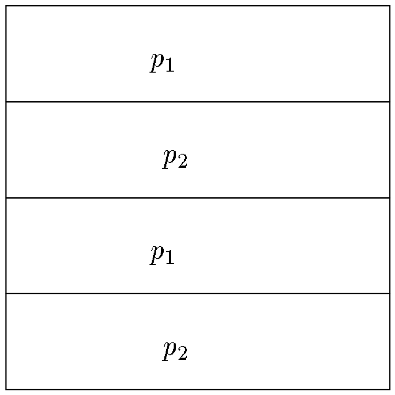
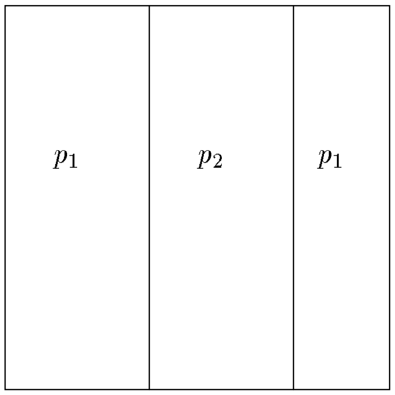


Figure 12 2-D Cyclic Distributions





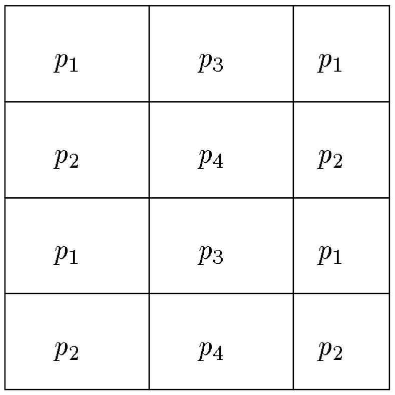
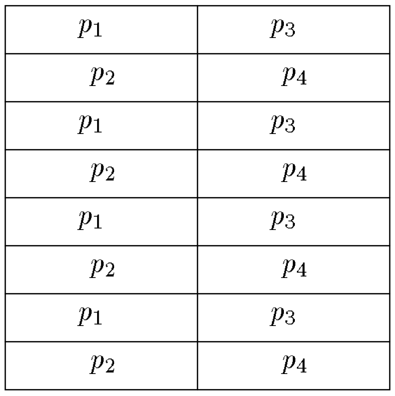
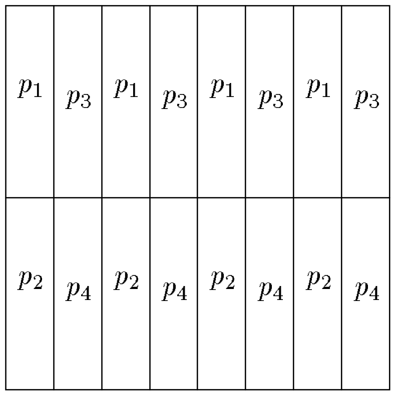


Figure 13 2-D BLOCK_CYCLIC Distributions




p_1 p_3 p_1 p_3 p_1 p_3 p_1 p_3
p_2 p_4 p_2 p_4 p_2 p_4 p_2 p_4
(BLOCK,CYCLIC)

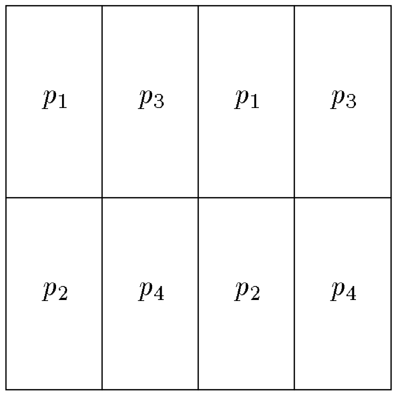


Figure 14 2-D Combination Distributions

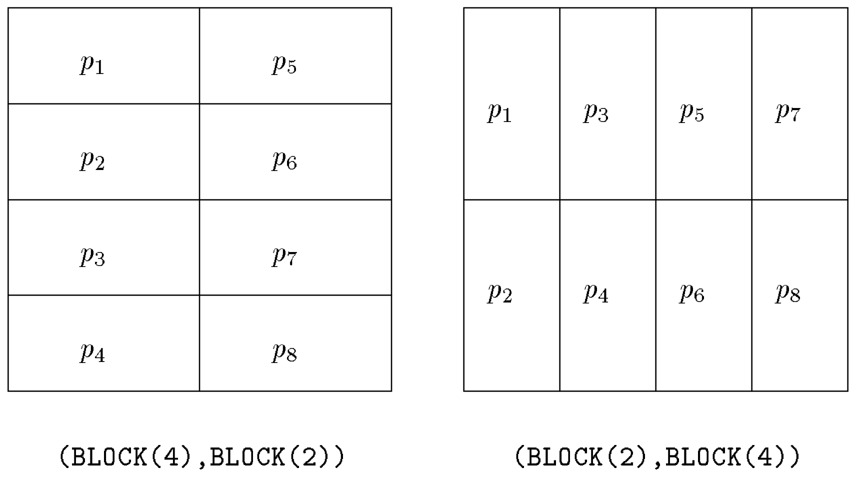


Figure 15 2-D Uneven Block Distributions

4.4 Processor Allocation

Fortran D also provides the capability of specifying processor allocations, where the allocation specifies the number of processors assigned to each dimension of the decomposition. Users can thus define their own uneven processor allocation, instead of using the even processor allocations the compiler generates by default.

Processor allocations are specified by adding an additional parameter indicating the number of processors for each dimension of the distribution. The multiplicand of the processors in each dimension must be less than or equal to the total number of processors defined by `n$proc`, since Fortran D does not support virtual processors. If `BLOCK_CYCLIC` is passed two parameters, the first parameter specifies the block size and the second specifies the number of processors. The following are some examples of uneven distributions:

```
DISTRIBUTE A(BLOCK(4),BLOCK(2)), B(BLOCK(2),BLOCK(4))
DISTRIBUTE A(BLOCK(4),CYCLIC(2)), B(BLOCK(2),CYCLIC(4))
DISTRIBUTE C(CYCLIC(4),BLOCK(2)), D(BLOCK_CYCLIC(2,2),BLOCK(4))
```

4.5 Unsupported Distributions

Though Fortran D supports several regular distribution patterns, our intention is to keep the distribution attributes relatively simple to allow straightforward communications generation by the compiler. As a result, Fortran D distributions obey these simple rules:

- decomposition dimensions are distributed independently
  (no diagonal distributions are possible)
- decomposition segments have uniform size and shape (except for boundary conditions)
- processor assignments are regular

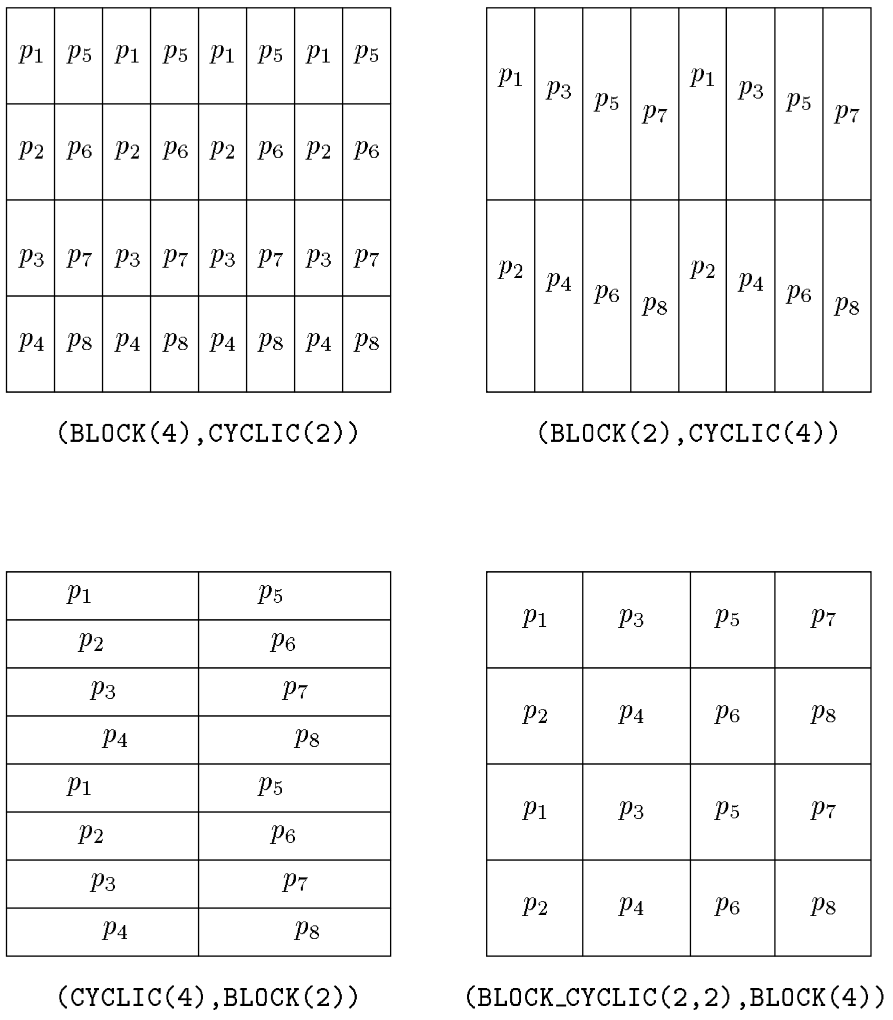


Figure 16 2-D Uneven Combination Distributions

Figure 17 shows some distributions we do not plan to support in Fortran D. We do not believe there will be a significant loss of performance caused by using the regular distributions provided in Fortran D.

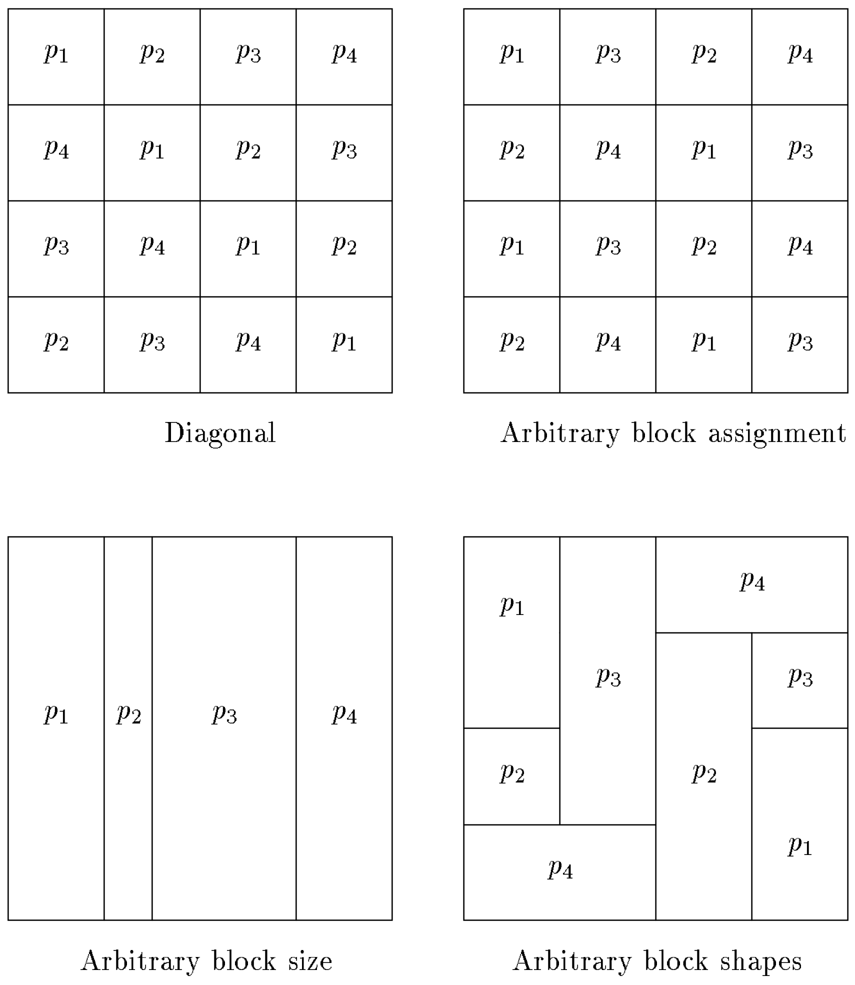


Figure 17 Unsupported Distributions

4.6 Irregular Distributions

For problems with irregular data parallelism, regular distributions may not be efficient. For these cases, FORTRAN D allows user specified irregular distributions through the use of a mapping array, which will itself usually be distributed. An example for implementing an irregular distribution in this manner is as follows:

```
n$proc = 4
REAL X(16)
INTEGER MAP(16)
DECOMPOSITION REG(16), IRREG(16)
ALIGN MAP with REG
ALIGN X with IRREG
DISTRIBUTE REG(BLOCK)
...set values of MAP array by some algorithm...
DISTRIBUTE IRREG(MAP)
```

In this example, the elements of MAP must be set to integers between 1 and 4 (the number of processors). IRREG(i) will then be stored on the processor value in MAP(i), as shown in Figure 18.

If an element of MAP is not a valid processor number, then that element of decomposition IRREG will not be mapped to any processor; accessing such an element is an error. This is the case with X(15) in the figure. Changes to MAP made after the `DISTRIBUTE` statement is executed do not affect the distribution. MAP may be either distributed or replicated; distributed MAP arrays will consume less memory, but may require more communication steps to access elements.

Combined Regular and Irregular Distribution

A mixture of regular and irregular distributions may also be used.

```
n$proc = 16
INTEGER MAP(16)
REAL X(16,16)
DECOMPOSITION A(16), B(16,16)
ALIGN MAP with A
ALIGN X with B
DISTRIBUTE A(BLOCK)
...set values of MAP array by some algorithm...
DISTRIBUTE B(MAP,BLOCK(4))
```

In this example, the map array is block distributed among all processors. The array X (aligned with decomposition B) is distributed irregularly in the first dimension according to the map array, and block distributed in the second dimension. Since only four processors are available in the first dimension, the map array must only provide a distribution for processors 1–4.

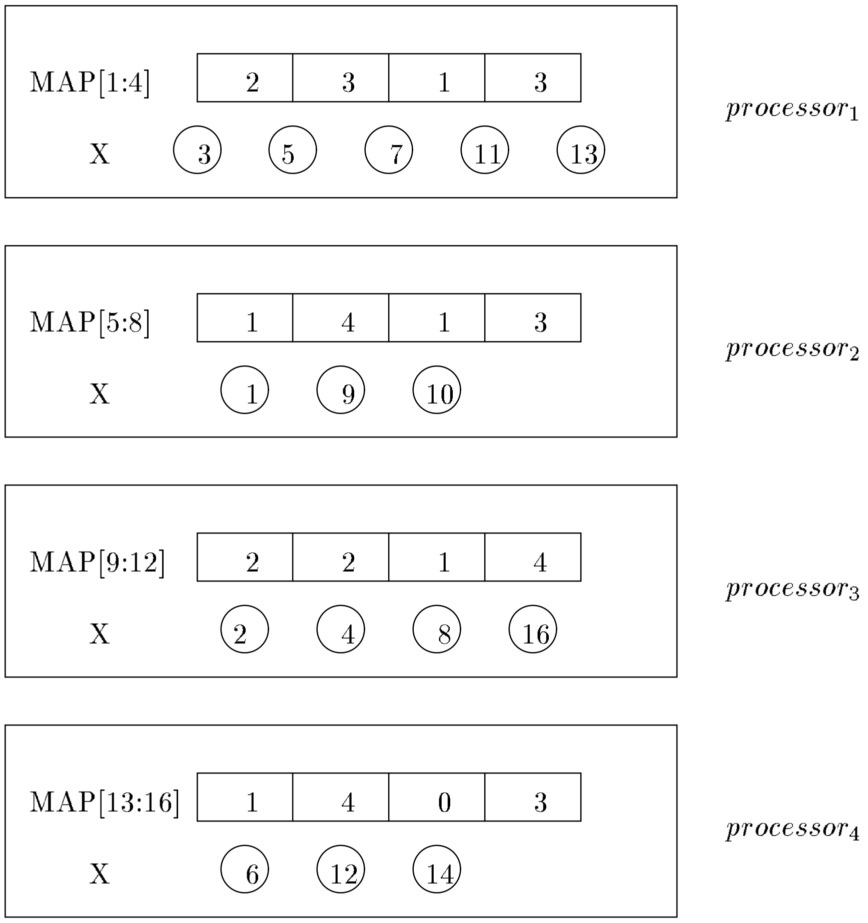


Figure 18 Irregular Distribution Example

5 Additional Features

In the previous sections we presented data decomposition specifications. Here we discuss some other features of FORTRAN D.

5.1 Dynamic Alignment and Distribution

Data mappings may also change between different phases, thereby requiring dynamic realignment and/or redistribution to reduce data movement. FORTRAN D thus supports dynamic data decomposition. Depending on their location, both `ALIGN` and `DISTRIBUTE` may be interpreted as executable statments rather than declarations. In the following example, the second set of data specifications cause dynamic realignment of arrays X and Y. This is done to reduce communications for the second loop.

```
REAL X(N), Y(N)
DECOMPOSITION A(N)
ALIGN X, Y with A
DISTRIBUTE A(BLOCK)
DO I = 1,N
     X(I)=Y(I)
ENDDO
...
ALIGN X(I) with A(I+1)
DO I = 1,N
     X(I)=Y(I+1)
ENDDO
```

Another reason to employ dynamic data distributions is to is to configure a program for greater efficiency, based on the problem size or number of available processors.

```
REAL X(N,N)
DECOMPOSITION B(N,N)
ALIGN X with B
IF (n$proc .GT. 20) THEN
     DISTRIBUTE B(BLOCK,BLOCK)
ELSE
     DISTRIBUTE B(BLOCK,*)
ENDDO
```

In this example, the program is configured so that the data distribution chosen is dependent on the total number of processors available. The FORTRAN D compiler will require additional sophistication in order to handle dynamic data decompositions.

5.2 Procedures

There are a number of issues concerning procedures in FORTRAN D. First, it is permitted to call procedures with distributed arrays as arguments. The compiler will perform all the analysis required to generate the correct code.

Second, in FORTRAN D the effect of all `DECOMPOSITION`, `ALIGN`, and `DISTRIBUTE` statements are limited to the scope of the enclosing procedure. This provides users with a structured method to limit the scope of their decompositions, and simplifies the problem of dealing with dynamic decompositions.

```
REAL X(N), Y(N)
DECOMPOSITION A(N)
ALIGN X, Y with A
DISTRIBUTE A(BLOCK)
CALL FOO(X)
X(1) = ...
```

For instance, in this example the scoping rule in FORTRAN D ensures that the array X will be `BLOCK` distributed at the assignment to X(1), even if procedure FOO redistributes X locally. However, procedures do inherit data decompositions from their callers. Upon entry of procedure FOO, the array X will be `BLOCK` distributed. Array X may be dynamically redistributed in procedure FOO, but FORTRAN D ensures that it will not affect the decomposition of X of the parent procedure.

5.3 FORALL Loops

Certain programming constructs, such as the use of index arrays, make compile-time detection of data dependences impossible. This is especially true for irregular computations, since many parallel loops cannot be detected by the compiler. The compiler is forced to assume *loop-carried* or inter-iteration dependences that force synchronization to be inserted [AK87].

As a remedy, FORTRAN D defines `FORALL` to be a loop such that each iteration can *only use values defined before the loop or within the current iteration.* When a statement in an iteration of the `FORALL` loop accesses a memory location, it will not get any value written by a different iteration of the loop. Instead, it will get the *old* value at that memory location (*i.e.,* the value at that location before the execution of the `FORALL` loop) or it will get some new value written on the current iteration. Another way of viewing the `FORALL` loop is that each iteration gets its own copy of the entire data space that exists before the execution of the loop.

At the end of a `FORALL` loop, any variables that are assigned *new* values by different iterations have these values *merged* at the end of the loop. Merges are performed deterministically, by using the value assigned from the latest sequential iteration. *Note: for performance reasons, some compilers may not wish to support this merge feature. See Appendix A.*

The major benefit of a `FORALL` loop is that since no values depend on other iterations, the loop may be executed in parallel without communication. However, communication may still be required before the loop to acquire non-local values, and after the loop to update or merge non-local values. Another advantage of the `FORALL` loop is that it has deterministic semantics, provided that the underlying system merges values in a deterministic manner. The syntax of the `FORALL` loop is shown in the following example:

```
FORALL I = 1,N
     X(IDX(I)) = ...
     ... = X(IDX(I+1))
ENDDO
```

In this example, the `FORALL` loop may be executed in parallel without communication or synchronization, even though loop-carried dependences cannot be eliminated by compile-time analysis. Instead, the compiler and run-time system will ensure that statements in the loop body access old values of X instead of new values written on other iterations.

5.3.1 Example FORALL Loop

Here we provide a more detailed example. In the following `FORALL` loop, there are three dependences caused by the assignment to X(I) at statement S_1—a loop-carried antidependence to X(I+1) at S_2, a loop-independent true dependence to X(I) at S_3, and a loop-carried true dependence to X(I-1) at S_4.

```
     FORALL I = 1,N
S1       X(I) = ...
S2       ... = X(I+1)
S3       ... = X(I)
S4       ... = X(I-1)
```

```
      ENDDO
```

Both sequential FORTRAN and FORALL semantics specify that the reference to X(I+1) at S_2 uses its old value; *i.e.,* the value of X(I+1) before it is assigned at statement S_1. Similarly, both sequential and FORALL semantics require the reference to X(I) at S_3 to use the new value; *i.e.,* the value assigned to X(I) at S_1. This is because the new value is assigned in the current iteration.

On the other hand, sequential and FORALL semantics differ for the loop-carried true dependence between X(I) and X(I-1). Sequential FORTRAN semantics require that the reference to X(I-1) at statement S_4 use the new value computed at S_1. However, FORALL semantics cause statements in the loop body to use new values *only if they are calculated in the current loop iteration.* All other values are old values from before the FORALL loop. The reference to X(I-1) thus uses the old value of X(I-1), before it is assigned to at statement S_1. In effect, all loop-carried true dependences are converted to antidependences in a FORALL loop.

5.3.2 Nested FORALL Loops

Multiple nested FORALL loops may be used to specify more than one level of data parallelism. A nested FORALL loop has exactly the same semantics as the standard FORALL loop—no value may be computed and used on different iterations of the FORALL loop. In most cases, all communications can be moved entirely out of several nested FORALL loops.

```
      FORALL I = 1,N
         FORALL J = 1,N
            X(...) = ...
         ENDDO
      ENDDO

      FORALL I = 1,N
         FORALL J = 1,N
            X(...) = ...
            ... = X(...)
         ENDDO
      ENDDO
```

For instance, consider the two example loop nests. Standard FORALL semantics allow all communications resulting from values assigned to X in the inner J loop to be moved outside the J loop. What is less clear is that the communications can actually be moved outside the outer I loop as well. This is because the semantics of the FORALL loop guarantee that the values of X produced (by the J loop) in one iteration of I cannot be used until the entire I loop is complete. Since there is no possible use of these values in the same iteration of loop I, the communications may be delayed to the end of the entire loop nest. However, a different situation exists in this example:

```
      FORALL I = 1,N
         FORALL J = 1,N
            X(...) = ...
         ENDDO
         ... = X(...)
      ENDDO
```

In this loop nest, there is actually a possible of use of X following the inner J loop. The difference here is that the values generated in the inner J loop may be used in the same iteration of the outer I loop. If the compiler cannot eliminate possible dependence between the definition and use of X, communications may be necessary at the end of the J loop to update values of X.

Our intent in providing the `FORALL` loop in FORTRAN D is to provide an *optional* method for users to aid the compiler in generating efficient codes for irregular or sparse computations. `FORALL` loops are unnecessary for regular computations—we believe that a sophisticated compiler can readily extract the parallelism from normal `do` loops for regular computations.

We have defined semantics of the `FORALL` loop to be quite close to sequential FORTRAN. In particular, `FORALL` loops are deterministic. As a result we believe that it will be easy to understand and use for scientific programmers. The `FORALL` loop possesses similar semantics to the CM FORTRAN `FORALL` statement [TMC89, ALS90] and the Myrias `PARDO` loop. In fact, the `FORALL` statement in CM FORTRAN may be used as a special form of the `FORALL` loop—one that has only one statement in the loop body.

5.4 Reductions

A reduction is an operation on a collection of data that results in new data of lesser dimensionality, usually a single scalar value. Simple but common examples of reductions include calculating the sum or maximum of a vector or array of numbers. FORTRAN D provides the `REDUCE` statement as an *optional* method of specifying reductions that the compiler may find difficult to detect. The syntax of the `REDUCE` statement is as follows:

`REDUCE(`*function*`,` LHS`,` RHS`)`

Where the *function* is the reduction function to be performed, the *lhs* is the target data, and the RHS is the source data. The following standard reduction functions are provided in FORTRAN D:

SUM	sum of a list of numbers
PROD	product of a list of numbers
MIN	minimum of a list of numbers
MAX	maximum of a list of numbers
AND	logical AND of a list of booleans
OR	logical OR of a list of booleans

Programmers may also define their own reduction functions, providing much greater flexibility in performing reductions. FORTRAN D will assume any function passed to a `REDUCE` statement to be user-defined if it does not match the name of a standard reduction function. In such cases, all other arguments to the `REDUCE` statement are passed as arguments to the user-defined reduction function. The following example shows reductions performed with both standard and user-defined reduction functions:

```
REAL X(N), S, P, M1, M2, Z
BOOLEAN B(N), T1, T2
DO I = 1,N
     REDUCE(SUM, S, X(I))
     REDUCE(PROD, P, X(I))
     REDUCE(MIN, M1, X(I))
     REDUCE(MAX, M2, X(I))
```

```
          REDUCE(AND, T1, B(I))
          REDUCE(OR, T2, B(I))
          REDUCE(USER_FUNCTION, Z, X(I))
     ENDDO
```

Reductions in DO loops may be automatically recognized by the FORTRAN D compiler, even if the REDUCE statement is not employed. However, use of the REDUCE statement is required for reductions in FORALL loops, since FORALL semantics change the meaning of standard user-programmed reductions. Reductions in essence provide appropriate merge functions for FORALL loops.

5.4.1 Restrictions

Reductions provide a means for executing otherwise sequential computations in parallel. However, several restrictions must be observed in order to avoid nondeterministic results when using reductions in FORALL loops. First, because reductions change the order of operations in a reduction, all user-defined reduction functions must be both associative and commutative. Otherwise, any change in the actual evaluation order of the reduction may affect the final value returned by the reduction operation. Note that this requirement has to be relaxed for floating point operations, which may prove unstable because of rounding errors.

Second, since intermediate values of the LHS variables in reductions are undefined, they must not be used within the loop. However, they may be employed in other REDUCE statements in the same loop, provided that the reduction functions are identical. The following shows some examples of variables involved in multiple reductions.

```
     REAL X(N), Y(N), S, M
     FORALL I = 1,N
          REDUCE(SUM, S, X(I))
          REDUCE(SUM, S, Y(I))
          IF (...) THEN
             REDUCE(MAX, M, X(I))
          ELSE
             REDUCE(MAX, M, Y(I))
          ENDIF
          ... = S
     ENDFOR
```

In the previous example, the variables S and M serve as the LHS of several reductions. Variable S is used to sum up the values in both arrays X and Y, and variable M is set to the maximum value in some subset of arrays X and Y. In both cases the reductions are legal since the same reduction function is used. On the other hand, the FORTRAN D compiler will mark the last statement in the loop as illegal, because it attempts to use an intermediate value of variable S during execution of the FORALL loop.

5.4.2 Location Reductions

FORTRAN D also provides additional support for determining the location of minimum or maximum values. For the MIN and MAX reductions, the REDUCE statement will accept additional pairs of arguments of the form <LHS,RHS>. In the course of the reduction, the values of the RHS will be assigned to that of the LHS when the minimum or maximum element is found. This provides a mechanism for determining the location of the minimum or maximum value.

```
INTEGER I, J, IDX1, IDX2, IDX3, IDX4, IDX5, IDX6
REAL X1(N), X2(N,N), M1, M2, M3, M4
DO I = 1,N
     REDUCE(MIN, M1, X1(I), IDX1, I)
     REDUCE(MAX, M2, X1(I), IDX2, I)
     DO J = 1,N
         REDUCE(MIN, M3, X2(I,J), IDX3, I, IDX4, J)
         REDUCE(MAX, M4, X2(I,J), IDX5, I, IDX6, J)
     ENDDO
ENDDO
```

In the previous example, additional arguments of the form <var, current index> are passed to the `REDUCE` statements for to find the index of the minimum or maximum element of the array. If there are multiple elements with the minimum or maximum value, the assignment is performed only for the first such value found.

5.5 On Clause

Fortran D provides a feature from Kali [KMV90], an *optional* `on` clause. The `on` clause is used to specify the processor which will execute each iteration of a loop. This allows user greater control of where the computation is performed for load-balancing and reducing communications.

```
n$proc = 4
REAL X(1024), Y(1024), Z(1024)
DECOMPOSITION A(1024)
ALIGN X, Y, Z with A
DISTRIBUTE A(BLOCK)
FORALL I = 1,512 on HOME(A(I))
     X(I+512) = F(X(I),Y(I),Z(I))
ENDFOR
```

In this example, it may be advantageous to perform the computation on the processor where the data is stored (where X(I) is) rather than where the results are to be sent (where X(I+512) is). This is precisely what the `on` clause specifies. There are three forms of the `on` clause.

```
n$proc = 4
REAL X(N)
DECOMPOSITION A(N)
ALIGN X(I) with A(I+1)
DISTRIBUTE A(CYCLIC)
FORALL I = 1,N on HOME(A(I))
FORALL I = 1,N on HOME(X(I))
FORALL I = 1,N on MOD(I, n$proc) + 1
```

In all cases, the expression in the `on` clause names the processor to execute a given iteration of the `FORALL` loop. `HOME` is used to derive the identifier of the actual processor assigned ownership. Referencing the `HOME` of a decomposition or array element in the `on` clause will cause the iteration to be assigned to the processor where that element is mapped. This is the case for the first two `FORALL` loops. Otherwise the `on` clause takes an expression to calculate a processor identifier between 1 and `n$proc`, and directly assigns each iteration of the loop to a specific processor. Arbitrary expressions are allowed in the processor, decomposition, or array subscripts. However, the user should be aware that complex expressions will be difficult for the compiler to implement efficiently.

6 Relationship to Other Research

A large number of researchers are investigating compilation for distributed-memory multiprocessors. Many of them have explored the problem of specifying data decompositions, and we have drawn upon their work. In particular, we have been influenced by alignment specifications and reduction functions from CM FORTRAN [TMC89] and structures to handle irregular distributions from PARTI [WSBH91] and Kali [KMV90, MV90]. Here we quickly describe other research in the area.

6.1 Single Array Decomposition

Some researchers concentrate on computations within loops that only involve a single array. These researchers do not need alignment or distribution specifications, since they automatically generate the data decomposition. Ramanujam and Sadayappan [RS89] examine both the data and iteration space to derive a combined task and data partition of the loop nest. Hudak and Abraham [HA90] find a stencil-based approach useful for analyzing communications and deriving efficient rectangular or hexagonal data distributions. These researchers do not discuss generating communication for the resulting distributions.

SPOT [SS90, Soc90] is a point-based SIMD data-parallel programming language for single arrays in loops. Computations are specified from the point of view of a single element in the array. This stencil-based approach allows the SPOT compiler to derive efficient *near-rectangular* data distributions. The compiler then generates computation and communication by expanding the single point algorithm to cover all points distributed onto a node.

6.2 Alignment

For computations involving multiple distributed arrays, both alignment and distribution must be dealt with in order to minimize data movement. The CRYSTAL compiler [CCL89, LC90a, LC90b] performs automatic data decomposition and communications generation for the CRYSTAL functional language. Heuristics are employed to align data arrays, both within and across dimensions. Different distributions are evaluated, then communication using CRYSTAL collective communication routines is generated. Since data decompositions are automatically calculated, no decomposition specifications are provided.

CM FORTRAN [AKLS88, TMC89] is a version of FORTRAN extended with vector notation, alignment, and data layout specifications. Programmers must explicitly specify data parallelism in CM FORTRAN programs by marking certain array dimensions as parallel. CM FORTRAN has a `FORALL` statement [ALS90] similar to the FORTRAN D `FORALL` loop, but which applies to only a single assignment statement. The `REDUCE` statement in FORTRAN D is patterned after equivalent reduction functions in CM FORTRAN.

CM FORTRAN does not possess distribution specifications since the operating system of the underlying SIMD distributed-memory machine provides the illusion of infinite machine size through the use of virtual processors. This approach has freed researchers to concentrate on deriving efficient data alignments [KLS88, TMC89, KLS90, KN90]. Proper alignment, including dynamic realignment, is especially important for SIMD machines, leading to a factor of 80-fold improvements in an example program [KN90]. More recently, researchers have also begun to study *strip mining* as a technique to avoid the inefficiencies of using virtual processors [Wei91].

AL [Tse90] is a language designed for the Warp distributed-memory systolic processor. The programmer utilizes `DARRAY` declarations to mark parallel arrays. The AL compiler then applies *data relations* to automatically align and distribute each `DARRAY`, detect parallelism, and generate communication. Only one dimension of each `DARRAY` may be distributed, and computations must be *linearly related.*

6.3 Distribution

Because of the lack of operating system support, research projects for MIMD distributed-memory machines have been forced to first tackle the data distribution problem. These systems do not directly specify alignments between arrays. Instead, they distribute each array individually and implicitly derive the alignment between different arrays based on their relative distributions. Systems such as DINO [RSW90], ID NOUVEAU [RP89], MIMDIZER [SWW91], OXYGEN [RA90], and PANDORE [APT90] all provide data distribution specifications equivalent to some combination of `BLOCK` and `CYCLIC`. DINO also supports special stencil-based data distributions with overlaps.

The ASPAR [IFKF90] compiler performs automatic data decomposition and communications generation. A *micro-stencil* is derived and used to generate a *macro-stencil* to identify communication requirements. Communications utilizing EXPRESS primitives are then automatically generated. In addition, reductions are identified and replaced with the appropriate EXPRESS *combine* operation. ASPAR derives simple `BLOCK` distributions; alignment specifications are not provided.

Gupta and Banerjee [GB90] propose a constraint-based approach to automatically calculate suitable data decompositions. They support `BLOCK` and `CYCLIC` distributions, but do not specify alignment. Instead of standard `BLOCK` distributions, SUPERB [ZBG88, Ger89, Ger90], SUSPENSE [RW88], and PARAGON [CR89, Ree90] support arbitrary user-specified contiguous rectangular distributions. SUPERB also originated the *overlap* concept as a means to both specify and store nonlocal data accesses.

Wolfe [Wol89, Wol90] describes transformations such as *loop rotation* for programs with `BLOCK` distributions. Callahan and Kennedy [CK88] propose methods for compiling programs with user-specified data distribution functions. They also demonstrate how such programs can be optimized using loop transformations. BOOSTER [PvGS90] also provides user-specified distribution functions defined as *program views*, but does not generate or optimize communications. Prins [Pri90] utilizes *shape refinement* in conjunction with linear transformations to specify data layouts and guide resulting data motion.

Finally, there are two systems that provide irregular data distributions to support irregular computations. PARTI [SBW90], a set of run-time library routines, is first to propose and implement user-defined irregular distributions [MSS$^+$88], as well as a hashed cache for nonlocal values [MSMB90]. PARTI has also motivated the ARF compiler which supports `BLOCK`, `CYCLIC`, and user-defined irregular distributions. Its goal is to demonstrate that *inspector* and *executor* loops for run-time preprocessing can be automatically generated by a compiler [KMSB90, WSBH91]. KALI [KMV90, MV90], a descendent of BLAZE, is the first compiler that supports both regular and irregular computations. It provides `BLOCK`, `CYCLIC`, `BLOCK_CYCLIC`, and user-specified data distributions. Like PARTI, KALI also uses an inspector/executor strategy to support run-time preprocessing of communication for irregularly distributed arrays [KMV90].

7 Conclusion

Programming languages lack support to efficiently exploit fine-grain data parallelism on distributed-memory machines. We believe that explicit data alignment and distribution specifications provide programmers and compiler writers with the correct paradigm for specifying data decompositions. We have designed FORTRAN D to be powerful enough to express most fine-grain parallel computations, but also simple enough that a sophisticated compiler can produce efficient programs for different parallel architectures. To make it usable for computational scientists, we have also made the meaning of FORTRAN D deterministic and quite close to sequential FORTRAN. In fact, any FORTRAN program is also a valid FORTRAN D program.

We are implementing a compiler to automatically convert FORTRAN 77D programs into message passing FORTRAN 77 programs that run on MIMD distributed-memory machines such as the Intel iPSC/860 [HKT91a, HKT91b]. In the process we will evaluate both the data decomposition specifications in FORTRAN D and the effectiveness of advanced compiler techniques for distributed-memory multiprocessors. We are also developing a FORTRAN 90D compiler for the iPSC/860 [WF91] and a FORTRAN 77D compiler for the Thinking Machines CM-2.

We are pursuing other projects in the FORTRAN D programming system [HKK$^+$91] at Rice and Syracuse. They include automatic data decomposition [BFKK90], static performance estimation using a machine-independent *training set* [BFKK91], compiler support for unstructured computations [KM90], run-time preprocessing using the PARTI communications library [SBW90], as well as a suite of applications programs to evaluate the effectiveness of the FORTRAN D compiler. We are also exploring the utility of additional language constructs to support parallel operations on high-level data structures such as trees or graphs.

8 Acknowledgements

We wish to acknowledge and thank Vasanth Balasundaram, Marina Kalem, John Mellor-Crummey, and Jaspal Subhlok for their many contributions to this work. In addition, we are grateful for the helpful comments from Hans Zima, Michael Gerndt, and the rest of the SUPERB research group.

References

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AKLS88] E. Albert, K. Knobe, J. Lukas, and G. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, New Haven, CT, July 1988.

[ALS90] E. Albert, J. Lukas, and G. Steele, Jr. Data parallel computers and the FORALL statement. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[APT90] F. André, J. Pazat, and H. Thomas. Pandore: A system to manage data distribution. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[BFKK90] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[BKK$^+$89] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[CCH$^+$88] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[CCL89] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Dept. of Computer Science, Yale University, New Haven, CT, December 1989.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[CR89] A. Cheung and A. Reeves. The Paragon multicomputer environment: A first implementation. Technical Report EE-CEG-89-9, Cornell University Computer Engineering Group, Ithaca, NY, July 1989.

[FO90] I. Foster and R. Overbeek. Bilingual parallel programming. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[FT90] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[GB90] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. Technical Report CRHC-90-14, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, October 1990.

[Ger89] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, December 1989.

[Ger90] M. Gerndt. Updating distributed variables in local computations. *Concurrency—Practice & Experience*, 2(3):171–193, September 1990.

[HA90] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. Technical Report TR91-154, Dept. of Computer Science, Rice University, March 1991.

[HKT91a] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. Technical Report TR91-156, Dept. of Computer Science, Rice University, April 1991.

[HKT91b] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report TR90-149, Dept. of Computer Science, Rice University, February 1991. To appear in J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*, Elsevier, 1991.

[HS86] W. Hillis and G. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[IFKF90] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[Kar87] A. Karp. Programming for parallelism. *IEEE Computer*, 20(5):43–57, May 1987.

[KLS88] K. Knobe, J. Lukas, and G. Steele, Jr. Massively parallel data optimization. In *Frontiers88: The 2nd Symposium on the Frontiers of Massively Parallel Computation*, Fairfax, VA, October 1988.

[KLS90] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[KM90] C. Koelbel and P. Mehrotra. Compiler support for unstructured scientific computations. Technical Report TR90-144, Dept. of Computer Science, Rice University, December 1990.

[KMSB90] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[KN90] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[LC90a] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[LC90b] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[LS91] S. Lucco and O. Sharp. Parallel programming with coordination structures. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, Orlando, FL, January 1991.

[MSMB90] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman. A scheme for supporting automatic data migration on multicomputers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[MSS$^+$88] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[MV90] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. ICASE Report 90-69, Institute for Computer Application in Science and Engineering, Hampton, VA, October 1990.

[PB90] C. Pancake and D. Bergmark. Do parallel languages respond to the needs of scientific programmers? *IEEE Computer*, 23(12):13–23, December 1990.

[Pri90] J. Prins. A framework for efficient execution of array-based languages on SIMD computers. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[PvGS90] E. Paalvast, A. van Gemund, and H. Sips. A method for parallel program generation with an application to the Booster language. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[RA90] R. Ruhl and M. Annaratone. Parallelization of FORTRAN code on distributed-memory parallel processors. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[Ree90] A. Reeves. The Paragon programming paradigm and distributed memory compilers. Technical Report EE-CEG-90-7, Cornell University Computer Engineering Group, Ithaca, NY, June 1990.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[RS89] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[RSW90] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, Dept. of Computer Science, University of Colorado, April 1990.

[RW88] Th. Ruppelt and G. Wirtz. From mathematical specifications to parallel programs on a message-based system. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[SBW90] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. ICASE Report 90-59, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1990.

[Soc90] D. Socha. Compiling single-point iterative programs for distributed memory computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[SS90] L. Snyder and D. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[SWW91] R. Sawdayi, G. Wagenbreth, and J. Williamson. MIMDizer: Functional and data decomposition; creating parallel programs from scratch, transforming existing Fortran programs to parallel. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors.* Elsevier, Amsterdam, The Netherlands, to appear 1991.

[TMC89] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 5.2-0.6 edition, September 1989.

[Tse90] P. S. Tseng. A parallelizing compiler for distributed memory parallel computers. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[Wei91] M. Weiss. Strip mining on SIMD architectures. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[WF91] M. Wu and G. Fox. Compiling Fortran90 programs for distributed memory MIMD parallel computers. CRPC Report CRPC-TR91126, Center for Research on Parallel Computation, Syracuse University, January 1991.

[Wol89] M. J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[Wol90] M. J. Wolfe. Loop rotation. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing.* The MIT Press, 1990.

[WSBH91] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. ICASE Report 91-13, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

A Fortran D Subset

This appendix suggests a subset of FORTRAN D that can simplify the implementation of a prototype compiler.

A.1 Dynamic Data Decomposition

The prototype may require that only one local data mapping can "reach" any reference to a distributed variable. This will simplify the job of communication generation. The compiler can ensure this by requiring all `ALIGN` and `DISTRIBUTE` statements to be unguarded.

```
REAL X(N)
DECOMPOSITION A(N)
ALIGN X with A
DISTRIBUTE A(BLOCK)
IF ( I .EQ. 1 ) THEN
    DISTRIBUTE A(CYCLIC)
ENDIF
X(I) = 1.0
```

For instance, this example will not be supported, because it determines at run-time whether X has a `BLOCK` or `CYCLIC` distribution.

A.2 Procedures

To reduce interprocedural analysis, the prototype compiler may place restrictions on procedures. First, to avoid calculating interprocedural reaching data decompositions, the prototype may require procedures to locally declare data decompositions for all distributed arrays accessed. Second, procedure calls may also be barred from `FORALL` loops.

A.3 FORALL Loops

With the assistance of dependence analysis, `FORALL` loops with regular computation patterns may be compiled into efficient code. However, there are irregular cases which will require run-time support to produce the proper results.

```
FORALL I = 1, N
    X(IDX(I)) = ...
    ... = X(IDX(I+1))
ENDFOR
```

For instance, in the previous example it is not possible to determine at compile-time whether there are any loop-carried true or output dependences that must be handled. As a result, code must be generated at run-time to ensure the semantics of the `FORALL` loop are observed by preprocessing the values of IDX(I). If the values of IDX(I) are not modified in the `FORALL` loop, an inspector prior to the loop will be sufficient to detect loop-carried dependences. Otherwise checking will be required on each iteration of the loop.

For ease of implementation the prototype compiler may wish to simply assume that no loop-carried true or output dependences exist for these loops. A compile-time warning should then list all references that may cause violations of `FORALL` semantics. Alternatively, special compile-time options may be provided to instruct the compiler to either generate code to ensure the proper results through some form of run-time resolution, or to generate debugging code that will detect when such violations occur.

A.4 Array Overflow

A similar problem exists with detecting array overflow that occur when attempting to access array elements that have not been mapped to a decomposition. Most cases may be detected at compile-time, but irregular or complex computations require run-time support. The prototype compiler may either provide a special compile-time option that generates code that assumes no array overflows, or an option that generates code to aid run-time detection of such overflows.

B Fortran D Examples

In this appendix we present some example FORTRAN D programs.

B.1 Red-Black Relaxation

```
DOUBLE PRECISION v(N,N), a, b

DECOMPOSITION d(N,N)
ALIGN v(I,J) WITH d(I,J)
DISTRIBUTE d(BLOCK,BLOCK)

DO k = 1, M
   // Compute the red points
   DO j = 1, N, 2
      DO i = 1, N, 2
         v(i,j) = a*(v(i,j-1) + v(i-1,j) + v(i,j+1) + v(i+1,j)) + b*v(i,j)
      ENDDO
   ENDDO
   DO j = 2, N, 2
      DO i = 2, N, 2
         v(i,j) = a*(v(i,j-1) + v(i-1,j) + v(i,j+1) + v(i+1,j)) + b*v(i,j)
      ENDDO
   ENDDO

   // Compute the black points
   DO j = 1, N, 2
      DO i = 2, N, 2
         v(i,j) = a*(v(i,j-1) + v(i-1,j) + v(i,j+1) + v(i+1,j)) + b*v(i,j)
      ENDDO
   ENDDO
   DO j = 2, N, 2
      DO i = 1, N, 2
         v(i,j) = a*(v(i,j-1) + v(i-1,j) + v(i,j+1) + v(i+1,j)) + b*v(i,j)
      ENDDO
   ENDDO
ENDDO
```

B.2 LU Decomposition

```
INTEGER ipvt(N)
DOUBLE PRECISION a(N,N), max, temp, da

DECOMPOSITION d(N,N)
ALIGN a(I,J) WITH d(I,J)
ALIGN ipvt(I) WITH d(I,I)
DISTRIBUTE d(*,CYCLIC)

DO k = 1, N-1
  // Find maximum element in column for use as pivot
  temp = 0.0
  DO i = 1, N-k+1
    REDUCE(max, temp, dabs(a(i,k)), ipvt(k), i)
  ENDDO

  // Swap diagonal element with pivot
  temp = a(ipvt(k),k)
  a(ipvt(k),k) = a(k,k)
  a(k,k) = temp

  // Divide remainder of column by pivot
  da = -1/a(k,k)
  DO i = 1, N-k
    a(k+i,k) = a(k+i,k) * da
  ENDDO

  // Reduce remaining columns
  DO j = k+1, N
    // Swap element in pivot column
    temp = a(ipvt(k),j)
    a(ipvt(k),j) = a(k,j)
    a(k,j) = temp

    // Reduce column
    DO i = 1, N-k
      a(k+i,j) = a(k+i,j) + a(k,j) * a(k+i,k)
    ENDDO
  ENDDO
ENDDO
```