The
Connection Machine
System

CM Fortran Programming Guide

Version 1.0
January 1991

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, November 1989
Revised, February 1990
Revised, January 1991

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

Part II Programming in CM Fortran

Part III Optimization

About This Manual

Objective

This manual introduces the array-processing features of CM Fortran in a task-oriented way. It is a companion volume to the *CM Fortran Reference Manual.*

Intended Audience

The reader of this manual is assumed to have a working knowledge of Fortran 77 (either VAX FORTRAN or Sun FORTRAN). No prior knowledge of CM Fortran or of the Connection Machine system is required.

Revision Information

This manual supersedes *Getting Started in CM Fortran*, Version 5.2–0.6, February 1990. A condensed version of this *Programming Guide* is now available as *Getting Started in CM Fortran,* January 1991.

Organization of This Manual

Part I Getting Started

A gentle introduction to the data parallel programming model, the basics of the Connection Machine system, and a simple program in CM Fortran.

Part II Programming in CM Fortran

The major array-processing features that CM Fortran adds to Fortran 77.

Part III Optimization

The techniques of optimizing CM Fortran programs by controlling the layout of arrays in the CM's distributed memory.

Appendix A Sample Programs

Several common problems solved in both Fortran 77 and CM Fortran.

Related Documents

- *Getting Started in CM Fortran.* Introduces the CM Fortran language. It is a condensed version of the present manual.
- *CM Fortran Reference Manual.* Defines the CM Fortran language, specifying the complete syntax and semantics of every feature and compiler directive.
- *CM Fortran User's Guide.* Describes the CM Fortran compiler command and switches, the CM Fortran library of utility procedures, and various program development tools.

  The CM libraries—scientific software, parallel I/O, visualization, and Paris (the CM Parallel Instruction Set)—are described in separate volumes.
- *CM Fortran Optimization Notes.* A pair of manuals that provide hints for getting best performance from CM Fortran programs. The two *Optimization Notes* manuals describe, respectively, the two execution models for which CM Fortran programs can be compiled.
- *CM Fortran Release Notes.* Summarizes the new features in the current release and lists restrictions and implementation errors.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a back-trace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
Usenet Electronic Mail:	ames!think!customer-support
Telephone:	(617) 234–4000 (617) 876–1111

Part I
Getting Started

Chapter 1

What Is CM Fortran?

The CM Fortran language is an implementation of Fortran 77 supplemented with array-processing extensions from the ANSI and ISO (draft) standard Fortran 90. These array-processing features map naturally onto the data parallel architecture of the Connection Machine (CM) system, which is designed for computations on large data sets. CM Fortran thus combines:

- The familiarity of Fortran 77, often the language of choice for scientific computing
- The expressive power of Fortran 90, which offers a rich selection of operations and intrinsic functions for manipulating arrays
- The computational power of the CM system, which brings thousands of processors to bear on large arrays, processing all the elements in unison

1.1 Array Processing in CM Fortran

The essence of the Fortran 90 array-processing features is that they treat arrays as first-class objects. An array object can be referenced by name in an expression or passed as an argument to an intrinsic function, and the operation is performed on every element of the array.

1.1.1 Compared with Fortran 77

In Fortran 77, operations are defined only on individual scalars. Operating on an array requires stepping through its elements, explicitly performing the operation on each one. With

Fortran 90 constructions, it is not necessary to reference array elements separately by means of subscripts, and it is not necessary to write DO loops or other such control constructs to have the operation repeated for each element. It is sufficient simply to name the array as an operand or argument.

1.1.2 Example in Fortran 77 and CM Fortran

Consider a 4-element array A, initialized to [1,2,3,4]:

```
      INTEGER A(4)
      DATA A  / 1, 2, 3, 4 /
```

Suppose you want to increment each of the values by 1, so that A contains [2,3,4,5]. The familiar method in Fortran 77 is to reference the elements by subscript and, through a looping construct, explicitly increment each value:

```
      DO 30 I=1,4
         A(I) = A(I) + 1
   30 CONTINUE
```

If the array is multidimensional, then the control sequence is nested to operate on all the elements:

```
      INTEGER A(4,4)

      DO 30 I=1,4
         DO 40 J=1,4
            A(I,J) = A(I,J) + 1
   40    CONTINUE
   30 CONTINUE
```

CM Fortran dispenses with the subscript references and the DO loops. Both the above operations are expressed simply as:

```
      A = A + 1
```

These code fragments perform the same set of operations, but their semantics are slightly different. The Fortran 77 statements are evaluated in the order specified by the nested loops, whereas the Fortran 90 construction allows the elements of **A** to be evaluated in any order, including simultaneously.

A Fortran 90 array reference can be used for any size or shape array and for any array operation defined in CM Fortran. The array could, for example, be 4-dimensional, and the operation could be any Fortran operator or intrinsic function:

```
REAL B( 512, 64, 8, 4 )

B = 8.0            ! Set all 1,048,516 elements to 8.0.
B = B * 2.0        ! All 1,048,516 elements contain 16.0.
B = SQRT( B )      ! All 1,048,516 elements contain 4.0.
```

NOTE

The simple array reference **A** or **B** is the default form of a Fortran 90 *triplet subscript.* A triplet subscript, such as **A(1:4:1)**, contains the information that Fortran 77 expresses in the control specification of a **DO** loop: the first and last elements and the increment.

Fortran 90 thus replaces **DO** loops with a form of array reference that indicates all the elements of interest. See Section 4.2 for more information about Fortran 90 array references.

This manual uses the term *array object* to mean any array that is referenced in the Fortran 90 manner. That is, an array object is one for which the array reference contains an explicit or implicit triplet subscript that indicates all the elements that are to be operated upon.

1.1.3 Compared with Fortran 90

CM Fortran implements the array-processing features of Fortran 90. Features proposed for Fortran 90 in the major areas other than array processing—such as pointers, structures, modules, and precision control—are not part of CM Fortran.

1.2 Data Parallel Processing

Fortran 90 array processing is reflected in the hardware of the data parallel Connection Machine (CM) system. From the software perspective, an array object refers to all the data elements of the array simultaneously. From the hardware perspective, the separate operations on the array's elements are all performed simultaneously.

1.2.1 Compared with Serial Processing

A serial implementation of Fortran 90 would have the syntactical convenience of referencing arrays as objects, but the compiler would necessarily generate serial loops. However, if the operations on individual data elements are independent of one another, there is no inherent need for them to be sequential. A computer that can store each data element in the memory of a separate processor is free to operate on all the data elements at the same time. For example, given a 40 x 40 x 40 array **A**, consider the statement:

```
    A = SQRT(A)/2
```

To execute this statement, a serial computer would need to perform 128,000 arithmetic computations. A data parallel computer, in contrast, provides a processor for each of the 64,000 data elements, and each processor needs to perform only two computations.

1.2.2 In the CM System

The CM system consists of a collection of simple processors, each with its own memory, all acting under the direction of a conventional processor called the front end. Arrays used in Fortran 90 constructions are stored in CM memory, one element per processor. Since many data sets are larger than even the largest CM, the system uses a *virtual processing* mechanism, whereby each physical processor simulates some number of virtual processors by subdividing its memory, to ensure that a processor is assigned to each array element.

When the front-end computer executes a CM Fortran program, it performs serial operations on scalar data stored in its own memory, but sends any instructions for array operations to the CM. When the CM receives an instruction, each (virtual) processor executes it on its own data point. Other instructions transfer data between the front end and the CM or between the CM and a peripheral storage device such as the DataVault mass storage system.

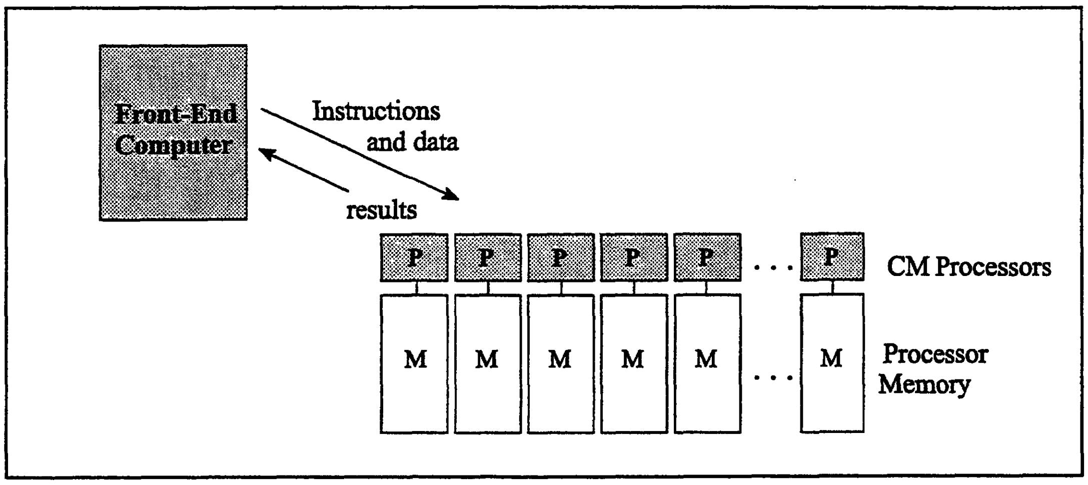


Figure 1. Interactions between front end and CM

Because CM memory is distributed among the processors, the system provides several mechanisms by which processors can access each other's memories. Interprocessor communication is transparent to the user, but many CM Fortran operations map onto communication instructions in a straightforward way. The three communication mechanisms are:

- Nearest-neighbor, or *NEWS*, communication, whereby each processor gets a value from its neighbor on an *n*-dimensional grid, all at the same time
- General-purpose, or *router*, communication, whereby each processor gets a value from any arbitrary processor, all at the same time
- Global communication, which includes cumulative computations along grid axes and reduction of an array to a single value

Notice that because there is only one instruction stream, the CM processors are naturally synchronized. Race conditions cannot develop because no processor proceeds to the next instruction until all have finished the current instruction. Processors for which the instruction is not relevant (because they have been used in a conditional construction, for instance) are *deactivated;* they do nothing until a later instruction reactivates them.

The array-processing constructions that CM Fortran has adopted from Fortran 90 map naturally onto this data parallel architecture. Although this manual does not focus on implementation issues, some of the basic programming practices it discusses follow clearly from even this brief introduction to CM architecture.

1.3 Memory Management in CM Fortran

CM Fortran is a superset of Fortran 77. The differences between the two languages reflect a basic fact of CM architecture: a CM Fortran program is directing two CM system components with different memory organizations. An array can have its *home* either in the centralized memory of the front end or in the distributed memory of the CM.

No new data structure is needed to express parallelism, and the programmer need not take any special action to invoke the CM. The CM Fortran compiler allocates arrays in the memory of one machine or the other *depending on how the arrays are used.* In brief:

- Arrays that are used *only* in Fortran 77 constructions reside on the front end. The front end stores and processes all scalar data, including subscripted arrays.
- Arrays that are used in Fortran 90 constructions reside on the CM. The CM stores and processes all arrays that are referenced as array objects. (Arrays that are referenced in both ways reside on the CM. See Section 2.3.3.)

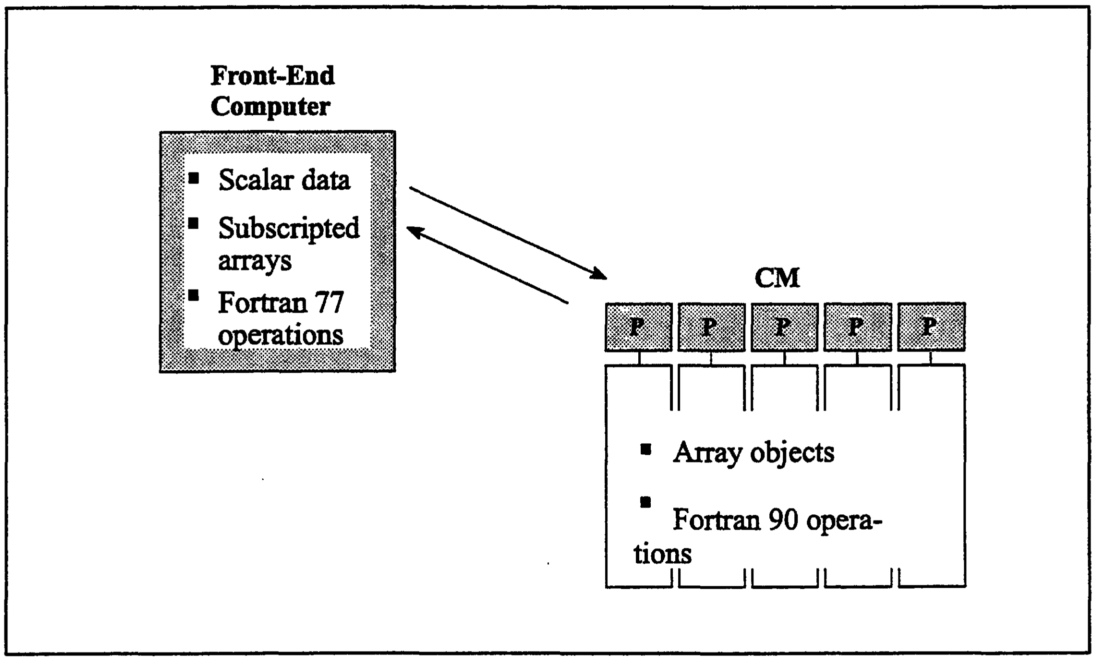


Figure 2. Division of labor between front end and CM

The CM Fortran programmer need not, of course, specify where data is stored (although compiler directives and switches do provide this capability). However, even the beginning user needs to understand the division of labor between the two machines so as to avoid trying to perform a CM (parallel) operation on front-end (scalar) data, and vice versa. Such pitfalls are pointed out throughout this manual.

Because of the CM's distributed memory, not all of Fortran 77 can be used with CM array objects. Most Fortran 77 features are extended for use with array objects, as the + and = operators are used above in the array operation `A = A + 1`. However, certain features with storage-order dependencies—most notably, the `EQUIVALENCE` statement—are not supported for CM data.

1.4 The Features of CM Fortran

The array-processing features that CM Fortran draws from Fortran 90 include:

- *Expanded semantics* for Fortran 77 operators and intrinsic functions, such that they can take an array object and operate on its elements
- *Array sections* and *vector-valued subscripts,* new syntax for selecting subarrays from array objects
- *The* `WHERE` *statement* and construct, which operate conditionally on an array's elements depending on the elements' values
- *New intrinsic functions* for permuting and transforming arrays, as well as for constructing arrays and inquiring about their properties
- *Attributed type declarations*, an alternative to Fortran 77 type declarations and the `DIMENSION` statement for declaring arrays

CM Fortran also includes some Fortran 90 features that are not specifically related to array processing, but are commonly found in implementations of Fortran 77. Examples of these are the control-flow statements `CASE`, `DO TIMES`, `DO WHILE`, and `END DO`. The *CM Fortran Reference Manual* gives a complete description of all CM Fortran features.

Finally, CM Fortran includes some non-standard features that are particularly useful for data parallel programming on the Connection Machine system:

- *The* FORALL *statement*, a powerful facility for initializing arrays, for selecting subarrays, and for specifying data movement in terms of array indices.
- *Compiler directives*, several of which control the layout of arrays in Connection Machine memory. Layout can have major effects on program performance.
- *Utility routines library*, which serves a number of purposes:
    - Providing language-level capabilities that are not yet implemented in CM Fortran, such as generating random numbers in an array
    - Providing CM system services, such as transferring arrays between front-end memory and CM memory or accessing the CM (parallel) file system on a peripheral storage device
    - Improving performance in cases where the CM Fortran compiler has a temporary "blind spot" and cannot translate language syntax into the optimal parallel instruction

1.5 CM Fortran Documentation

- *CM Fortran Reference Manual* defines the language and the compiler directives.
- *CM Fortran User's Guide* describes the compiler and its switches, some development utilities such as the timer and debugger, and the library of utility routines.
- *CM Fortran Optimization Notes* describe the mapping of CM arrays onto the underlying machine and provide some hints about efficient programming practices.

NOTE

Please see the current *CM Fortran Release Notes* for any restrictions or bugs in the features described in this manual.

Chapter 2

Basic Operations

This chapter examines some simple CM Fortran code to illustrate the operations that are fundamental to any array-processing program:

- Declaring arrays
- Initializing or otherwise moving data into arrays
- Computations on arrays
- Retrieving the results of computations
- Compiling and executing a program

For simplicity, this chapter focuses on *elemental operations* on *whole arrays.*

- An elemental operation affects the elements of an array as if it had been applied separately to each element (in undefined order). Such an operation occurs within each CM processor independently of the others. Operations that specify data movement between processors are deferred to Chapter 6.
- A whole array is an array object specified simply by name, which indicates all the elements of the array. Such a reference has an implicit triplet subscript that corresponds to the declared bounds of the array. Chapter 4 shows operations that apply only to selected elements of an array.

2.1 A Simple Program

The following program shows all the basic operations noted above. It declares and initializes two vectors, computes the sum of the squares of their corresponding elements, and prints out the result vector and the result vector's highest value. The remainder of this chapter steps through this program, pointing out the basic features of CM Fortran.

```
PROGRAM SIMPLE

IMPLICIT NONE
INTEGER A, B, C, N, MAXVALUE
PARAMETER (N=5)
DIMENSION A(N), B(N), C(N)

DATA A / 1,2,3,4,5 /
B = 2                         ! a CM array assignment

C = A**2 + B**2               ! array-valued expressions

PRINT *, 'Array C contains:'
PRINT *, C                    ! output of CM data

MAXVALUE = MAXVAL( C )        ! a CMF intrinsic function
PRINT *, 'The largest value in C is ', MAXVALUE

STOP
END
```

2.2 Declaring and Initializing Arrays

CM Fortran supports all standard Fortran 77 syntax for declaring and initializing both scalar values and arrays.

Program **simple.fcm** uses Fortran 77 type specification statements, as well as the statements **DIMENSION**, **PARAMETER**, and **DATA**, to declare and initialize the scalar values **N** and **MAXVALUE** and the array **A**. Since CM Fortran supports standard Fortran I/O features on the UNIX file system, programs can also use the **READ** statement for initialization. (To read data from the CM file system, see the volume *Connection Machine I/O Programming* in the CM documentation set.)

CM Fortran supports seven data types:

`CHARACTER`	**`REAL`**
`LOGICAL`	**`DOUBLE PRECISION`** (real)
`INTEGER`	**`COMPLEX`**
	`DOUBLE COMPLEX` (double-precision complex)

The **`IMPLICIT`** statement allows the programmer to override Fortran's implicit typing rules with specified typing rules, each of which associates one or more letters with a data type. The form used in program **`simple`**, **`IMPLICIT NONE`**, means that all identifiers must be declared. This form is useful for catching misspellings at compile type.

Arrays can be of any type, and from one to seven dimensions. However, character arrays are not supported on the CM. That is, an array declared as type character is always stored in the memory of the front-end computer, and its elements are processed serially in the Fortran 77 manner. All other arrays can be stored and processed either on the front end or on the CM depending on the use the program makes of them.

As an alternative to the Fortran 77 syntax shown, CM Fortran also offers the more economical Fortran 90 syntax. The new declaration syntax can specify type, dimensionality, and initial values all in a single statement (see Chapter 3).

2.3 Array Operations

An array operation is any reference to an array object (that is, any reference using Fortran 90 syntax) in an expression, assignment, or intrinsic function call. Fortran 90 and CM Fortran extend the semantics of Fortran 77 such that operators and intrinsic functions can take array objects and operate on their elements. CM Fortran also adds a number of Fortran 90 intrinsic functions that manipulate or transform array objects.

The various forms of array operations are all illustrated in program **`simple`**:

```
B = 2                       ! a CM array assignment
C = A**2 + B**2             ! array-valued expressions
PRINT *, C                  ! output of CM data
MAXVALUE = MAXVAL( C )      ! a CMF intrinsic function
```

Notice the use of the Fortran 77 operators +, =, and **. These features have been extended to operate on all the elements of an array object. Similarly, array objects can be passed as arguments to all the Fortran 77 intrinsic numeric and mathematical functions, such as **ABS**, **MAX**, and **SIN**. However, because character arrays are not supported on the CM, the intrinsic character functions, such as **CHAR** and **INDEX**, cannot take array objects as arguments.

The function **MAXVAL** is an example of the array-processing intrinsic functions that CM Fortran adds to Fortran 77. The array argument(s) to these functions are taken to be CM array objects.

2.3.1 Conformable Arrays

When an expression or assignment involves two or more arrays, the arrays must be *conformable,* that is, they must be of the same size and shape. Each set of corresponding elements of conformable arrays resides in the memory of a single CM processor, which performs the computation on that set of elements.

For each group of conformable arrays, the system configures a set of virtual processors into a logical grid that reflects the shape of the arrays. Although arrays of many different sizes and shapes can coexist in CM memory, conformable whole arrays are always allocated in the same set of processors in the same order.

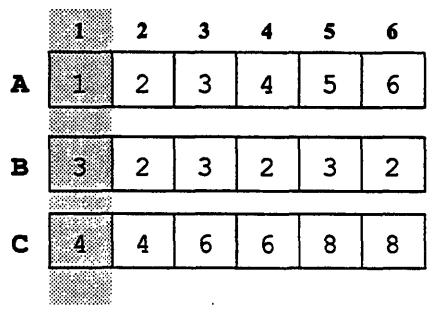


Figure 3. An elemental vector addition, **C = A + B**, with one processor highlighted

As a result, elemental operations on conformable arrays are extremely efficient. Such operations need not move data into the appropriate processors: it is already there. Each processor need only index within its own memory to locate the operands. For example, Figure 3 highlights the memory of a single processor as the CM adds the corresponding elements of two vectors, **A** and **B**, and places the results in a third vector, **C**.

The comparable operation on matrices is shown in Figure 4, assuming three 6 x 4 arrays. As in the figure above, a set of virtual processors is configured to reflect the shape of the arrays, and each processor has a set of corresponding elements within its own memory. Each processor does exactly the same operation it would do for a vector addition; the difference is that the processors are configured differently.

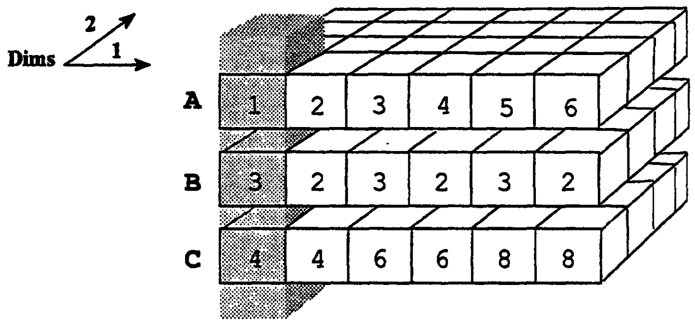


Figure 4. An elemental matrix addition, **C** = **A** + **B**, with one processor highlighted

This manual adopts the Fortran convention of depicting array elements in column-major order, although there is no implication that successive column elements are in contiguous memory locations. That is, element `A(2,1)` is not necessarily contiguous with element `A(3,1)` in CM memory, as it would be in the memory of a serial computer. Instead, the CM virtual processor that contains elements `(2,1)` of all three conformable arrays is considered a nearest neighbor of the virtual processor that contains elements `(3,1)` of all three arrays. Moving data from one array position to another thus entails interprocessor communication, as described in Chapter 6.

NOTE

CM programmers should avoid the common Fortran 77 practice of declaring one large array and reshaping pieces of it as needed. With distributed memory, and given that arrays of different shapes are allocated in different sets of virtual processors, this practice can lead to unexpected results.

It is preferable instead to declare all the needed arrays separately, each in the desired shape. If you need to reshape a particular array —for instance, to change **A(N,M)** to **A(N*M)**—use the intrinsic function **RESHAPE** (see Chapter 7).

2.3.2 Scalar Extension

Scalars may be intermixed freely in expressions that have array-valued components. When a scalar appears in such an expression, it is treated as if it were an array conformable with the other array(s) in the expression. For example:

```
A = B/5
C = D * 3.14159
```

The first statement divides each element of **B** by the constant 5 and assigns it to the corresponding element of **A**. In the second statement, each element of **C** gets the circumference of a circle whose diameter is **D**.

Recall that scalars are stored in front-end memory. When the CM system encounters a scalar in an array-valued expression, the front end "broadcasts" the value to all the CM processors. (The scalar value serves as an immediate operand; no CM space is allocated for it.) In effect, a scalar is conformable with any array and with different arrays in different expressions.

Scalar extension ("broadcasting") is not only good use of CM memory; it is also extremely efficient. It is a waste of memory for a program to store a separate copy of a constant in every processor, since broadcasting a value from the front end takes no more time than accessing it in CM memory. In fact, assuming that array **A** has been set to 5.0, the first of these statements is much faster than the second:

```
B = B * 5.0

B = B * A
```

2.3.3 Array Homes

The machine on which an array is allocated is called its *home*. Aside from character arrays, which always reside on the front end, an array's home is determined by how the array is used *within a program unit.* A program unit is a main program, a subroutine, or a function.

- The front end stores all scalar data, including arrays that are referenced *only* as subscripted variables (in the Fortran 77 way) within a program unit. All serial operations, including looping operations on array elements, execute on the front end. Essentially, the front end executes all of CM Fortran that is Fortran 77.

  This is a front-end operation:

  ```
      INTEGER A(4)
      DATA A  / 1, 2, 3, 4 /

      DO 30 I=1,4
         A(I) = A(I) + 1        ! A is allocated on front end
   30 CONTINUE
  ```

- The CM stores all arrays that are referenced *at all* as array objects within a program unit. All operations on array objects execute on the CM. Essentially, the CM executes all of CM Fortran that is drawn from Fortran 90.

  This is a CM operation:

  ```
      INTEGER A(4)
      DATA A  / 1, 2, 3, 4 /

      A = A + 1                 ! A is allocated on CM
  ```

- The CM stores all arrays that are referenced *both* as array objects and as subscripted variables within a program unit, although the serial operations *execute* on the front end.

This is a (rather pointless) *mixed-home* operation:

```
   INTEGER A(4)
   A = 5                      ! A is allocated on CM

   DO 30 I=1,4
      A(I) = A(I) + 1         ! Loop executes on front end
30 CONTINUE
```

These mixed-home operations tend to be inefficient, since the system moves the CM array one element at a time to the front end to perform the serial operation. If the algorithm demands a mixed-home operation, it is often advisable to use the CM Fortran utility routines **CMF_FE_ARRAY_TO_CM** and **CMF_FE_ARRAY_FROM_CM**, which copy an array *en masse* from one machine to the other. (See the *CM Fortran User's Guide* for information on CM Fortran utility routines.)

- Arrays declared as common reside on the CM unless otherwise specified with a compiler directive or switch. Discussion of common arrays is deferred until the chapter on subroutines, Chapter 5.

CM programmers need to be aware of where particular arrays are allocated so as to avoid using mixed-home operations unintentionally. Using array operations on CM arrays gives the program the performance benefits of parallelism, but using a front-end looping operation on a CM array can exact a high cost in performance as the system moves the array element by element from one machine to the other.

It is crucial to consider arrays' homes when calling subroutines, since CM Fortran requires that the home of an actual array argument be the same as the home of a dummy argument in the procedure. However, arrays' homes *are determined at compile time separately for each program unit.* The programmer must therefore be aware of the homes of arrays used as arguments and must often take explicit steps to ensure that the homes of actual and dummy arguments match. See Chapter 5 for more information on arrays as arguments and on controlling their homes.

2.4 Retrieving CM Data

CM Fortran supports all Fortran I/O operations—the **READ**, **WRITE**, and **PRINT** statements—for both front-end and CM data.

There are various ways to retrieve and display the results of CM computations. Some intrinsic functions, such as **MAXVAL** or **SUM**, perform a combining operation on an array's elements and return the scalar result to the front end. The result of a *reduction* function such as this can, like any other scalar, be displayed with a **PRINT** statement. As shown in program **simple.fcm**:

```
      INTEGER MAXVALUE
      . . .
      MAXVALUE = MAXVAL( C )          ! a CMF intrinsic function
      PRINT *, 'The largest value in C is ', MAXVALUE
```

You can also retrieve a scalar value by subscripting a CM array in the Fortran 77 fashion to indicate the desired element. Notice that this is a deliberate use of the mixed-home construction: the array element that is referenced with a Fortran 77 subscript is automatically moved to the front end, where it is displayed by the **PRINT** statement:

```
      PRINT *, 'The third element of array C is ', C(3)
```

You could also use the **PRINT** statement to view all the results stored in array **C**, since Fortran I/O statements are extended for use with CM data. As shown in program **simple.fcm**:

```
      C = A**2 + B**2             ! array-valued expression
      PRINT *, 'Array C contains:'
      PRINT *, C                  ! output of CM data
```

For large vectors or for matrices, a **FORMAT** statement might be used to improve the readability of the output:

```
      INTEGER, ARRAY (4,4) :: D
      . . .
      PRINT 10, D
   10 FORMAT (4I9)
```

2.5 Compiling and Executing

To compile and execute a CM Fortran program:

1. Place the program in a file with the filename extension **.fcm**.
2. Compile the file with the CM Fortran compiler command **cmf**.

```
% cmf simple.fcm -o simple
```

3. Execute the program by specifying it as an argument to the CM System Software command **cmattach**.

   The following command line attaches the front end logically to the CM, making some default number of processors available to execute program **simple**.

```
% cmattach simple

Attaching the Connection Machine system NAME...
cold booting... done.
Attached to 8192 processors on sequencer 0
Paris safety is off.
Array C contains:
         5         8        13        20        29
The largest value in C is  29
 FORTRAN STOP
Detaching... done.
```

See the *CM Fortran User's Guide* for more information about compiling and executing CM Fortran programs.

2.5.1 Specifying Execution Model

CM Fortran programs can be compiled for either of two execution models, specified with the switches **-paris** and **-slicewise**. The default execution model is determined locally at installation time.

The execution models differ in the way they use the underlying CM hardware. Programs compiled for the slicewise execution model generate special optimizations for CM systems equipped with the optional 64-bit floating-point accelerator; such programs can execute *only* on CM systems that include this hardware option. Programs compiled for the Paris execution model lack the special optimizations, but they can execute on any CM hardware configuration.

2.5.2 Incorporating Existing Routines

Object modules generated by Fortran 77 compilers, such as Digital Equipment Corporation's VAX FORTRAN compiler and Sun Microsystems' Sun FORTRAN compiler, may be linked with modules produced by the CM Fortran compiler. This facility is very useful for incorporating existing library routines into a CM Fortran application, as well as supporting the incremental conversion of an application from serial code to parallel array operations. Procedures compiled by foreign Fortran compilers will not, of course, make use of the CM processors.

Part II
Programming in CM Fortran

Chapter 3

Array Declarations and Initial Values

CM Fortran supports all the Fortran 77 syntax for declaring and initializing arrays. Any array that is declared and/or initialized in the Fortran 77 manner can be used in an array operation; as always, such use will cause the array to be allocated in CM memory and processed in parallel.

This chapter introduces a number of additional CM Fortran features, many drawn from Fortran 90, for declaring and initializing arrays and, in some cases, scalars as well. These features include:

- *Attributed type declarations,* which specify not only an object's type but also such attributes as **ARRAY** (dimensionality), **DATA**, or **PARAMETER**. These attributes substitute for a separate **DIMENSION**, **DATA**, or **PARAMETER** statement.
- *Array constructors,* which specify the values of a 1-dimensional array. An array constructor can be used, for example, to initialize a vector or to pass a vector of values as an argument to an intrinsic function or external procedure.
- Dynamic initialization by means of array assignments and a CM Fortran utility routine for generating random numbers.

NOTE

Declaration syntax does not determine where an array is allocated or what operations may be performed on it. The choice of syntax is entirely a matter of style and convenience; it does not affect program behavior.

This chapter focuses on declaring and setting the initial values of local arrays, including arrays that are local to the main program and passed as actual arguments. The subject of common arrays is deferred until the discussion of subroutines in Chapter 5.

3.1 Declaring Arrays

A Fortran 90 attributed type declaration is a single statement that can associate several attributes besides type with the object being declared. One of these attributes is **ARRAY**, which has the same semantics as the **DIMENSION** statement.

3.1.1 Using the ARRAY Attribute

An attributed type declaration with the attribute **ARRAY** has the form:

type , **ARRAY (*dims*)** :: *name* [, *name* , ...]

For example, the declaration

```
INTEGER, ARRAY(10) :: ARRAY_NAME
```

is equivalent to the two Fortran 77 statements

```
INTEGER ARRAY_NAME
DIMENSION ARRAY_NAME(10)
```

Attributed type declarations are a convenient way to declare several conformable arrays:

```
REAL, ARRAY(512) :: A,B,C,D         ! four conformable vectors

COMPLEX, ARRAY(100,100) :: E,F,G,H ! four conformable matrices
```

3.1.2 Array Properties

Declarations of the **ARRAY** attribute describe two properties of the arrays:

- *Rank* (the number of dimensions)
- *Shape* (the extents of the dimensions in rank order)

Scalars have rank 0, and arrays can have rank 1 through 7. Vectors **A**, **B**, **C**, and **D** shown above have shape [512]; matrices **E**, **F**, **G**, and **H** all have shape [100,100]. The *size* of an array is the product of the extents of its dimensions. Thus, the vectors are all of size 512, and the matrices are of size 10,000.

As in Fortran 77, the extent of a dimension can be specified with a lower bound as well as an upper bound. For example:

```
INTEGER, ARRAY(-40:100) :: CTEMP
```

This statement declares an array element for each Celsius temperature between the point where the Celsius and Fahrenheit scales converge and the boiling point.

3.2 Defining Named Constants

A type declaration with the attribute **PARAMETER** defines a named constant. Like the statement **PARAMETER**, this attribute can be used only for scalar objects, not for arrays.

A declaration with the attribute **PARAMETER** has the form

type , **PARAMETER** :: *name* = *constant-expression*

For example:

```
INTEGER, PARAMETER :: ONE = 1
REAL, PARAMETER    :: PI = 22.0 / 7.0
LOGICAL, PARAMETER :: T = .TRUE., F = .FALSE.
```

3.3 Setting Array Values

Arrays can be initialized with a **DATA** statement, as in Fortran 77, or with the **DATA** attribute in a type declaration. The values of a **DATA** attribute for an array are specified with a CM Fortran feature called an *array constructor.*

As with declaration syntax, the form chosen for static initialization does not affect the home of an array or the operations that may be performed on it.

Initial array values can also be set dynamically by means of assignment. CM array objects can be assigned array constructors, or they can be seeded with random values by means of a CM Fortran utility routine. These operations cannot be performed on front-end arrays.

3.3.1 Assignment

As in Fortran 77, arrays can be initialized dynamically by assigning a value to each element. However, in CM Fortran the assignment can also be an array operation. For example:

```
INTEGER, ARRAY(5) :: A, B

DO I=1,5
   B(I) = 0              ! a front-end loop assignment
END DO

A = 0                    ! a CM array assignment
```

Like any array operation, the array assignment that initializes **A** guarantees that **A** will be allocated on the CM. Array **B**, initialized with a serial loop, will be allocated on the front end *unless* it is used in an array operation elsewhere in the program unit. (Notice that if **B** is allocated on the CM, the use of a loop to initialize it is much less efficient than an array operation would be.)

The values assigned to a CM array can be any expression of the appropriate type, including values retrieved by a **READ** statement or those generated by an array constructor (see Section 3.3.4).

Initialization is one of several uses of the specialized CM Fortran assignment statements **FORALL**, or *elemental array assignment,* and **WHERE**, or *masked array assignment.* These statements are described later in this manual.

3.3.2 CMF_RANDOM Subroutine

CM Fortran does not yet implement the Fortran 90 intrinsic function **RANDOM** for filling a numeric array with random values. For the interim, CM Fortran provides a utility procedure:

```
CMF_RANDOM( ARRAY, LIMIT )
```

The argument array must be an array object, residing on the CM. This procedure cannot be used with an array that is used only in the Fortran 77 manner (with scalar subscripts) in the program unit.

The range of values generated by **CMF_RANDOM** depends on the type of the argument array. If **ARRAY** is of type integer, then the **LIMIT** argument serves as an exclusive upper bound. That is, the array is filled with random values in the range 0 to LIMIT – 1. If **ARRAY** is real or double-precision real, then **LIMIT** should be 1.0 and the values are in the range 0.0 to 1.0 (exclusive). For example:

```
DIMENSION I(100), A(50,50)
CALL CMF_RANDOM( I, 1000 )      ! range: (0, 999)
CALL CMF_RANDOM( A )            ! range: (0.0, 1.0)
```

See the *CM Fortran User's Guide* for more information about this and other utility procedures that generate random numbers.

3.3.3 DATA Statement

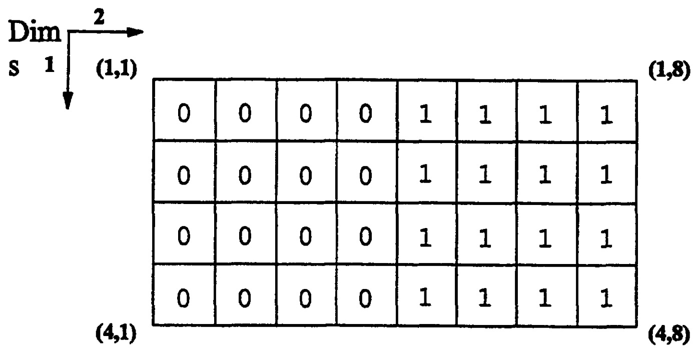
A **DATA** statement can be used in the Fortran 77 fashion to initialize both scalars and arrays. It does not matter which syntax was used to declare the object.

```
INTEGER I,A(5),B(10)
DATA I /1000/
DATA A / 1,2,3,4,5 /
DATA B / 10*0 /
```

As in Fortran 77, a **DATA** statement may also have an implied **DO** loop, which expands under the control of index variables to form a sequence of values. (The effect is similar to using a loop construct for dynamic initialization.) For example, given a 4 x 8 array **D**, the following two **DATA** statements initialize the left half to 0 and the right half to 1:

```
INTEGER, ARRAY(4,8) :: D

DATA ( ( D(I,J), J=1,4 ) I = 1,4 ) / 16*0 /
DATA ( ( D(I,J), J=5,8 ) I = 1,4 ) / 16*1 /
```



3.3.4 Array Constructors

An *array constructor* is a means of constructing an unnamed, 1-dimensional array by specifying an arbitrary sequence of scalar values.

Assigning an array constructor to a vector serves to initialize the vector with that sequence of values; the assignment is a CM operation, which causes the vector being assigned to be allocated on the CM. An array constructor may also be passed as an argument to a procedure, which has the same effect as passing a CM vector as an argument.

Array constructors can be specified in several ways. The simplest form is a list of the desired values, separated by commas and enclosed in square brackets:

```
INTEGER, ARRAY(4) :: V

V = [ 10, 20, 30, 40 ]
```

If the values are not of the same type, all values are coerced to the type of the first value:

```
R = [ 10.0, 20, 30.0d0, 40 ]      ! R must be type REAL

C = [ (0.0,0.0), 1, 2.0, 3 ]      ! C must be type COMPLEX
```

Another form of array constructor uses a repeat count syntax, shown here with 100-element vectors:

```
A = [ 50[1], 50[0] ]           ! 50 1's and 50 0's

B = [ 25[ 1.0,2.0,3.0,4.0 ] ] ! sequence repeated 25 times
```

Array constructors can also take a form resembling the control variables of a DO loop or a triplet subscript:

[*first* : *last* : *increment*]

This form of array constructor uses only integer values. It builds a vector of values in the sequence specified; the resulting vector may be coerced to another type. For example,

```
C = [1:100]                    ! integers from 1 to 100

D = REAL( [2:200:2] )          ! even reals from 2 to 200

E = CMPLX( [0:99],[0:99] )     ! sequence of 100 complex nos.

F = SQRT( [1:500:5] )          ! sequence of real square roots
```

3.3.5 DATA Attribute

The **DATA** attribute in a type declaration serves to initialize a scalar variable or an array.

The form for a scalar declaration is

type , **DATA** :: *name* = *value*

For example,

```
INTEGER, DATA :: SIZE = 1024
```

To declare and initialize an array, the declaration associates the **DATA** attribute with an array constructor. Since array constructors are always 1-dimensional, the **DATA** attribute can be applied only to 1-dimensional arrays. The form is

```
type, ARRAY (dim), DATA  :: name = [ array-constructor ]
```

For example,

```
INTEGER, ARRAY(10), DATA :: J = [ 0,9,4,7,2,8,6,5,1,3 ]

REAL, ARRAY(10), DATA :: R = REAL( [0:9] ), B = [0.0,1:9]

DOUBLE PRECISION, ARRAY(10), DATA :: D = [5[0.0d0],5[1.0d0]]

COMPLEX, ARRAY(10), DATA :: C = [ (0,0), 1:9 ]
```

As these examples suggest, the different forms of array constructor provide some flexibility in specifying sequences of values as the **DATA** attribute of an array. However, unlike a **DATA** statement, a **DATA** attribute cannot be associated with an implied **DO** loop.

When used in a type declaration, an array constructor is not considered an array operation and has no effect on the array's home. The arrays **J**, **R**, **B**, **D**, and **C** could be allocated either on the front end or on the CM, depending on how they are used in the program unit.

As shown in the case of **R**, the array constructor can be passed as an argument to a type-conversion function in the declaration. However, other elemental and transformational intrinsic functions—such as **RESHAPE**—cannot be used in specification statements.

3.4 A Note on Common Arrays

The features described in this chapter for declaration and dynamic initialization (assignment) apply to arrays in common blocks as well as to local arrays. However, the homes of common arrays are determined differently from those of local arrays, and the home can affect the method of *statically* initializing a common array. CM Fortran's treatment of common arrays is decribed in conjunction with subroutines in Chapter 5.

Chapter 4

Selecting Array Elements

The operations discussed so far have affected all the elements of an array object. CM Fortran offers the two Fortran 90 methods for selecting only certain array elements for an operation:

- By value: Conditional (or *masked)* operations include or exclude elements depending on their values.
- By position: Operations on an *array section* affect only a subarray of elements specified by their positions along each dimension of the parent array.

Another method of selecting subarrays, the FORALL statement, can select elements both by value and by position. FORALL is described in Chapter 8. As with whole arrays, the language references all the elements of an array section or a masked array at once, and the Connection Machine system processes the elements in parallel.

4.1 Conditional Operations

Operations that are conditional on the element values of (CM) array objects include:

- The WHERE statement and construct, which can be used to make an array assignment conditional
- CM Fortran intrinsic functions that take a MASK argument
- Front-end (Fortran 77) conditionals or loop constructs that use a scalar value returned from the CM as a control specification

4.1.1 The WHERE Statement

The CM Fortran **WHERE** statement is similar to the Fortran 77 **IF** statement. The **WHERE** statement tests array elements (in parallel) and then performs an assignment of those that meet the specified condition. The format is:

WHERE **(** *mask-expression* **)** *array-assignment*

For example, to avoid division by zero in an elemental array operation:

```
WHERE ( A.NE.0 ) C = B/A
```

When executing this statement, the system creates a logical array containing the results of the relational operation **.NE.** It uses this array as a mask to deactivate the processors that contain 0 values for **A**, while the remaining processors perform the division (see Figure 5). **WHERE** is thus called a *masked array assignment.*

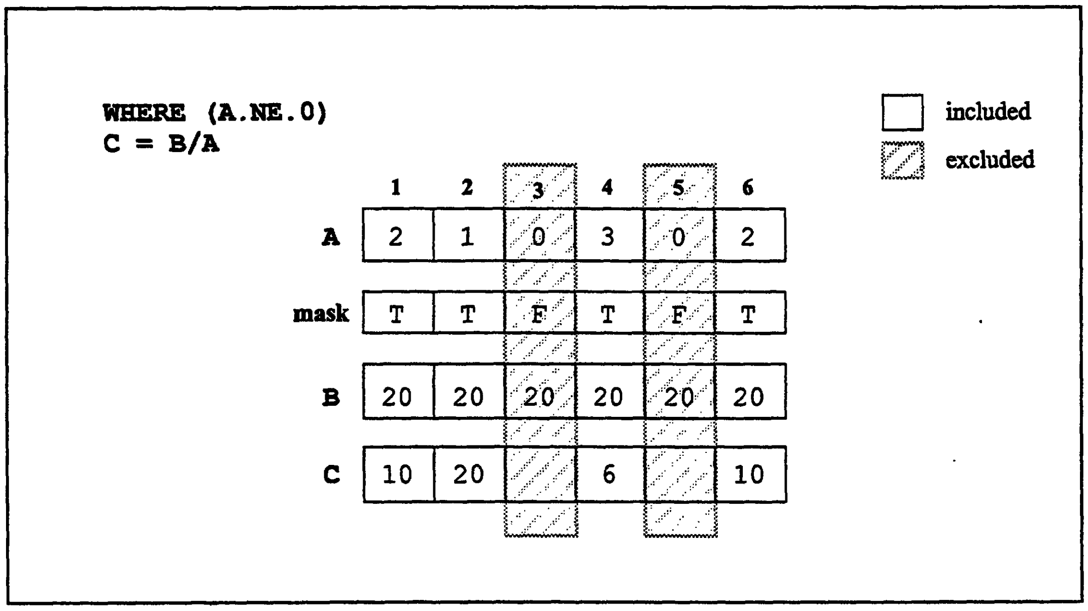


Figure 5. Processors deactivated in a masked array assignment

Notice that arrays can be masked by an expression that does not reference them, but instead references another (conformable) array:

```
WHERE ( A.NE.0 ) C = B**2
```

4.1.2 The WHERE Construct

Like the **IF** statement, the **WHERE** statement is expanded into a construct with the **END WHERE** statement and an optional **ELSEWHERE**. The **WHERE** construct is comparatively restricted: Fortran 90 does not define nested **WHERE** statements, and the body of the construct can contain only array assignments. A simple **WHERE** construct is:

```
WHERE ( A.GE.0 )
   B = SQRT( A )
ELSEWHERE
   B = 0
END WHERE
```

When executing this construct, the system computes a logical mask to screen out negative values of **A** from the **SQRT** operation. It then inverts the mask, reversing the true and false values, to activate only the processors that have negative values for A. The newly active processors then assign 0 to **B**.

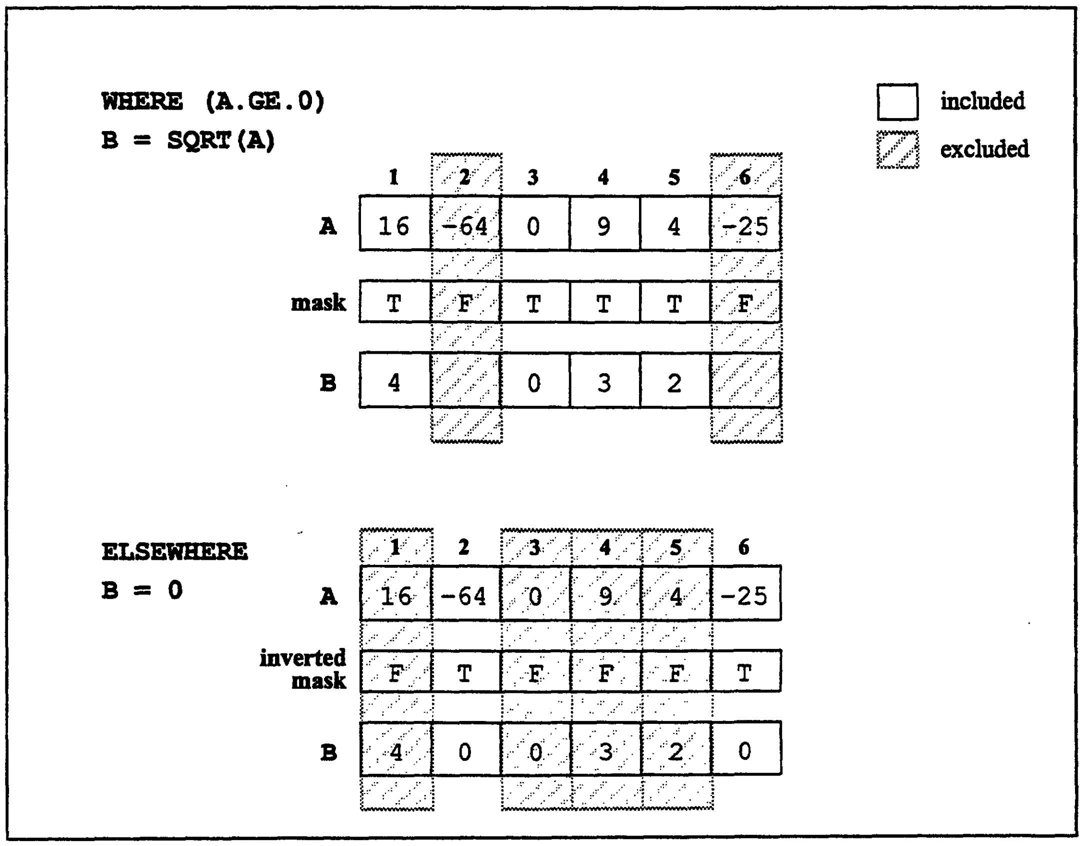


Figure 6. Inversion of the mask array in a **WHERE-ELSEWHERE** construct

4.1.3 Conditional Intrinsic Functions

Many of the array-processing intrinsic functions (described in Chapter 7) perform a masking operation similar to the action of the **WHERE** statement. With these functions, the mask is supplied as an argument. For example, to take the product of the non-zero values of array **A**, invoke the function **PRODUCT** with a mask that eliminates the zero values:

```
SCALAR_PRODUCT_OF_A = PRODUCT( A, A.NE.0 )
```

If a masked intrinsic function is used within a **WHERE** statement, the two masks are completely separate. The **WHERE** mask has no effect on the evaluation of the function reference; the function's mask argument has no effect on the evaluation of the **WHERE** assignment. For example:

```
WHERE ( A.LE.10 ) B = A + PRODUCT( A, A.NE.0 )
```

The function reference in this example computes the product of **A**'s values, excluding any zero values but *not* excluding values over 10. This scalar product is then replicated to conform in shape to array **A** and added (elementally) to any elements that are less than or equal to 10—including any zero values. Thus:

- An array element of 12 would be included in the computed product but *not* included in the addition and assignment operations; it is excluded by the **WHERE** mask but not by the function's mask argument.

- An array element *of 0 wou*ld *not* be included in the product but would be included in the addition and assignment; it is excluded by the function's mask argument but not by the **WHERE** mask.

The behavior of masked intrinsic functions within a **WHERE** statement is a CM Fortran extension to the standard language. Fortran 90 permits *only* elemental functions, such as **SQRT**, in **WHERE** statements. None of the elemental functions takes a mask argument.

4.1.4 Front-End Conditionals

A scalar value derived from a CM array can be used in a serial (front-end) construction to control a conditional or iterative operation. The scalar can be either:

- The result of scalar subscripting of a CM array, such as **CM_ARRAY(3,10)**
- The scalar result returned by an intrinsic reduction function, such as **ANY** or **SUM** (see Chapter 7)

For example, this **IF** construct uses the logical result of the reduction function **ANY** as its test condition. It branches according to whether any of the elements of the array object **A** is 0.

```
IF (ANY ( A.EQ.0) ) THEN
   PRINT *, 'Array A contains a 0.'
ELSE
   PRINT *, 'Array A does not contain a 0.'
END IF
```

A somewhat more elaborate operation combines a **WHERE** construct on the CM with a front-end control construct, such as **DO**, **DO WHILE**, **DO TIMES**, or **CASE**. The purpose is to perform a parallel operation repeatedly until a local condition is met in each processor. The **WHERE** construct masks out the processors that have met the local condition, and the front-end loop construct checks the result of a reduction function after each iteration to determine whether any processors remain active.

For example, the following program uses the scalar result of **ANY** to control a serial **DO WHILE** loop. In this program, the CM computes the floor of the base-2 logarithm of each element of array **A**. Each CM processor divides its element of **A** by 2 repeatedly until the element becomes 1, accumulating the iteration count in **B**. When the **A** value becomes 1, the processor "drops out," masked out by the **WHERE** construct. Different processors repeat the operation different numbers of times, depending on their initial values for **A**. The serial **DO WHILE** loop continues until no processor is left active.

```
      PROGRAM BASE2_LOG

      INTEGER N
      PARAMETER (N = 256)
      INTEGER, ARRAY (N,N) :: A,B
      IMPLICIT NONE

C     Initialize A with non-zero random numbers

      CALL CMF_RANDOM( A, 1000)       ! A CMF utility routine
      A = A + 1                       ! All A.GT.0

C     Compute B = log_2(A)

      B = 0
      DO WHILE ( ANY( A.GT.1 ))       ! Front-end loop
         WHERE ( A.GT.1 )             ! CM conditional
            A = A/2
            B = B+1
         END WHERE
      END DO

      STOP
      END
```

4.2 Array Sections

Fortran 90 defines syntax for specifying *some* or *all* of an array's elements in a particular reference. This *triplet* notation, mentioned above in Section 1.1.2, resembles a **DO** loop control specification:

lower-bound : *upper-bound* : *stride*

For example, **A(1:4:1)** references every element of an array declared as **A(4)**, while **B(1:3:1, 1:5:1)** references every element of a matrix declared as **B(3,5)**. The lower and upper bounds default to the declared bounds of the array; the stride (increment) defaults to 1. Thus, these array references default to, simply, **A** and **B**.

As you might expect, you can specify bounds other than the array's declared bounds, or a stride other than 1, to indicate a subset of array elements. A subset of elements is called an *array section;* the array from which a section is specified is called the *parent* array.

4.2.1 Triplet Examples

Given a 10-element vector **A**, the expression

```
A(1:5)
```

refers to its first five elements (the stride defaults to 1); whereas the expression

```
A(1:10:2)
```

refers to elements 1, 3, 5, 7, and 9 in that order.

Negative Strides

Negative strides count down from the first value specified to the second. Thus, the expression

```
A(10:2:-2)
```

specifies the elements 10, 8, 6, 4, 2. However,

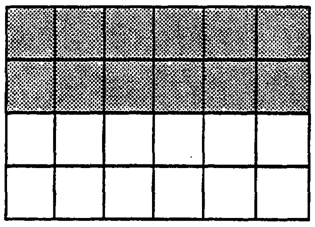
```
A(10:2:2)
```

specifies a null sequence of subscript values (an empty array) because the first value is greater than the second one, but the stride is positive.

Multidimensional Parent Arrays

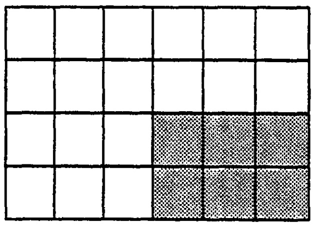
Sections of multidimensional parent arrays are specified with a triplet for each dimension, separated by commas. For example, to specify the upper half of a 4 x 6 matrix B:

```
B(1:2, 1:6)
```



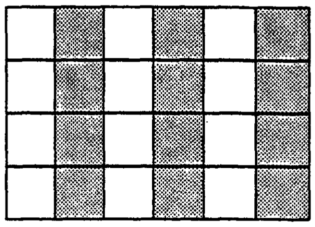
And, to specify the lower right quadrant:

```
B(3:4, 4:6)
```



Finally, to specify only the even columns:

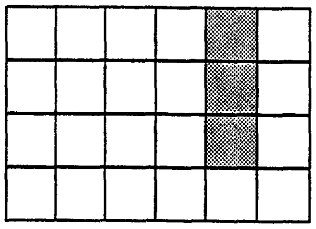
```
B(1:4, 2:6:2)
```



Triplets and Scalar Subscripts

Triplet subscripts can be intermixed in an array reference with Fortran 77–style scalar subscripts. For example, given the 4 x 6 matrix **B**, the following specifies a vector-shaped section that contains the first three elements in column 5:

```
B(1:3, 5)
```



Notice that the rank of this section is not the same as the rank of its parent array. The rank of an array section is the number of dimensions that are specified with a Fortran 90 subscript, not counting any that are specified with Fortran 77 (scalar) subscripts. As another example, the following specifies a 2-dimensional section of a 4-dimensional array **C**:

```
C(1:10, 1, 1:50:2, 3)
```

Default Triplet Forms

Any item of the triplet, or the triplet for any dimension, can be allowed to default, as long as placeholder colons and commas are retained to avoid ambiguity. (The stride in particular is often omitted.) For the sake of clarity, this manual usually specifies both bounds in a triplet, but the following sets of forms are equivalent when referencing sections of an array declared as **D(8,10)**:

```
D(1:4:1, 1:5:1)   <=>   D(1:4, 1:5)    <=>   D(:4, :5)

D(5:8:1, 1:10:1)  <=>   D(5:8, 1:10)   <=>   D(5:8, :)

D(1:3:1, 9)       <=>   D(1:3, 9)      <=>   D(:3, 9)
```

4.2.2 Using Array Sections

An array section can be used as an operand or argument in the same way that whole arrays are used, and the CM system operates on all the specified elements in parallel.

Conformable Sections

When an array section is used in an expression or assignment with other array sections or whole arrays, they must all be of the same shape. Scalars can, of course, be intermixed freely in such expressions.

For example, given two 8 x 10 matrices **D** and **E**, the following statement adds the corresponding elements in the upper left quadrant. Notice that the two sections specified are identical.

```
D(1:4, 1:5) = D(1:4, 1:5) + E(1:4, 1:5)
```

Sections and Communication

Operations on array sections do not require interprocessor communication if the array sections are made up of corresponding elements of conformable parent arrays. For example, the arrays

```
REAL, ARRAY (20,20) :: A,B,C
```

are allocated in the same set of virtual processors. When you select the same elements from all the arrays for an operation, the data items are already in the right processors for the operation to proceed. For example, this is an in-processor operation:

```
A(1:10, 2:20:2) = B(1:10, 2:20:2) + C(1:10, 2:20:2)
```

In this example, the system simply deactivates the processors that contain the array elements that are not included in the sections. The other (active) processors then perform an elemental addition and assignment on values stored in their own memories.

In contrast, operations on array sections that are of the same shape but are not the corresponding elements of their parent arrays do require interprocessor communication. For example:

```
A(1:10, :) = B(1:10, :) + C(11:20, :)
```

The three parent arrays are all in the same set of processors. However, the elements selected from array **C** are in different processors from those taken from **A** and **B**. Before the system can perform the operation, it must allocate temporary storage for the section drawn from **C** in the processors that contain the specified elements of **A** and **B**. Naturally, this (transparent) interprocessor communication adds to the program's execution time.

Array sections that are selected from parents of different shape are not located in the same set of processors because the parents were not allocated in the same set of processors to begin with. The system must always move the specified array elements into the appropriate processors before it can perform an operation. For example, this assignment statement requires interprocessor communication:

```
REAL, ARRAY (5) :: D
REAL, ARRAY (10) :: E

E(5:10) = D
```

Since the two parent arrays are of different shape, they are allocated in different sets of processors. Therefore, the system must (transparently) perform interprocessor communication to line up the operands in the same set of virtual processors before executing the assignment statement.

CM Fortran provides a compiler directive, **LAYOUT**, that you can use to minimize interprocessor communication when assigning arrays and array sections. See Chapter 9.

Chapter 5

Subroutines

To operate on scalars and front-end arrays, a CM Fortran program can define and invoke subprograms—subroutines and functions—in the Fortran 77 fashion. However, some special considerations enter when the subprograms operate on CM array objects (that is, arrays referenced in the Fortran 90 manner).

The special considerations are:

- The ability to pass whole arrays and array sections as actual arguments
- The requirement that the shapes of actual and dummy arguments must match
- The requirement that the homes of actual and dummy arguments must match
- The use of the `INTENT` attribute to save time and memory when passing array sections as arguments
- The ability to allocate local arrays dynamically
- The means of controlling the homes of common arrays

A Note on Functions

Functions can, like subroutines, take array objects as arguments and modify them. Hence, the special considerations for operating on CM array objects apply to both kinds of procedures. In addition, user-defined functions can return array-valued results (see Section 5.6).

CM Fortran also supports Fortran 77 *statement functions,* which define a function in a single statement. A statement function can operate only on scalar data (that is, it cannot use array operations), and it can be called only within the program unit where it is defined.

5.1 Array Objects as Arguments

The general rule of Fortran 90–style array references—that they refer to all the array elements of interest—applies to array arguments just as it does to array-valued expressions and assignments. The syntax by which you reference an array object as an actual argument is sometimes identical to the Fortran 77 syntax; that is, you specify just the array name to pass the whole array. However, the meaning of the reference is slightly different depending on whether the actual argument is a front-end array or a CM array object.

To illustrate the difference, assume that the following array has been declared in a main program to hold some information from a database. The main program can then pass the array as an actual argument to various subroutines that sort, compute, and display the data.

```
INTEGER MY_DATA(100)
. . .
CALL SORT( MY_DATA )
. . .
END

SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(100)
. . .
```

The meaning of the reference to the actual argument **MY_DATA** depends on whether the array has been established in the calling procedure as a front-end array or a CM array:

- *As a front-end array reference,* **MY_DATA** refers to the first element of the array, and the declaration of the dummy argument **ARRAY** indicates how many elements are needed.
- *As a CM array reference,* **MY_DATA** refers to all the elements of the array, and the declaration of the dummy argument must conform to it in size and shape.

The distinction becomes more obvious if we decide to sort only part of the array. If we want to pass, say, the second half of array **MY_DATA** to subroutine **SORT**, the syntax makes it clear that the Fortran 77 argument is a scalar value, while the Fortran 90 argument is an array object:

- If **MY_DATA** is a front-end array, a scalar subscript is used:

```
CALL SORT( MY_DATA(51) )        ! Actual arg is 51st element
. . .
SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(50)               ! Dummy is 50 elements
. . .
```

- If **MY_DATA** is a CM array, a triplet subscript is used:

```
CALL SORT( MY_DATA(51:100) )    ! Actual arg is 50 elements
. . .
SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(50)               ! Dummy conforms to actual
. . .
```

NOTE

This chapter uses literal constants in declaring CM array arguments for the sake of clarity in comparing their shapes. Naturally, a general-purpose procedure is more likely to use variables or named constants in declaring array arguments.

Actual CM array arguments may not be referenced by scalar subscripts in the Fortran 77 fashion. Recall from the discussion of mixed-home array operations in Chapter 2 that a scalar subscript reference to a CM array causes that element to be moved to the front end. Thus, specifying the single CM array element **MY_DATA(51)** passes a single front-end value, not a CM array that begins at that element.

When an array argument is unsubscripted, its semantics *and* its implementation are different from typical Fortran 77 implementations. An unsubscripted reference to a CM array as an argument is a pointer to an *array descriptor*, not a pointer to the first element of the array itself. An array descriptor is a front-end structure that identifies the array and describes its properties, including its shape. In CM Fortran, an array reference with an implicit or explicit triplet subscript points to the descriptor of that array.

5.1.1 The SAVE Attribute

CM Fortran permits the values of an array object to be retained between procedure calls. As in Fortran 77, you specify this behavior by inserting a **SAVE** statement in the specification part of the main program (before any static initialization of the array). Alternatively, you can achieve the same effect with the **SAVE** attribute in the type declaration:

```
    INTEGER, ARRAY(3), SAVE, DATA :: ORDER = [ 1, 2, 3 ]
```

5.2 Homes of Array Arguments

As shown in Chapter 2, the home of an array—front end or CM—is determined by *how the array is referenced* within a program unit: main program, function, or subroutine. Program units are compiled separately from one another, and the compiler carries no information about arrays (or any other object) from one compilation to another.

CM Fortran requires that actual and dummy arguments have the same home. The separate compilation of program units means that an array might come to have different homes in different program units—unless the programmer explicitly prevents this from happening.

5.2.1 Mismatched Homes

As an example of mismatched array homes, imagine that the main database program simply initializes **MY_DATA** with a **READ** statement and then calls subroutines **SORT**, **COMPUTE**, and so on. The compiler will allocate array **MY_DATA** on the front end in the main program. However, if subroutine **SORT** uses its dummy array in a Fortran 90 array operation, the compiler expects that array to be allocated on the CM.

When at run time the front-end array is passed as an argument to **SORT**, *the result will be an error or incorrect results.* The same result occurs when a CM array is passed to a procedure that expects a front-end array. The error of mismatched array homes cannot be detected at compile time in the current version of CM Fortran.

(The error of mismatched homes can, however, be detected at run time if the program is compiled with the switch **-argument_checking**.)

5.2.2 Avoiding Mismatched Homes

To avoid run-time errors, the programmer must ensure that the homes of actual and dummy array arguments match. If an array is referenced in the Fortran 90 manner in any program unit, then all other program units that use that array should cause it to be allocated on the CM.

There are three ways to force an array to have a CM home:

- If appropriate, declare the array in a **COMMON** block. All common arrays are allocated on the CM unless otherwise specified. (See Section 5.7 for more information on the homes of common arrays.)
- Alternatively, use the array in a Fortran 90 array operation in every program unit, even in program units where such an operation is not really needed.
- Alternatively, and somewhat more elegantly, use the compiler directive **LAYOUT** to control the home of the array in each program unit.

  The **LAYOUT** directive, described in Chapter 9, serves primarily to control the mapping of array elements onto CM virtual processors. When the directive applies the keyword **:NEWS** to any dimension of an array, that array is allocated on the CM no matter how it is used in the program unit.

  For example, the following directive line forces **MY_DATA** to be allocated on the CM:

```
      INTEGER MY_DATA(100)
CMF$  LAYOUT MY_DATA (:NEWS)
```

  CMF$ must appear, starting in column 1, to indicate that the line is a compiler directive; column position is otherwise unimportant. Spaces must separate the components of a compiler directive.

5.2.3 Verifying Array Homes

It may be useful during program development and debugging to check where the compiler has allocated particular arrays. The compiler switch **-list** provides this information.

This switch produces the file *program-name*.**lis**, which contains information about a number of program features. Under the "Arrays" section of the listing, the column "Home" specifies either CM or FE for each array in the compiled program. A sample listing is:

```
% cmf -list my-program.fcm
% more my-program.lis
. . .
ARRAYS
  Offset     Size  Type  Block/Class      Home     Name
     ---      10*  L*4   local            CM       ARRAY_1
     ---       40  R*4   local            CM       ARRAY_2
       0       80  C*8   local            FE       ARRAY_3
     120       40  I*4   local            FE       ARRAY_4
```

See the *CM Fortran User's Guide* for explanation of the other entries in this listing.

5.3 Types and Shapes of Array Arguments

Actual array arguments must always be of the same type as their corresponding dummy arguments in a subroutine. As in Fortran 77, mismatched types result in a run-time error or incorrect results.

If the actual and dummy arguments are CM arrays, they must also be of the same shape. This requirement follows from the semantics of a Fortran 90–style array reference to an actual argument: since the actual argument *has* a shape, its shape must conform to that of the dummy. It is an error to reshape a CM array across procedure boundaries.

(The requirement that argument shapes must match is imposed by CM Fortran. Fortran 90 does not require argument shapes to match.)

5.3.1 Declaring Dummy Array Shapes

A dummy CM array can be either *explicit-shape* or *assumed-shape*, depending on how much information its declaration provides about its dimensions.

Explicit-Shape Dummies

An explicit-shape dummy array is one whose dimension sizes (or bounds) are all declared, either with constants or with variables. If a dimension or bound is declared with a variable whose value is not known until run time, the dummy array is called an *adjustable* array.

```
        SUBROUTINE SUB1( A, B, C, D, E, F )

        REAL A(100,100)      ! explicit-shape
        REAL B(-10:5, 0:100)! explicit-shape
        INTEGER C,D
        REAL E(C,D)          ! explicit-shape, adjustable
        REAL F(-10:20, C:D) ! explicit-shape, adjustable
```

An explicit-shape dummy array may be allocated either on the front end or on the CM, depending on how it is used in the procedure (unless its home is constrained as described above in Section 5.2.2).

Assumed-Shape Dummies

An assumed-shape dummy array has an explicit rank, but its dimension extents are unspecified. The dummy array assumes the shape of the actual argument passed. An assumed-shape dummy is a CM array, regardless of how it is used, and the corresponding actual argument must be established in the calling routine as a CM array.

Some examples of assumed-shape dummy arrays are:

```
        SUBROUTINE SUB2( F, G, H, N )

        REAL F(:,:,:)
        REAL G( 0:, 100: )
        INTEGER N
        REAL H( N: )
```

Note the difference in syntax between an assumed-*shape* dummy array and an assumed-*size* dummy array, where the last dimension assumes the size of the actual argument passed:

```
        SUBROUTINE SUB3( CM_ARRAY, FE_ARRAY )

        REAL CM_ARRAY(:,:)       ! assumed-shape; CM array
        REAL FE_ARRAY(100,*)     ! assumed-size; FE array
```

Assumed-size arrays are a feature CM Fortran draws from Fortran 77; only front-end arrays can be assumed-size. Assumed-shape arrays are a feature CM Fortran draws from Fortran 90; only CM arrays can be assumed-shape.

5.3.2 Declaring Local Array Shapes

An array that is local to a subprogram is temporary: it is allocated upon entry to the procedure and deallocated upon exit from the procedure. A local array can be declared with explicit bounds, or it can be made to conform to an adjustable dummy array. A local CM array can also be made to conform to an assumed-shape dummy array.

In cases where a dummy array is non-adjustable, the declaration of a conformable local array is straightforward. As in Fortran 77, the local array is declared with the same constant size (or bounds) as the dummy array. For example:

```
SUBROUTINE SUB4( A )

REAL A(100,100)
REAL TEMP(100,100)
```

When a local CM array needs to conform to an adjustable dummy array, the local is declared with the same variables that specify the shape of the dummy. Such a local array is called an *automatic* array; space for it is allocated at run time on the CM. The array TEMP in the following fragment is an automatic array:

```
SUBROUTINE SUB5( B,I,J )

INTEGER I,J
REAL B(I,J)            ! Dummy B is adjustable
REAL TEMP(I,J)         ! Automatic TEMP conforms to B
TEMP = B               ! TEMP and B are CM arrays
```

You can also declare an automatic array to be conformable with an assumed-shape dummy array. Doing so requires finding, at run time, the dimensions of the corresponding dummy array. The intrinsic functions that return this information are shown in Section 5.5.

Automatic arrays need not be associated with dummy arguments, of course. Array TEMP in the following fragment is also a CM automatic array. It is allocated at run time, with its shape dependent on the values of the actual arguments I and J.

```
SUBROUTINE SUB4( I,J )

INTEGER I,J
REAL TEMP(I,J)         ! TEMP is automatic
TEMP = TEMP + 1        ! TEMP is a CM array
```

Automatic arrays are useful, for example, in writing a program that scales at run time to the size of a data set. The following program reads from a file the information needed to determine array size. It then calls a subprogram, which uses the size data to specify the dimensions of two automatic arrays **A** and **B**.

```
PROGRAM ANY_SIZE
INTEGER N,M

READ (*,*) N,M              ! Get size
CALL DUMMYMAIN( N,M )

END

SUBROUTINE DUMMYMAIN( N,M )
INTEGER N,M
REAL A(N,M), B(N,M)         ! automatic arrays

. . . code that uses A and B in array operations . . .

RETURN
END
```

5.3.3 Making Subroutine Interfaces Explicit

The separate compilation of program units means that the compiler cannot ordinarily compare the types and shapes of actual and dummy arguments across subroutine boundaries. However, the compiler can check for type and shape mismatches if the program makes the subroutine interface explicit to the calling routine.

The interface can be made explicit by means of an *interface block*, which duplicates the subroutine's interface within the calling routine. For example, recall the interface to subroutine **SORT**:

```
SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(100)
```

To make that interface explicit to the main program that calls subroutine **SORT**, insert the following interface block into the specification part of the main program:

```
      INTERFACE
         SUBROUTINE SORT( ARRAY )
         INTEGER ARRAY(100)
      END INTERFACE
```

Interface blocks are not required in CM Fortran, but they are advisable when calling any procedure that has dummy CM array arguments. The explicit interface helps in catching type errors and shape errors at compile time.

Notice, however, that the interface block shown contains no information on the home of the argument array. Thus, the compiler cannot detect an error of mismatched array homes. This information is available to the compiler *only* if the array's home is determined by a compiler directive, such as **LAYOUT**. In that case, the directive line must appear in the interface block, as well as in the declaration of the dummy argument in the subroutine and in the declaration of the actual argument in the calling routine.

```
      INTERFACE
         SUBROUTINE SORT( ARRAY )
         INTEGER SORT(ARRAY)
CMF$     LAYOUT ARRAY (:NEWS)
      END INTERFACE
```

5.4 Array Sections as Arguments

The example at the beginning of this chapter showed a section of array **MY_DATA** passed as an argument to subroutine **SORT**. To sort only the second half of the data:

```
      INTEGER MY_DATA(100)
      . . .
      CALL SORT( MY_DATA(51:100) )   ! Actual arg is 50 elements
      . . .
      END

      SUBROUTINE SORT( ARRAY )
      INTEGER ARRAY(50)              ! Dummy conforms to actual
      . . .
```

Similarly, to select and sort every fourth data element, pass a section with a stride of 4:

```
CALL SORT( MY_DATA(1:100:4) )       ! Actual is 25 elements
. . .
SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(25)                ! Dummy conforms to actual
. . .
```

Array sections can be used anywhere that whole CM arrays are used, with no difference in the semantics of the array reference. (Recall from Chapter 4 that a reference to a whole array is simply the default form of a triplet subscript.) Thus, actual arguments that are array sections must match their corresponding dummy arguments in shape as well as in type, and interface blocks are recommended to facilitate error checking.

5.4.1 Passing Array Sections

One way in which array sections *do* differ from whole arrays is in the way the system passes them as arguments. Whole arrays are passed in place, whereas array sections are first copied to a temporary location. If the subroutine alters the array section, then it is also copied back into the original (parent) array upon exit from the subroutine.

(Some sections taken from an array that is subject to a **LAYOUT** directive are passed in place. See Section 9.4 for a discussion of this special case.)

One way to avoid any needless copying is to avoid passing array sections as arguments when possible. If the algorithm permits, you can pass the whole array and have the subroutine select the desired section. For example:

```
INTEGER MY_DATA(100)

INTERFACE                        ! Optional interface block
   SUBROUTINE SORT( ARRAY )
   INTEGER ARRAY(100)
END INTERFACE
. . .
CALL SORT( MY_DATA )             ! Actual arg is 100 elements
. . .
END

SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(100)               ! Dummy conforms to actual
ARRAY(1:100:4) = [some sorting algorithm] ! Operate on 25 elements
END
```

5.4.2 The INTENT Attribute

When array sections must be passed as arguments, the **INTENT** attribute can increase program performance by preventing needless copying. The **INTENT** attribute, applied to any dummy argument, indicates whether that argument will be used to receive data from, or return data to, the calling procedure, or both.

One way to apply the attribute to a dummy array in the **INTENT** statement:

```
SUBROUTINE SUB4( A,B,C )
REAL A(100,100), B(100,100), C(100,100)

INTENT(OUT) :: A     ! A values will only be written out
INTENT(IN)  :: B     ! B values will only be read in
INTENT(INOUT) :: C   ! C values will be both read and written
```

If the **INTENT** statement indicates that the dummy argument need not receive data from the calling procedure, then the system will not copy in the values of the actual argument. If the **INTENT** statement indicates that the dummy need not return data to the calling procedure, then the system will not copy values back into the original array.

Alternatively, the **INTENT** attribute can be applied to a dummy array in a Fortran 90 attributed type declaration:

```
SUBROUTINE SUB5( D,E,F )
REAL, ARRAY(100,100), INTENT(IN)  :: D,E
REAL, ARRAY(100,100), INTENT(OUT) :: F
```

Since the calling procedure needs this information about the intended use of the dummy argument, you must provide an interface block when specifying an **INTENT** attribute, and the interface block must contain the **INTENT** statement or attribute declaration. Without an interface block, the **INTENT** statement or attribute has no effect.

5.4.3 Example of INTENT

Imagine that a main game-playing program does not initialize an array, but instead simply passes half the array to subroutine **SORT** to be initialized. If the dummy array in the subroutine has the attribute **INTENT(OUT)**, the system will not bother to copy in the values from the original array, but it will copy the sorted values back out to the original array:

```
INTEGER TOKENS(100)

INTERFACE                               ! Interface block required
   SUBROUTINE SORT( ARRAY )
   INTEGER ARRAY(50)
   INTENT(OUT) :: ARRAY
END INTERFACE
. . .
CALL SORT( TOKENS(1:50) )
. . .
END

SUBROUTINE SORT( ARRAY )
INTEGER ARRAY(50)
INTENT(OUT) :: ARRAY
. . .
```

Subroutine **PLAY**, on the other hand, needs to receive the values of the sorted tokens *and* pass the values back to the main program so that they can be passed on to subroutine **DISPLAY**. Thus, **PLAY** can give the dummy argument the attribute **INTENT(INOUT)**, although the effect on copying behavior is the same as specifying no **INTENT** attribute.

Finally, subroutine **DISPLAY** needs to receive the values as computed by **PLAY**, but it need not store the final state of the tokens back into the original array. This subroutine therefore gives the array the attribute **INTENT(IN)**:

```
INTEGER TOKENS(100)

INTERFACE                               ! Interface block required
   SUBROUTINE DISPLAY( ARRAY)
   INTEGER ARRAY(50)
   INTENT(IN) :: ARRAY
END INTERFACE
. . .
CALL DISPLAY( TOKENS(1:50) )
. . .
END

SUBROUTINE DISPLAY( ARRAY )
INTEGER ARRAY(50)
INTENT(IN) :: ARRAY
. . .
```

5.5 Retrieving Array Properties

CM Fortran provides a number of intrinsic functions that inquire about the size, shape, or bounds of an array or one of its dimensions. These functions are **DSIZE**, **DSHAPE**, **RANK**, **DUBOUND**, and **DLBOUND**.

For convenience, the following examples illustrate these functions in relation to arrays whose shapes are declared—a rather pointless exercise. These functions are particularly useful in relation to adjustable dummy array arguments, assumed-shape arrays, and automatic arrays—that is, arrays whose size and shape are not established until run time.

Unlike other CM Fortran intrinsic functions, these inquiry functions can operate on front-end arrays as well as CM arrays.

5.5.1 Array Shape

The function **DSHAPE** takes an array (or section) and returns a vector whose elements are the lengths of the array's dimensions. The result is returned on the front end (regardless of the home of the argument array). For example:

```
REAL, ARRAY (5,10) :: A

PRINT *, DSHAPE( A )                ! Prints [ 5,10 ]
PRINT *, DSHAPE( A(3,:) )           ! Prints [ 10 ]
```

DSHAPE is useful when retrieving the properties of adjustable and assumed-shape arrays; it cannot be used with assumed-*size* arrays. Alone among the inquiry functions, **DSHAPE** can also take a scalar argument. In this case, the value returned is an array of size zero (not the scalar zero).

5.5.2 Array Size

The function **DSIZE** takes an array and an optional dimension argument. It returns a scalar that is the number of elements in the whole array (or section) or in the specified dimension. For example:

```
REAL, ARRAY (5,10) :: A

PRINT *, DSIZE( A )                     ! Prints 50
PRINT *, DSIZE( A(3,:) )                ! Prints 10
PRINT *, DSIZE( A, DIM=1 )              ! Prints 5
```

If the argument is an assumed-size array, **DSIZE** cannot retrieve the size of the last dimension—or, therefore, the size of the whole array. For assumed-size arrays, the dimension argument must be supplied and it must not be the last dimension. For example:

```
REAL, ARRAY (5,*) :: B

PRINT *, DSIZE( B, DIM=1 )              ! Prints 5
PRINT *, DSIZE( B, DIM=2 )              ! Not supported
PRINT *, DSIZE( B )                     ! Not supported
```

5.5.3 Dimension Bounds

The functions **DUBOUND** and **DLBOUND** take an array and return, as a front-end vector, all its upper or lower bounds. For example:

```
REAL, ARRAY (5,10) :: A
REAL, ARRAY (-10:100, 20:50) :: C

PRINT *, DUBOUND( A )                   ! Prints [ 5,10 ]
PRINT *, DLBOUND( A )                   ! Prints [ 1,1 ]
PRINT *, DLBOUND( C )                   ! Prints [ -10,20 ]
PRINT *, DUBOUND( C(-10:0,:) )          ! Prints [ 0,50 ]
```

If the dimension argument is supplied, these functions return a scalar that is the requested bound of that dimension:

```
REAL, ARRAY (-10:100, 20:50) :: C

PRINT *, DUBOUND( C, DIM=1 )            ! Prints 100
PRINT *, DLBOUND( C, DIM=2 )            ! Prints 20
PRINT *, DLBOUND( C(40:,:40), DIM=1)! Prints 40
```

If the argument is an assumed-size array, **DUBOUND** cannot retrieve the upper bound of the last dimension—or, therefore, those of the whole array. For assumed-size arrays, the dimension argument must be supplied and it must not be the last dimension. For example:

```
REAL, ARRAY (5,*) :: B

PRINT *, DUBOUND( B, DIM=1 )        ! Prints 5
PRINT *, DUBOUND( B, DIM=2 )        ! Not supported
PRINT *, DUBOUND( B )               ! Not supported
```

5.5.4 Inquiry Example

All the inquiry intrinsic functions can be used in specification statements as well as in executable statements.The function **DUBOUND** is particularly useful for finding the upper bounds of an assumed-shape dummy array. This information can be used to declare an automatic (temporary) array that conforms to the dummy array:

```
SUBROUTINE SUB7( A )
REAL A(:,:)
REAL TEMP( DUBOUND( A,1 ), DUBOUND( A,2 ))

TEMP=A         ! Automatic TEMP and dummy A are CM arrays
```

The two arrays in this subroutine would be taken by the compiler to be CM arrays even if they weren't used in an array assignment. Array **A** is a CM array because it is assumed-shape; array **TEMP** is a CM array because it is automatic. The actual argument passed in at run time as the **A** argument must be established as a CM array in the calling routine.

5.6 Array-Valued Functions

All the considerations that this chapter describes for defining and invoking subroutines apply also to user-defined functions (with certain temporary restrictions noted here).

When a function takes CM array arguments, its returned value can be defined as either a scalar or an array. An example of a function that returns a scalar is:

```
INTEGER FUNCTION AVERAGE( ARRAY )
INTEGER ARRAY (:)

AVERAGE = SUM( ARRAY ) / DSIZE( ARRAY )
END
```

The argument passed to this function must be a CM array. The dummy **ARRAY** is assumed-shape and is passed as an argument to the intrinsic reduction function **SUM**. Either of these factors would be sufficient to cause the dummy to be a CM array. The functions **SUM** and **DSIZE** return their scalar results to the front end, and the division operation executes on the front end.

User-defined functions that return arrays are somewhat restricted in Version 1.0, compared with subroutines and with functions that return scalars. The shape of the returned array must be known at compile time (that is, it must be constant), and the function body must use the array in a Fortran 90 construction to force it onto the CM. In addition, the program unit that invokes the function must include an interface block for the function (as described in Section 5.3.3).

Note that the behavior of an array-valued function is like that of a subroutine that takes an array as its first argument and stores its results there. To avoid the current restrictions on array-valued functions, you could write the procedures as subroutines instead.

5.7 Common Arrays

CM Fortran allows any array to be placed in a common block, either on the CM or on the front-end computer, and then be shared by program units. However, the homes of common arrays are determined differently from those of local arrays, and the homes constrain the ways in which the arrays can be used. It is not possible to allocate a given common block on both machines.

5.7.1 Common Array Homes

The homes of common arrays are determined in the specification part of the program—*not* by how the arrays are used. Programmers must take care that the use made of a particular common array in every program unit is appropriate to its home:

- A common array that is allocated on the front-end computer cannot be used in an array operation.
- A common array that is allocated on the CM can be referenced in the Fortran 77 manner (with scalar subscripts rather than triplet subscripts), but the operation is likely to be very slow as the system copies the array values one at a time between machines.

  The utility routines **CMF_FE_ARRAY_FROM_CM** and **CMF_FE_ARRAY_TO_CM** can perform a faster bulk data transfer for those cases where it is necessary to use a CM common array in a serial operation. (See the *CM Fortran User's Guide* for information on utility routines.)
- A common array that is allocated on the CM cannot be used by a program unit or library procedure that was compiled by a foreign compiler.

All common arrays retain their values when control passes from one program unit to the next.

5.7.2 Declaring Common Arrays

Like local arrays, common arrays can be declared either with Fortran 77 syntax or with Fortran 90 attributed type declarations (as described in Chapter 3), with no difference in the semantics of the specification. CM common arrays must be declared in the main program, as well as in any subprogram that uses them.

Any array (or scalar) can be placed in a block of common storage by means of the **COMMON** statement:

> **COMMON** [*/block-name/*] *var-list*

Unlike local arrays, common arrays are allocated by default on the CM. The home then determines the permissible uses of the common array, rather than vice versa. This section describes the various methods of overriding the default CM allocation of common arrays.

Character Type and LAYOUT Directive

A common array is allocated on the front end if it is of type character or if it is constrained to a front-end home by the compiler directive **LAYOUT** or **ALIGN**. The **LAYOUT** directive was introduced in Section 5.2.2 above and is described in Chapter 9, along with **ALIGN**. All the following arrays are allocated on the front end and cannot be used in array operations:

```
      CHARACTER A                    ! Character type is FE only
      REAL, ARRAY(100,100) :: B,C

CMF$  LAYOUT B(:SERIAL, :SERIAL)     ! Layout places B on FE
CMF$  ALIGN C(I,J) WITH B(I,J)       ! C has same home as B

      COMMON /BLOCK_1/ A(10),B,C     ! 3 arrays in FE common block
```

COMMON Directive

The compiler directive **COMMON** can specify the home of a common block by designating the block as **FEONLY** or **CMONLY**. Both the following arrays are allocated on the front end and cannot be used in array operations:

```
      REAL, ARRAY(100,100) :: D,E
      COMMON /BLOCK_2/ D,E           ! Would default to CM, but...

CMF$  COMMON FEONLY /block_2/        ! COMMON overrides default home
```

The **COMMON** directive has no effect on an array's home if the home is already constrained by **LAYOUT** or **ALIGN** or if the array is of type character.

Compiler Switches

The CM Fortran compiler switch **-fecommon** places common blocks on the front end, overriding the default placement of common blocks on the CM. This switch does *not* affect arrays that are constrained to a CM home by the directives **LAYOUT**, **ALIGN**, or **COMMON**.

Another compiler switch, **-nodirectives**, disables all the compiler directives in the program. Using this switch causes all arrays and common blocks to revert to their default homes: character arrays (as always) to the front end, all non-character common arrays to the CM, and all non-character local arrays to one machine or the other depending on how they are used. This switch should be used with care, since, in possibly changing array homes, it can cause valid programs to fail.

5.7.3 Initializing Common Arrays

The initialization techniques described in Chapter 3 for local arrays apply also to common arrays: any array, front-end or CM, local or common, can be initialized by means of a DATA statement or a DATA attribute associated with an array constructor in the type declaration.

However, some special actions are needed to initialize a CM common array "statically," that is, with a DATA statement or attribute. In fact, there is no static initialization of CM arrays, since CM space is not allocated until run time. When the system encounters a DATA statement or attribute for a CM array, it places the values in front-end storage and moves them to the CM at program start-up.

For local arrays, this action happens transparently; the programmer need only ensure that there is adequate front-end space available for storing the CM values. For common arrays, however, the programmer must do the following to ensure that the necessary storage is allocated on the front end:

- Name all common blocks that contain statically initialized CM arrays. This step is a prerequisite to the second step.
- Initialize CM common arrays in a BLOCK DATA program unit. As with all common arrays, the DATA statement or attribute that initializes a CM common array cannot appear in a main program or external procedure.
- Indicate to the compiler that CM common arrays will be statically initialized. This can be done either with the compiler switch **`-common_initialized`** or with another form of the compiler directive COMMON:

  ```
  CMF$  COMMON INITIALIZED /block-name/
  ```

  This form of the COMMON directive indicates that the common block is to reside on the CM (the default home in any case) and that front-end storage is to be allocated for its initial values.

  The COMMON directive indicating that a CM common array is initialized must appear in the BLOCK DATA program unit and in all program units where the array is used. See the following page for an example.

```
      PROGRAM CM_DATA
      REAL A(1024)
      COMMON /CMBLK/ A
CMF$  COMMON INITIALIZED /CMBLK/

      PRINT *, A(1)

      STOP
      END

      BLOCK DATA
      REAL A(1024)
CMF$  COMMON INITIALIZED /CMBLK/
      COMMON /CMBLK/ A
      DATA A/1024*0.2/
      END
```

Chapter 6

Data Movement

CM Fortran, like Fortran 90, provides new syntax and intrinsic functions for rearranging the elements of arrays. Dimensional shifts, permutations, and transpositions of array elements all require interprocessor communication on the Connection Machine system.

The features that enable the program to specify interprocessor data movement explicitly are:

- Array sections
- Vector-valued subscripts
- The intrinsic functions **CSHIFT**, **EOSHIFT**, and **TRANSPOSE**

Other intrinsic functions perform data movement implicitly in the course of some other kind of array transformation. These functions are presented in Chapter 7.

6.1 Array Sections

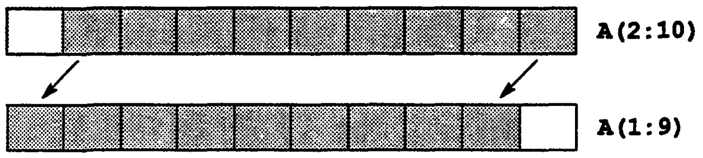
The discussion of array sections (Section 4.2.2) mentioned some cases where implicit data movement occurs in the course of an operation, such as expressions involving array sections from nonconformable parent arrays. This section focuses on using array sections deliberately to reposition the values of arrays.

6.1.1 Dimensional Shifts

Array sections, in conjunction with assignments, are useful for moving data in regular grid patterns. An example is a "dimensional shift," where element values are all shifted some number of positions along an array dimension. The following assignment statement shifts the elements of a 10-element vector **A** by one position:

```
A(1:9) = A(2:10)
```

as illustrated by:

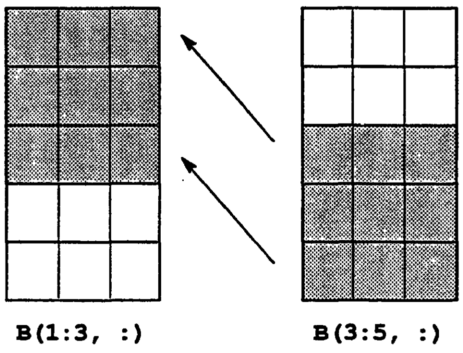


Notice that—as with any array assignment—the two array sections must be conformable. Thus, a number of elements at least equal to the shift offset must be excluded from the computation. In this example, a section that excludes the first element of the parent array is assigned to a section that excludes the last element.

Extending this example to a 2-dimensional array is straightforward. Suppose that **B** is a 5 x 3 matrix and you want to shift by two positions on the first dimension:

```
B(1:3, :) = B(3:5, :)
```

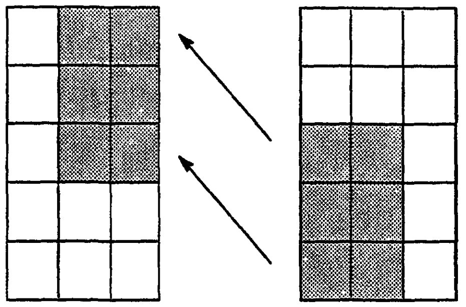
as illustrated by:



Similarly, to shift along more than one dimension at a time:

```
B(1:3, 2:3) = B(3:5, 1:2)
```

as illustrated by:



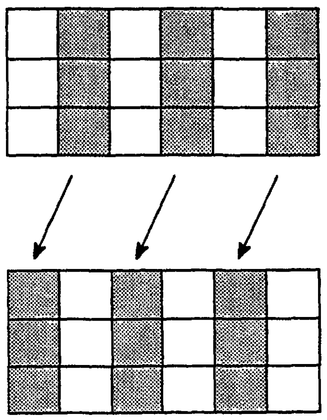
B(1:3, 2:3) B(3:5, 1:2)

6.1.2 Shifting Noncontiguous Sections

Specifying a stride greater than 1 yields a noncontiguous array section, but the mechanics of shifting element values along a dimension are the same. For example, this statement copies the values from the even columns of a 3 x 6 array **C** into the elements in the odd columns in each row.

```
C(:, 1:5:2) = C (:, 2:6:2)
```

as illustrated by:



The before-and-after values in **C** might be:

$$C = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 3 & 1 & 3 & 1 & 3 \end{bmatrix} \Rightarrow C = \begin{bmatrix} 2 & 2 & 4 & 4 & 6 & 6 \\ 2 & 2 & 4 & 4 & 6 & 6 \\ 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

6.1.3 Efficiency Note

Some dimensional shifts use the CM's NEWS communication network, which optimizes the special case of grid communication for speed. Generally speaking, the compiler generates NEWS instructions if the parent arrays of the sections are conformable (or are the same array, as in the examples shown) *and* if shift offset can be broken into a few power-of-2 distances. Otherwise, the compiler uses the more general (and therefore slower) router communication network.

6.1.4 Permutations

Array sections can be used for data motion other than dimensional shifts by mixing positive and negative strides or by mixing unequal strides in the assignment statement. For example, to reverse the elements of a 10-element vector **D**:

```
D = D(10:1:-1)
```

Similarly, to reverse the columns of values in an *n* x 6 matrix **E**:

```
E  =  E(:, 6:1:-1)
```

And, to replace the left half of **E** with the values in the even columns:

```
E(:, 1:3)  =  E(:, 2:6:2)
```

The CM system cannot use NEWS communication in cases where the data motion specified is not a dimensional shift. In the examples shown here, the values move different distances to reach their destination positions (processors). Any data motion where the pattern of movement is not the same for all selected elements must use the router communication network.

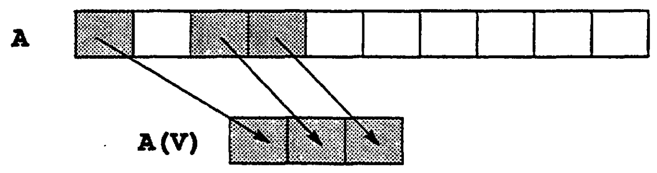
6.2 Vector-Valued Subscripts

A special form of array section uses a *vector-valued subscript,* an integer vector that serves as a sequence of index values in an array reference. Since the index vector need not be ordered—that is, there is no fixed stride—this syntax permits an arbitrary selection of array values along a dimension. A vector-valued subscript is essentially an indirection vector; it is particularly useful for mapping unstructured problems onto a rectangular grid.

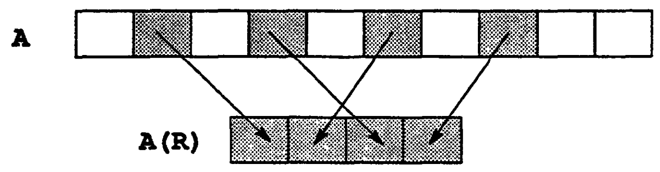
Like other array sections, array sections specified with vector-valued subscripts can be used in conjunction with assignment statements to perform data movement. This facility always uses the router communication network. It is thus one of the more general but not the fastest of CM Fortran features.

6.2.1 Examples

If **A** is a vector of length 10 and **V** is the vector [1,3,4], then **A(V)** is a vector of length 3 whose elements are **A(1)**, **A(3)**, and **A(4)**.



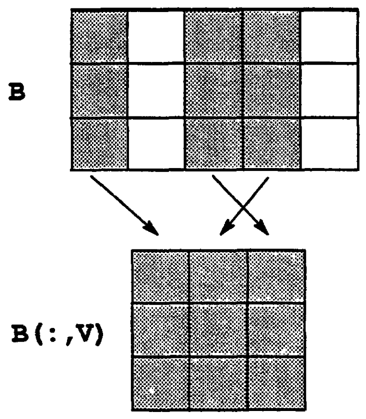
Similarly, if **R** is the vector [2,6,4,8], then **A(R)** is a vector of length 4 whose elements are **A(2)**, **A(6)**, **A(4)**, and **A(8)**.



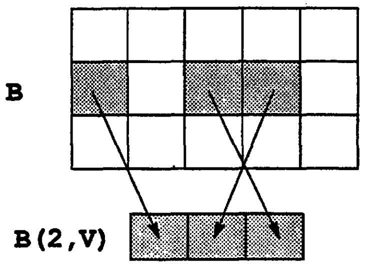
Array sections specified with vector-valued subscripts always involve data movement, even in the absence of a statement assigning the values to different positions. That is, the system creates a temporary array of the same length as the index vector and moves the specified values of the parent array into the temporary before performing any operation on them. (Because the indices are not necessarily in their natural order, the system cannot simply deactivate the processors whose indices are excluded, as it does with array sections.) As a result, even a statement like the following requires the use of router communication:

```
        A(V) = A(V) + 1
```

Vector-valued subscripts can be intermixed in a reference with triplet subscripts and scalar subscripts. For example, given a 3 x 5 array **B** and the vector **V = [1,4,3]**, then **B(:, V)** specifies a 2-dimensional array section that consists of the first, fourth, and third columns of **B**, in that order:



The section **B(2,V)** specifies a 1-dimensional section of length 3 consisting of the first, fourth, and third values in the second row of **B**.



Notice that the rank of an array section equals the number of dimensions that are specified with either triplet subscripts or vector-valued subscripts, not counting any dimensions that are specified with scalar subscripts. Thus, **B(:,V)** in this example is smaller than the parent array **B** but still 2-dimensional. The section **B(2,V)**, however, has only one Fortran 90–style subscript and is therefore 1-dimensional.

6.2.2 Permutations

One use of vector-valued subscripts is to rearrange array elements. For example, if **A** and **B** are 10-element vectors, and vector **P** is a permutation of the numbers 1 to 10, the statement

```
A = B(P)
```

rearranges the values of **B** according to the permutation specified by **P** and assigns them to **A**. These actions are equivalent to the following Fortran 77 operation:

```
   DO 30 I = 1,10
      A(I) = B( P(I) )
30 CONTINUE
```

6.2.3 Assigning Permuted Sections

Like any array object, a section specified with a vector-valued subscript must be conformable with other array objects used in the same expression or in an assignment.

Thus, when the index vector is on the right-hand side of an assignment statement, it must be conformable with the destination array on the left. The source array can be any size or shape, since the index vector is specifying a section of the source array that is appropriately shaped for the assignment.

For example, assume the following vectors:

```
SOURCE = [ 10, 20, 30, 40, 50, 60, 70 ]
INDEX  = [ 3, 1, 4, 6 ]
```

Executing the statement

```
DEST = SOURCE(INDEX)
```

assigns the following values to **DEST**:

```
DEST   = [ 30, 10, 40, 60 ]
```

Similarly, when the index vector is on the left, as in the statement

```
DEST_2(INDEX_2) = SOURCE_2
```

the index vector must be conformable with the source array. The destination array can be any size or shape, since the index vector is specifying a section of it that is appropriate for the assignment.

6.2.4 Replicating Data

Vector-valued subscripts can go beyond simply rearranging data, since an index number may appear more than once in the index vector (as long as the vector appears on the right-hand side of the assignment). For example, assume the following vectors:

```
SOURCE_3 = [ 15, 20, 25, 30, 35, 40, 45 ]
INDEX_3  = [ 3, 1, 4, 4 ]
```

Notice that index 4 is specified twice in the index vector. Executing the statement

```
DEST_3 = SOURCE_3(INDEX_3)
```

assigns the following values to **DEST_3**:

```
[ 25, 15, 30, 30 ]
```

The index numbers may not be replicated when the index vector appears on the left-hand side of an assignment, since the language does not define the effect of assigning more than one value to an array element.

6.3 Intrinsic Shift Functions

CM Fortran provides two intrinsic functions for shifting array elements in regular patterns along array dimensions. Their actions differ from the dimensional shifts performed with array sections in that the functions deal with the boundary elements.

- **CSHIFT**, for "circular shift," causes values that move off the edge of the array to reappear at the opposite edge.

```
CSHIFT( ARRAY, DIM, SHIFT )
```

- **EOSHIFT**, for "end-off shift," discards values that move off the edge of the array and moves some specified or default value into the positions vacated at the opposite edge.

```
EOSHIFT( ARRAY, DIM, SHIFT [, BOUNDARY] )
```

6.3.1 Using CSHIFT

Given a 3 x 5 array **A**, the statement

```
A = CSHIFT( A, DIM=2, SHIFT=-1 )
```

shifts the values in the second dimension of **A** (which you can picture as the columns, or as the values *in* the rows) by one column position in the negative direction, wrapping the right-most values around to the first column.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \quad \Rightarrow \quad A = \begin{bmatrix} 5 & 1 & 2 & 3 & 4 \\ 5 & 1 & 2 & 3 & 4 \\ 5 & 1 & 2 & 3 & 4 \end{bmatrix}$$

The negative direction—meaning that element **A(I,J)** gets the value of **A(I,J-1)**—is described here as "shifting to the right." An alternative view of the same action is that each element gets the value of its neighbor to the left, with wrapping at the edge.

(Notice in the above example that CM Fortran supports the use of predefined keywords in intrinsic function calls.)

Calls to **CSHIFT** can be nested to access diagonal neighbors. For example, to have each element of a matrix get the sum of its four diagonal neighbors:

```
      A = CSHIFT( CSHIFT( A,1,1 ),  2,  1)     +
     $    CSHIFT( CSHIFT( A,1,1 ),  2, -1)     +
     $    CSHIFT( CSHIFT( A,1,-1 ), 2,  1)     +
     $    CSHIFT( CSHIFT( A,1,-1 ), 2, -1)
```

The **SHIFT** argument to **CSHIFT** can also be an array, indicating a possibly different shift distance for each row or column of the argument array. The **SHIFT** array must be of rank one less than the argument array. For 2-dimensional argument arrays, the **SHIFT** argument can be any vector, including an array constructor (see Chapter 3). For example, if **B** is a 2-dimensional array, then

```
      B = CSHIFT( B, DIM=2, SHIFT=[1,2,3] )
```

has the following effect on **B**:

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} \Rightarrow B = \begin{bmatrix} 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix} \begin{array}{l} \text{(shift by 1)} \\ \text{(shift by 2)} \\ \text{(shift by 3)} \end{array}$$

The **DIM** argument to **CSHIFT** can be either a constant expression or a variable—as long as **SHIFT** is a scalar. If **SHIFT** is an array, however, **DIM** must be a constant expression.

For example, all the following combinations of constant and variable **DIM** arguments and scalar and array **SHIFT** arguments are supported:

```
      A = CSHIFT( A, DIM=2, SHIFT=1 )
      A = CSHIFT( A, DIM=J, SHIFT=1 )
      A = CSHIFT( A, DIM=2, SHIFT=[1,2,3] )
```

The illegal combination is variable **DIM** with array **SHIFT**:

```
      A = CSHIFT( A, DIM=J, SHIFT=[1,2,3] )   ! Not supported
```

6.3.2 Using EOSHIFT

EOSHIFT is similar to **CSHIFT** except for its treatment of boundary elements. This function takes an optional **BOUNDARY** argument that specifies the value to move into the elements that are vacated by the shift operation. The default is zero or **.FALSE.**, depending on the type of array.

6.4 The TRANSPOSE Function

The intrinsic function **TRANSPOSE** transposes the axes of a matrix, creating a matrix of shape (**M,N**) from an argument matrix of shape (**N,M**). For example,

```
MATRIX_2 = TRANSPOSE( MATRIX_1 )
```

has the following effect when **MATRIX_1** is 4 x 3:

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{bmatrix}$$

If the argument matrix is a square, the effect is to flip its values over the diagonal. For example, calling **TRANSPOSE** on a 4 x 4 matrix has the following effect:

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 2 & 2 & 0 & 1 \\ 2 & 2 & 2 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 2 & 2 & 2 \\ 1 & 0 & 2 & 2 \\ 1 & 1 & 0 & 2 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Notice that the **TRANSPOSE** function moves the various element values different distances and directions. To perform this pattern of data movement the system necessarily uses the router communication network. The **TRANSPOSE** function therefore does not execute as fast as the dimensional shifts shown above; its speed is comparable to that of permutations.

Chapter 7

Array Transformations

CM Fortran defines a rich set of intrinsic functions for manipulating and transforming arrays and inquiring about their properties. Three categories of intrinsic functions have already been introduced:

- *Elemental* functions (Chapter 2), which apply Fortran 77 numeric and mathematical operations to all the elements of array arguments (as well as to scalars)
- *Inquiry* functions (Chapter 5), which return information about an array (CM or front-end), such as its size or shape
- *Movement* functions (Chapter 6), which reposition the elements of a CM array by performing a dimensional shift or a matrix transposition

The other categories of array-related intrinsic functions are introduced in this chapter. These are:

- *Array reduction* functions, which take an array and summarize it by applying some combining operation over its values
- *Element location* functions, which determine the location (subscripts) of particular elements such as the maximum or minimum value
- *Array construction* functions, which construct a new array from information contained in one or more argument arrays
- *Array multiplication* functions, which include `DOTPRODUCT` and `MATMUL`

All the CM Fortran movement, reduction, location, construction, and array multiplication intrinsic functions operate only on array objects (CM arrays or array sections referenced in the Fortran 90 manner).

7.1 Array Reduction

The array reduction functions take an array and "summarize" it by applying some combining operation across its elements. Each of the reduction functions may be applied either to an array object (whole array or section) or to a single dimension of an array object. The result may be a scalar or an array, depending on the arguments.

The numeric reduction functions are **MAXVAL**, **MINVAL**, **SUM**, and **PRODUCT**. The logical reduction functions are **ANY**, **ALL**, and **COUNT**.

7.1.1 The ARRAY Argument

As an example of the numeric reduction functions, consider the action of **MAXVAL** on the 2-dimensional array **A**:

```
MAXVAL( ARRAY [, DIM] [, MASK] )
```

$$A = \begin{bmatrix} 1 & 5 & 3 & 7 \\ 4 & 2 & 6 & 3 \\ 9 & 2 & 1 & 5 \end{bmatrix}$$

When the function is applied to the whole array or to a section, the result is a scalar and is returned to the front end:

```
I = MAXVAL( A )                   ! I = 9

J = MAXVAL( A(:,2:3) )            ! J = 6
```

If the optional dimension argument is supplied, the result is an array with one fewer dimension than the argument array. The array result of a reduction function is a CM array.

```
K = MAXVAL( A, DIM=1 )            ! K = [ 9,5,6,7 ]

L = MAXVAL( A(:,2:3), DIM=2 )     ! L = [ 5,6,2 ]
```

7.1.2 The MASK Argument

The numeric reduction functions (**MAXVAL**, **MINVAL**, **SUM**, **PRODUCT**) take an optional **MASK** argument indicating which elements to include in computing the summary result. The mask must be a logical array or array-valued expression of the same shape as the argument array. For example, to find the highest value in **A** that is not greater than 8:

```
I = MAXVAL( A, MASK = A .LE. 8 )        ! I = 7
```

Similarly, if array **B** is specified as a mask for **A** (**T** indicates .**TRUE**. and a dot indicates .**FALSE**.):

$$A = \begin{bmatrix} 1 & 5 & 3 & 7 \\ 4 & 2 & 6 & 3 \\ 9 & 2 & 1 & 5 \end{bmatrix} \qquad B = \begin{bmatrix} \cdot & T & \cdot & T \\ T & T & \cdot & T \\ \cdot & T & T & \cdot \end{bmatrix}$$

Then,

```
J = MAXVAL( A, MASK=B )                 ! J = 7

K = MAXVAL( A, DIM=1, MASK=B )          ! K = [ 4,5,1,7 ]
```

Notice that if the **ARRAY** argument is an array section, then the mask must conform to the section:

```
L = MAXVAL( A(:,2:3), MASK = B(:,2:3) )         ! L = 5

M = MAXVAL( A(:,2:3), MASK = A(:,2:3).LE.5 )    ! M = 5
```

7.1.3 Logical Arrays

The logical reduction functions (**ANY**, **ALL**, **COUNT**) take a logical array (identified with the keyword **MASK**) and an optional **DIM** argument and work with the elements that are **.TRUE.**

```
LOGICAL-REDUCTION-FUNCTION( MASK [, DIM] )
```

For example, consider the logical array **C**:

$$C = \begin{bmatrix} \cdot & T & \cdot & T \\ \cdot & T & \cdot & T \\ \cdot & T & \cdot & T \end{bmatrix}$$

ANY returns the scalar value **T** if any of the elements in the array (or section) is true. If a dimension is specified, the result is a front-end array of rank one less than the argument array:

```
X   = ANY( MASK=C )                  ! X   = T
XX  = ANY( MASK=C, DIM=2 )           ! XX  = [ T,T,T ]
XXX = ANY( MASK=C(:,1:3), DIM=1 )    ! XXX = [ F,T,F ]
```

ALL returns **T** only if all the elements in the array or on the specified dimension are true:

```
Y   = ALL( MASK=C )                  ! Y   = F
YY  = ALL( MASK=C, DIM=1 )           ! YY  = [ F,T,F,T ]
YYY = ALL( MASK=C(:,2:4:2) )         ! YYY = T
```

COUNT returns the number of true elements in the array or on the specified dimension:

```
Z   = COUNT( MASK=C(:,4) )           ! Z   = 3
ZZ  = COUNT( MASK=C(:,2:4:2) )       ! ZZ  = 6
ZZZ = COUNT( MASK=C(1:2,:) DIM=1)    ! ZZZ = [ 0,2,0,2 ]
```

7.2 Element Location

The location functions determine the location in an array of certain element values. The numeric location functions are **MAXLOC** and **MINLOC**; the logical location functions are **FIRSTLOC** and **LASTLOC**, referring to the first and last true elements in array-index order. A related function, **PROJECT**, determines the value of a numeric array element that corresponds to the first true element of a mask (logical) array.

7.2.1 Numeric Arrays

MAXLOC and **MINLOC** determine the array indices of the maximum or minimum value in a numeric array (or section). These functions return a vector whose elements are the array indices of the value in question. (Notice that the result vector's length therefore equals the rank of the argument array.) The result vector resides on the CM.

```
NUMERIC-LOCATION-FUNCTION( ARRAY [, MASK] )
```

For example, consider the effect of the location functions on a 2-dimensional array **A**:

$$A = \begin{bmatrix} 1 & 5 & 3 & 7 \\ 4 & 2 & 6 & 3 \\ 9 & 2 & 1 & 5 \end{bmatrix}$$

the numeric location functions behave as follows:

```
I = MAXLOC( A )                   ! I = [ 3,1 ]

J = MINLOC( A(:,1:2) )            ! J = [ 1,1 ]
```

When the argument is an array section, as in the second line above, the indices returned refer to the section, not to the parent array. This becomes more obvious when the section is not taken from the origin of the array:

```
K = MAXLOC( A(:,3:4) )            ! K = [ 1,2 ]

L = MINLOC( A(:,3:4) )            ! L = [ 3,1 ]
```

Special Cases

If more than one element value meets the condition, the one chosen is unpredictable. For example, the array **A** above has two instances of the value 1. Thus:

```
K = MINLOC( A )                ! K = [ 1,1 ]  OR  K = [ 3,3 ]
```

If the argument array is only one dimension, the result is a CM vector of length 1 (not a scalar value):

```
L = MINLOC( A(:,3) )           ! L = [ 1 ]
```

No DIM Argument

The numeric location functions do not take a dimension argument. The element located is always a maximum or minimum of the whole array or section.

MASK Argument

The numeric location functions can take a **MASK** argument to indicate which values are candidates. The mask is a logical array or array-valued expression that conforms to the argument array. For example:

```
I = MAXLOC( A, MASK=A.LT.7 )                   ! I = [ 2,3 ]

J = MINLOC( A(:,2:3), MASK=A(:,2:3).GE.3 )    ! J = [ 1,2 ]
```

Similarly, if array **B** is specified as a mask for **A**:

$$A = \begin{bmatrix} 1 & 5 & 3 & 7 \\ 4 & 2 & 6 & 3 \\ 9 & 2 & 1 & 5 \end{bmatrix} \qquad B = \begin{bmatrix} \cdot & T & \cdot & T \\ T & T & \cdot & T \\ \cdot & T & T & \cdot \end{bmatrix}$$

Then,

```
K = MAXLOC( A, MASK=B )                  ! K = [ 1,4 ]

L = MAXLOC( A(:,1:2), MASK=B(:,1:2) )   ! L = [ 1,2 ]
```

7.2.2 Logical Arrays

Unlike the numeric location functions, the logical location functions do not return a set of indices. Instead, the functions **FIRSTLOC** and **LASTLOC** take a logical array and return a logical array of the same shape with at most one true element. The one true element is in the location of the first (or last) true element of the argument array in array-index order. If the argument array contains no true values, the result array is also all false.

```
LOGICAL-LOCATION-FUNCTION( ARRAY [, DIM] )
```

For example,

```
B = FIRSTLOC( A )
```

assigns the following array to **B** given the values shown in **A**:

$$A = \begin{bmatrix} \cdot & \cdot & T & T \\ \cdot & T & \cdot & \cdot \\ \cdot & \cdot & \cdot & T \end{bmatrix} \Rightarrow B = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & T & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

Similarly,

```
C = LASTLOC( A )
```

assigns the following array to **C** given the values shown in **A**:

$$A = \begin{bmatrix} \cdot & \cdot & T & T \\ \cdot & T & \cdot & \cdot \\ \cdot & \cdot & \cdot & T \end{bmatrix} \Rightarrow C = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & T \end{bmatrix}$$

If the argument array is a section of **A**,

```
D = LASTLOC( A(:,1:3) )
```

the result is an array that conforms to the section:

$$A = \begin{bmatrix} \cdot & \cdot & T & T \\ \cdot & T & \cdot & \cdot \\ \cdot & \cdot & \cdot & T \end{bmatrix} \Rightarrow D = \begin{bmatrix} \cdot & \cdot & T \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$$

DIM Argument

FIRSTLOC and **LASTLOC** can take an optional dimension argument, in which case the result array locates the first true element in each row, column, etc., of the specified dimension. For example,

```
E = FIRSTLOC( A, DIM=1 )
```

creates the following array **E** given the values shown in **A**:

$$A = \begin{bmatrix} \cdot & \cdot & T & T \\ \cdot & T & \cdot & \cdot \\ \cdot & \cdot & \cdot & T \end{bmatrix} \quad \Rightarrow \quad E = \begin{bmatrix} \cdot & \cdot & T & T \\ \cdot & T & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

Similarly,

```
F = LASTLOC( A, DIM=2 )
```

creates the following array **F** given the values shown in **A**:

$$A = \begin{bmatrix} \cdot & \cdot & T & T \\ \cdot & T & \cdot & \cdot \\ \cdot & \cdot & \cdot & T \end{bmatrix} \quad \Rightarrow \quad F = \begin{bmatrix} \cdot & \cdot & \cdot & T \\ \cdot & T & \cdot & \cdot \\ \cdot & \cdot & \cdot & T \end{bmatrix}$$

7.2.3 Element Location Example

Location functions can be used, in combination with a numeric reduction function, to locate the first of multiple instances of the maximum or minimum value of a numeric array. (Recall that the numeric location functions make an arbitrary choice among multiple instances of a maximum or minimum value.)

For example, for an array **A** of any rank, the following line computes the indices of the first minimum. The mask is a logical array with the value **.TRUE.** at the first location where the value of **A** equals the minimum value of **A** and **.FALSE.** elsewhere. The function **MINLOC** then returns the array index or indices of that position:

```
        LOC1 = MINLOC( A, MASK = FIRSTLOC( A .EQ. MINVAL(A) ))
```

The location functions are general-purpose procedures that are not particularly fast. For some special cases, it is possible to determine element location without using these functions. For example, in the special case of a 1-dimensional array, an array constructor of the same length as the array of interest serves as the array argument:

```
        LOC2 = MINVAL( [1:M], MASK = V .EQ. MINVAL(V) )
```

Notice that the array constructor **[1:M]** used as the argument array represents the *indices* of vector **V**. The outer call to **MINVAL** returns the lowest index where the value of **V** equals the minimum value of **V**.

A notable difference between these two lines of code—besides their speed—is that in the first, the variable **LOC1** is a CM vector of length equal to the rank of the argument array **A**, whereas **LOC2** in the second line is a scalar value residing on the front end. It is because the array constructor argument can only be 1-dimensional and because the result is a scalar that the second technique is useful only with vectors.

7.3 Array Construction

The array construction functions create new CM arrays from the information in existing ones:

- **RESHAPE** takes an array and constructs a new array with the same elements but a different shape.
- **DIAGONAL** takes a vector and constructs a matrix whose diagonal elements are those of the vector and whose other ("fill") elements are all a specified or default value.
- **MERGE** combines two conformable arrays into a new array by means of an elementwise choice guided by a logical mask.
- **PACK** and **UNPACK** behave as gather and scatter operations. **PACK** gathers an *n*-dimensional array into a vector; **UNPACK** scatters a vector into an *n*-dimensional array.

- **REPLICATE** and **SPREAD** construct arrays from a specified number of copies of the argument array. **SPREAD** adds a new dimension to accommodate the copies; **REPLICATE** lengthens one of the existing dimensions.

This section introduces the array construction functions by illustrating the behavior of **RESHAPE**, **MERGE**, and **SPREAD**. For more information about all these functions, please see the *CM Fortran Reference Manual.*

7.3.1 Reshaping an Array

With the CM system's distributed memory, reshaping arrays is not as routine a practice as it is with linear memory machines. As mentioned in Chapter 2, reshaping a single large array should not be used as a substitute for separate declarations of smaller arrays, each in the shape needed. On the CM system, reshaping entails actual data movement—most often with the router communication network—rather than a substitution of indices.

Array reshaping in CM Fortran is useful when the algorithm requires manipulating the same set of data in more than one shape. The intrinsic function **RESHAPE** creates a new array with the same elements as the argument array, but with a different shape and perhaps a different size. Its format is:

```
RESHAPE( MOLD, SOURCE [, PAD] [, ORDER] )
```

The Target Array

The **MOLD** argument specifies the target shape; it is a vector of positive integers, each indicating the extent of a target dimension. (Since each vector element corresponds to an array dimension, the vector's length is limited to 7.) For example, the following calls reshape a matrix **A(100,100)** in various ways:

```
B  =  RESHAPE( [ 10000 ], A )          ! One dimension
C  =  RESHAPE( [ 10,10,100 ], A )      ! Three dimensions
D  =  RESHAPE( [ 10,10,10,10 ], A )    ! Four dimensions
```

Unless the call specifies otherwise, the source array elements are placed in the target array in array-index order. For example, the call

```
X  =  RESHAPE( [3,4], SOURCE=[1:12] )
```

places the following values in the 3 x 4 array **X**:

$$X = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

Any source array values that do not fit into the mold are ignored. For example, the result array **X** in the above example would be the same if **SOURCE** had been **[1:20]**.

Padding the Target Array

If the source array is smaller than the mold, the call must include the **PAD** argument. **PAD** is an array of the same type as **SOURCE** and any size or shape. When the source array elements are used up, the system uses one or more copies of the pad array elements (in array-index order) to fill the target array. For example:

```
Y  =  RESHAPE( [3,4], SOURCE=[1:5], PAD=[10:12] )
```

places the following values in the 3 x 4 array **Y**:

$$Y = \begin{bmatrix} 1 & 4 & 11 & 11 \\ 2 & 5 & 12 & 12 \\ 3 & 10 & 10 & 10 \end{bmatrix}$$

Permuting the Axes

The optional **ORDER** argument is used to change the order in which the target dimensions are filled. The default order is the vector [1:*n*]; the alternative order for a 2-dimensional mold would be [2,1], with the following effect:

```
Z = RESHAPE( [3,4], SOURCE=[1:5], PAD=[10:12], ORDER=[2,1] )
```

$$Z = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 10 & 11 & 12 \\ 10 & 11 & 12 & 10 \end{bmatrix}$$

7.3.2 Merging Arrays

The function **MERGE** constructs an array of the same shape as two conformable argument arrays. It then fills the new array with values from one or the other of the source arrays, depending on the corresponding value in a logical mask. Its format is:

```
MERGE( TSOURCE, FSOURCE, MASK )
```

For example, given two source arrays **A** and **B** and a logical array **M**:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 6 & 7 & 8 & 9 \\ 6 & 7 & 8 & 9 \\ 6 & 7 & 8 & 9 \end{bmatrix} \quad M = \begin{bmatrix} \cdot & T & \cdot & T \\ \cdot & T & \cdot & T \\ \cdot & T & \cdot & T \end{bmatrix}$$

the following call yields the array **C** shown:

```
C = MERGE( TSOURCE=A, FSOURCE=B, MASK=M )
```

$$C = \begin{bmatrix} 6 & 2 & 8 & 4 \\ 6 & 2 & 8 & 4 \\ 6 & 2 & 8 & 4 \end{bmatrix}$$

Alone among the transformational intrinsic functions, **MERGE** need not perform implicit data motion. If its three conformable arguments and the destination array are whole arrays, their corresponding elements are already lined up in the same set of virtual processors (unless the program specifies otherwise with a compiler directive). The behavior of **MERGE** is the same as the behavior of a **WHERE** construct (as described in Section 4.1.2):

```
WHERE ( M )
   C = TSOURCE
ELSEWHERE
   C = FSOURCE
END WHERE
```

If the arguments and the destination are array sections drawn from different parts of conformable parents or if they are sections drawn from any part of nonconformable parents, the system must perform interprocessor data motion to align the corresponding elements in the same set of virtual processors.

7.3.3 Spreading an Array

SPREAD takes a source array and creates a new array with an additional dimension. It then broadcasts a specified number of copies of the argument array along the specified dimension of the new array. Its format is:

```
SPREAD( SOURCE, DIM, NCOPIES )
```

For example, if the source array is the vector

```
A = [ 4, 2, 6, 3 ]
```

Then,

```
B = SPREAD( A, DIM=1, NCOPIES=3 )
```

replicates the values in **A** along the first dimension of a new 3 x 4 array. The value of **B** in this assignment statement becomes:

$$B = \begin{bmatrix} 4 & 2 & 6 & 3 \\ 4 & 2 & 6 & 3 \\ 4 & 2 & 6 & 3 \end{bmatrix}$$

Similarly, spreading three copies of the same source vector along the second dimension of a new array

```
C = SPREAD( A, DIM=2, NCOPIES=3 )
```

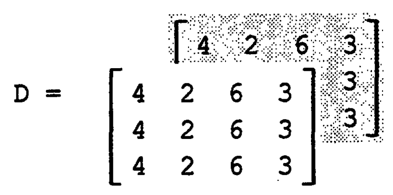
results in assigning the following values to **C**:

$$C = \begin{bmatrix} 4 & 4 & 4 \\ 2 & 2 & 2 \\ 6 & 6 & 6 \\ 3 & 3 & 3 \end{bmatrix}$$

When the argument array is 2-dimensional, the result array is 3-dimensional. For example, spreading two copies of **B** along the third dimension

```
      D = SPREAD( B, DIM=3, NCOPIES=2 )
```

results in the following 3 x 4 x 2 array, assigned to **D**:



7.4 Array Multiplication

The array multiplication functions are **DOTPRODUCT** for vectors and **MATMUL** for vectors or matrices. For example, given two vectors such as the following array constructors, their vector dot product is:

```
      I = DOTPRODUCT( [ 1,2,3 ], [ 2,3,4 ] )      ! I = 20
```

And, to compute the matrix-matrix product of two arrays, such as **A** and **B**,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}$$

the code and its effect are:

```
      C = MATMUL( A,B )
```

$$C = \begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$$

Many other mathematical procedures are provided as subroutines in the CM Scientific Software Library. These are described in a separate volume.

Chapter 8

The FORALL Statement

A FORALL statement describes a collection of assignments to designated array elements, to be executed in undefined order but as if simultaneously. This *elemental array assignment* is an extremely powerful way to express location-dependent action. It can be used for:

- Dynamically initializing arrays with sequences of values.
- Selecting subarrays, both by position and by value.
- Data motion in patterns that are otherwise difficult to express in CM Fortran as parallel operations. These include *parallel prefix* (cumulative) computations along array dimensions and certain operations on irregularly shaped data.

FORALL provides many of the same capabilities as DO constructs, including those with nested DO's and those with embedded IF statements. However, since the individual assignments occur in an undefined order, FORALL statements can be executed in parallel on the CM.

Although not a part of the draft standard Fortran 90, FORALL is included in CM Fortran because of its particular expressive power. In some ways, FORALL is redundant with other CM Fortran features, such as assignment of array sections specified with triplet notation or vector-valued subscripts. However, FORALL is the only feature that can express certain very efficient CM operations, such as parallel prefix operations.

After introducing the syntax of the FORALL statement, this chapter examines its execution model—particularly the significance of *as if simultaneous* execution—and concludes with some examples of using FORALL to express various patterns of data motion.

8.1 Syntax

FORALL uses triplet notation to specify one or more sequences of integer values and associates each with a symbolic name. The names are then used in expressions in an assignment statement, where they can indicate either array indices or values. The basic format is:

FORALL (*triplet-spec*[*s*]) *assignment*

For example, to seed the array **IDENTITY(N)** with "self-addresses," that is, to assign each element its own index value:

```
FORALL ( I=1:N )  IDENTITY(I) = I
```

Similarly, to clear every tenth element of vector **V(N)**:

```
FORALL ( I=1:N:10 )  V(I) = 0
```

In simple examples like these, **FORALL** is redundant with other CM Fortran features. These statements can also be expressed as CM array assignments, using an array constructor in the first case and a triplet subscript in the second:

```
IDENTITY = [ 1:N ]

V( 1:N:10 ) = 0
```

However, **FORALL** is the feature of choice for more elaborate assignments, such as those involving multidimensional arrays and combinations of symbolic names. For example, the following statement initializes matrix **H** to contain a Hilbert matrix of size **N**:

```
FORALL ( I=1:N, J=1:N )  H(I,J) = 1.0 / REAL( I+J-1 )
```

Notice that the definition of the symbolic names in a **FORALL** statement resembles the definition of index variables in a **DO** construct. Where **DO** might define **I** as **I=1,N,10**, **FORALL** substitutes colons, **I=1:N:10**. This chapter adopts the term *index variable* to refer to a symbolic name defined in a **FORALL** triplet-spec.

8.1.1 The FORALL Assignment

The assignment in a **FORALL** statement is the same as the assignment statement in a **DO** construct. In CM Fortran, both **DO** and **FORALL** can assign individual array elements or array objects (sections).

FORALL assignments have certain restrictions: no left-hand-side element can be assigned more than one value, and, as with **DO** constructs, a function reference appearing anywhere in the assignment must not alter the value of an index variable.

A **FORALL** statement can have only one assignment. That is, there is no block **FORALL** construct and no embedding of **WHERE** or other CM Fortran statements.

The Target Elements

The left-hand side of a **FORALL** assignment uses the index variables to indicate the array indices of interest. It can do so either with scalar subscripts in the Fortran 77 manner,

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

or with triplet subscripts in the Fortran 90 manner,

```
      FORALL ( K=1:N )  A(K,:) = expression

      FORALL ( K=1:N )  A(1:K, K:1:-1) = expression
```

When the left-hand side is specified with scalar subscripts, the target of each assignment is a single array element. When the left-hand side includes triplet subscripts, the target of each assignment is an array section (as described in Chapter 4).

In either case, the left-hand side must use *all* the index variables defined in the statement; array references without subscripts are not permitted. For example, the following assignments to vector **B** are *not* legal, the first because no index variables appear on the left and the second because **J** does not appear on the left:

```
      FORALL ( I=1:N, J=1:M ) B = I+J      ! Error

      FORALL ( I=1:N, J=1:M ) B(I) = J     ! Error
```

As in a DO construct, the index variables can be used either as primary expressions or as operands in an operation that specifies the target elements:

```
FORALL ( I=1:N, J=1:M ) C(I + J*2) = J

FORALL ( I=1:N, J=1:M ) C(:, (I + J*2)) = J

FORALL ( K=0:10 ) LOGS( 2**K:2**(K+1)-1 ) = K
```

The target elements specified need not constitute the whole array, of course. FORALL provides two ways to select a subarray by element position:

- By having the index variables refer to a sequence that covers only part of the corresponding array dimension:

```
DIMENSION A(100,100)
...
FORALL (I=1:50, J=1:50) A(I,J) = 0        ! Clear one quadrant

FORALL (K=2:100:2, L=5:100:5) A(K,L) = 0 ! Selected elements
```

- By indicating an array section on the left-hand side of the assignment:

```
DIMENSION B(10,10)
...
FORALL (M=1:10)  B(M,:) = M       ! Self-index the first dim.
```

The Source Values

The right-hand side of a FORALL assignment must provide no more than one value for each element specified on the left. The set of permissible expressions varies depending on whether the left-hand side is a series of single array elements (scalar subscripts) or a series of array sections (triplet subscripts).

If the target elements are specified with scalar subscripts, the right-hand side must be a scalar-valued expression. For example, given the three arrays shown, the following statements assign a scalar value to each successive element of vector A:

```
DIMENSION A(N),B(N)
DIMENSION C(N,M)

FORALL (I=1:N)  A(I) = 10          ! Constant expression
FORALL (I=1:N)  A(I) = I           ! Scalar variable
FORALL (I=1:N)  A(I) = B(I)        ! Array element
FORALL (I=1:N)  A(I) = SIN(I)      ! Elemental (scalar) function
FORALL (I=1:N)  A(I) = SUM(C(I,:)) ! Scalar-valued intrinsic
FORALL (I=1:N)  A(I) = FUNC(B(I))  ! Scalar-valued user func.
```

If, on the other hand, the left-hand side of a **FORALL** assignment is specified with triplet subscripts, the target of each assignment is an array section rather than an individual element. In this case, the right-hand expression must conform to each of the target array sections. Thus, the right-hand side can be either:

- Any of the scalar-valued expressions listed above. Recall that a scalar is replicated to conform to any array.
- An array-valued expression of the same shape as a target array section.

For example, given the arrays shown, the following statements assign a vector to each successive row or column of matrix **A**. (Several of these options are illustrated below.)

```
DIMENSION A(N,M),B(N,M)
DIMENSION C(N)

FORALL (I=1:N)  A(I,:) = 10       ! Scalar extension (to rows)
FORALL (I=1:M)  A(:,I) = C        ! Array (to columns)
FORALL (I=1:N)  A(I,:) = B(I,:)   ! Array section
FORALL (I=1:N)  A(I,:) = [1:M]    ! Array constructor
FORALL (I=1:M)  A(:,I) = SIN(C)   ! Elemental (array) function
FORALL (I=1:M)  A(:,I) = SUM(B,2) ! Array-valued intrinsic
FORALL (I=1:N)  A(I,:) = FUNC(C)  ! Array-valued user func.
```

To illustrate some of these cases, assume that **A** is a 4 x 3 matrix. In the case of a scalar-valued right-hand expression,

```
FORALL ( I=1:3 ) A(:,I) = I
```

you can picture each of the three scalar values of **I** being promoted to a vector for (elemental) assignment to the corresponding column of array **A**. (Alternatively, you can picture the promotion of the scalars as spreading the whole vector sequence **I** on the first dimension of **A**.)

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

The effect is similar when the right-hand expression is an array constructor that conforms to the target section. In this example, a vector is assigned to each successive column. Since the right-hand side in this example is the same for every assignment, the effect is a spread operation on the second dimension of **A**.

```
FORALL ( I=1:3 ) A(:,I) = [ 1:4 ]
```

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \\ 4 & 4 & 4 \end{bmatrix}$$

Similarly, consider an array-valued intrinsic function whose result is of the appropriate shape. The function reference **SUM(B,2)** returns a vector containing the sum of the elements in each row of matrix **B**. If **B**'s and **A**'s first dimensions (that is, their columns) are of the same length, the vector result of **SUM(B,2)** conforms to each of the columns of **A**.

```
FORALL (I=1:N)  A(:,I) = SUM( B, DIM=2 )
```

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{bmatrix} \quad \Rightarrow \quad A = \begin{bmatrix} 15 & 15 & 15 \\ 20 & 20 & 20 \\ 25 & 25 & 25 \\ 30 & 30 & 30 \end{bmatrix}$$

Again, the result is a spread operation because the right-hand expression is the same for each of the assignments. Some complicated patterns of data movement occur when the right-hand side references the index variables and varies accordingly. See below, Section 8.3, for examples that use **FORALL** to specify these other patterns of data movement.

8.1.2 The FORALL Mask

A **FORALL** statement can contain a mask expression, which prevents certain elements from being assigned. The mask is a scalar-valued expression, placed inside the parentheses that enclose the triplet-specs:

FORALL (*triplet-spec*[*s*] [, *scalar-mask*]) *assignment*

The mask expression usually references one or more index variables, although it can be a constant expression. Its action is comparable to embedding an **IF** statement in a **DO** construct.

For example, to avoid division by zero in a **FORALL** statement:

```
FORALL ( I=1:N, J=1:M, A(I,J).NE.0.0 ) B(I,J) = 1.0/A(I,J)
```

Similarly, to clear the part of a square matrix below the diagonal:

```
FORALL ( I=1:N, J=1:N, I.GT.J ) H(I,J) = 0.0
```

Order of Execution

The mask is applied as the third step of a four-step **FORALL** operation:

1. Compute the combinations of index variables.
2. Compute the mask value, true or false, for each combination of index variables.
3. Evaluate the right-hand expression only for the combinations of index variables where the mask expression is true.
4. Perform the assignments (in unspecified order) where the right-hand side has been evaluated.

Array-Valued Masks

The **FORALL** mask expression is always scalar-valued, even when the left-hand side of the assignment is an array section and the right is array-valued:

```
FORALL ( I=1:N, I.GT.5 ) B(I,:) = A(:,I)  ! Legal: scalar mask

FORALL ( I=1:N, A.GT.5 ) B(I,:) = A(:,I)  ! Error: array mask
```

However, array-valued masks can appear in intrinsic function references in the **FORALL** assignment. Notice that the two masks in the following statement are entirely independent of each other:

```
FORALL (I=1:N, I.NE.5) C(I) = SUM( D(:I,:I), MASK = D.GT.0 )
```

This example sums the positive elements in varying sections of array **D** and assigns the results to the successive elements of array **C**. The array-valued mask argument to **SUM** excludes the zero and negative elements of **D**; the scalar-valued **FORALL** mask excludes the fifth element of **C**.

8.2 Execution Model

This chapter has pointed out many ways in which a **FORALL** statement resembles a **DO** construct. The most significant difference between them is the order of execution of the individual assignments. Since the order is undefined, but as if simultaneous, the assignments can be simultaneous in fact when **FORALL** is executed by the CM.

8.2.1 As If Simultaneous

Any **FORALL** statement can be translated into a **DO** construct with equivalent effects. Consider a simple example:

```
FORALL ( I=1:N )  A(I) = I

DO I=1,N
   A(I) = I
END DO
```

Both these operations seed array **A** with its own index values. However, the semantics of the two operations differ in one crucial respect: the order of execution of the **DO** loop is index order, while the order of execution of the **FORALL** assignments is undefined. The **FORALL** statement, but not the **DO** loop, can therefore be executed in parallel by the individual CM processors.

The significance of as-if-simultaneous execution becomes clear when we use **FORALL** to assign an element whose own value is assigned to some other element. For example, consider a transposition of an **N** x **N** matrix **B**:

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

Notice that the statement does not explicitly save the initial array values in a temporary location, as a **DO** construct would need to do. If all the element values are moved simultaneously, an element being moved cannot have already received the value of its opposite number. (In fact, when executed serially, **FORALL** creates any necessary temporaries transparently to the user.) Thus, the programmer need not take steps to ensure that only the original values are used in a **FORALL** operation. All the elemental assignments are executed *as if* at the same moment, when only the original values are available.

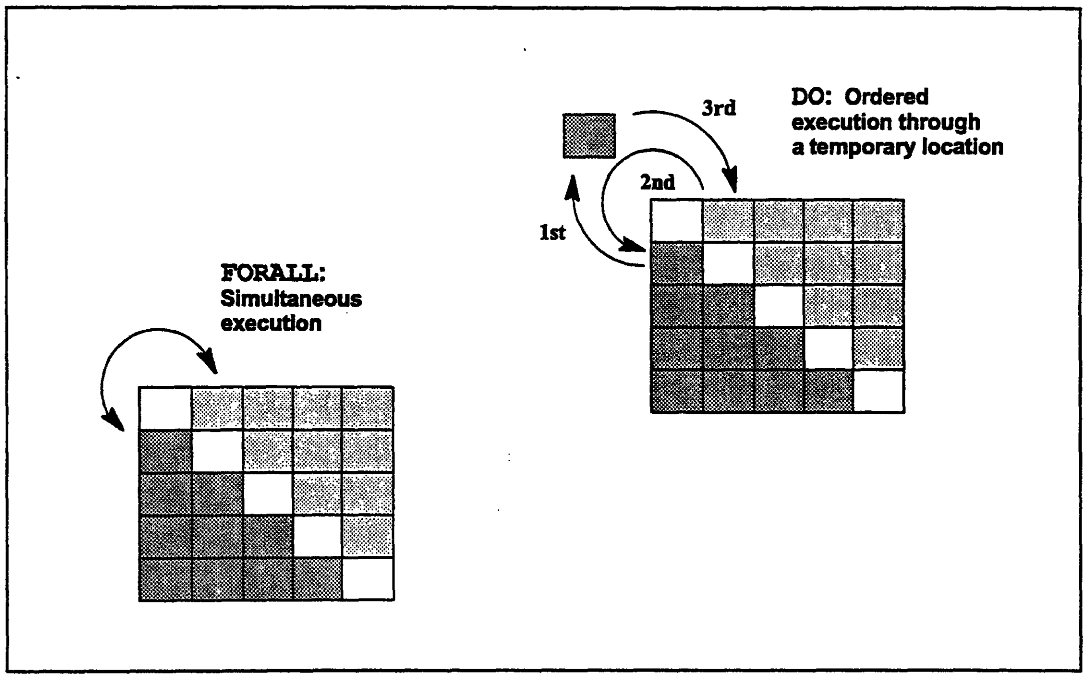


Figure 7. Execution order of **DO** and **FORALL** in transposing array elements (2,1) and (1,2)

8.2.2 Serial versus Parallel

The FORALL statement is not considered an array operation for the purpose of determining an array's home. FORALL can operate on either CM arrays or front-end arrays (or both at once), and it can execute either serially on the front-end computer or in parallel on the CM.

A third possible mode of execution is *serially on the CM.* When either FORALL or DO includes an array operation—such as assigning an array section, evaluating an array-valued expression, or invoking a transformational intrinsic function—the array operation is performed on the CM. The complete FORALL or DO operation is considered serial if the CM executes the assignment statement separately for each combination of index variables. The complete FORALL operation is considered parallel if the CM executes the assignment statements for all combinations of index variables at once.

CM Fortran programmers usually intend that FORALL be executed in parallel, for reasons of performance. When executed serially, either on the front end or on the CM, FORALL has no advantage over DO except perhaps some syntactical convenience. In fact, when FORALL needs to execute serially, the CM Fortran compiler generates a DO construct.

This section outlines the conditions under which FORALL executes serially or in parallel.

Determined by Array Home

Generally, the homes of the arrays referenced in the FORALL assignment determine whether FORALL executes serially or in parallel (the next section notes the exceptions). Specifically:

- FORALL assignments that reference only front-end arrays always execute serially:

```
FORALL (I=1:N)  FE_ARRAY(I) = I               ! Serial, FE
FORALL (I=1:N)  FE_ARRAY(I) = FE_ARRAY(I+1)   ! Serial, FE
```

- FORALL assignments that reference only CM arrays generally execute in parallel (with exceptions as noted below):

```
FORALL (I=1:N)  CM_ARRAY(I) = I               ! Parallel
FORALL (I=1:N)  CM_ARRAY(I) = CM_ARRAY(I+1)   ! Parallel
```

- **FORALL** assignments that reference both front-end arrays and CM arrays always execute serially:

```
FORALL (I=1:N)  FE_ARRAY(I) = CM_ARRAY(I)     ! Serial, FE
FORALL (I=1:N)  CM_ARRAY(I) = FE_ARRAY(I)     ! Serial, FE
```

When a **FORALL** assignment references arrays with different homes, the system moves CM values one at a time to or from the front end to perform the serial operations. As with any mixed-home array operation, this data transfer degrades program performance considerably. See Chapter 2 for discussion of mixed-home operations and some suggestions for avoiding them.

Determined by Kind of Expression

Certain kinds of expressions in the **FORALL** assignment always cause the statement to be executed serially, even when only CM arrays are referenced. These are:

- Any reference to a user-defined function, including a statement function:

```
FORALL (I=1:N)  A(I) = USER_FUNCTION(I)       ! Serial, FE
```

- Use of a **FORALL** index variable in an array constructor:

```
FORALL (I=1:N)  B(I:N,:) = [ I:I+N-1 ]        ! Serial, CM
```

- Most references to transformational intrinsic functions:

```
FORALL (I=1:N) B(I,:) = ANY_TRANSFORM( C(I,:) ) ! Serial, CM
```

  Future versions of CM Fortran will remove this restriction for some transformational intrinsic functions and thus enable **FORALL** statements that reference them to execute in parallel.

Determined by Temporary Restrictions

The implementation of serial **FORALL** is complete, but development is continuing on the parallel version. As new **FORALL** capabilities are added, their initial implementation may be serial. Later versions should remove these restrictions on new capabilities and enable

FORALL to execute them in parallel. The CM Fortran compiler generates a warning whenever it serializes a FORALL statement.

NOTE

Please see the *CM Fortran Release Notes* for the current version for any restrictions on parallel execution of FORALL.

8.3 Data Motion

The FORALL statement offers an alternative to other CM Fortran features in expressing various forms of data motion. The alternatives often vary in syntactical convenience and sometimes in performance. FORALL statements are often as fast as the fastest alternative means of expression, though rarely faster.

In addition, FORALL can express several patterns of data motion that are otherwise difficult to express as parallel operations in CM Fortran. Among these are arbitrary permutations of multidimensional arrays, operations on nonrectangular or irregular data, and parallel prefix operations along array dimensions.

8.3.1 Compared with Other Features

FORALL can express all the array transformations and data motion operations described in Chapters 6 and 7, mimicking the behavior of array assignments that reference array sections, vector-valued subscripts, and transformational intrinsic functions. This section reviews the major categories of data motion already discussed and illustrates FORALL in expressing each. The sections that follow show patterns of movement for which FORALL is the only convenient choice.

Grid Communications

Grid communications, including dimensional shifts and other nearest-neighbor accesses, have in common the fact that element values all move the same distance in the same direction to reach their respective destinations.

Grid operations expressed with **FORALL** and with array sections both use the CM system's NEWS communication network and achieve about the same performance. For power-of-2 shifts on power-of-2 array dimensions, the shift intrinsics also use NEWS. Thus, to shift matrix values by, say, four positions on the first dimension, the following statements all yield about the same performance:

```
DIMENSION A(N,M), B(N,M)
. . .
FORALL (I=1:N-4) B(I,:) = A(I+4,:)

B(1:N-4,:) = A(4:N,:)

B = CSHIFT( A, DIM=1, SHIFT=4 )
```

The semantics of the last statement differs slightly from the other two: the function **CSHIFT** wraps the elements that move off the edge of the array, whereas the other two statements explicitly prevent any elements from moving off the edge.

General Communications

General communications, including transpositions and arbitrary permutations, have in common the fact that element values move different distances or different directions to reach their respective destinations.

General communication operations, whether expressed with **FORALL**, with vector-valued subscripts, or with transformational intrinsic functions, always use the router communication network. This more general facility is of course slower than the NEWS network, which optimizes the special case of grid communication.

FORALL and vector-valued subscripts are the only features CM Fortran has for expressing an unordered set of indices. For example, the following statement uses two index vectors to access locations in a matrix. The lengths of the index vectors **V** and **R** are unrelated to the source array **C**, but they must equal the length of the corresponding dimensions of the destination array **D**:

```
DIMENSION C(M,M), D(N,N)
DIMENSION V(N), R(N)
...
D = C(V,R)
```

FORALL can mimic this behavior, albeit in a more cumbersome manner:

```
FORALL (I=1:N, J=1:N) D(I,J) = C( V(I), R(J) )
```

The real power of **FORALL**, however, is that it is not limited to vectors for indexing into multidimensional arrays. **FORALL** can perform a perfectly arbitrary permutation such as the following, which uses index arrays **X** and **Y**:

```
FORALL (I=1:N,J=1:M) D(I,J) = C( X(I,J), Y(I,J) )
```

Global Communication

A third form of communication involves cumulative computations along array dimensions. Examples of such *global communication* are the reduction intrinsics and the transformational function **SPREAD** (see Chapter 7). Global communication is much faster than general communication on the CM system, although not as fast as NEWS.

FORALL can express a spread operation—that is, replicate an array along a specified dimension of another array of rank one higher—with about the same performance as the function **SPREAD**. For example, both the following statements spread a vector along the first dimension of a matrix:

```
DIMENSION A(N,M), V(M)
...
FORALL ( I=1:N ) A(I,:) = V
```

is functionally equivalent to:

```
A = SPREAD( V, DIM=1, NCOPIES=N )
```

Similarly, **FORALL** can express a "reduce and spread" operation, where the result of a reduction operation along an array dimension is replicated across a dimension:

```
DIMENSION A(N,M)
...
FORALL ( I=1:M ) A(:,I) = SUM( A, DIM=2 )
```

is functionally equivalent to:

```
A = SPREAD( SUM( A, DIM=2 ), DIM=2, NCOPIES=M )
```

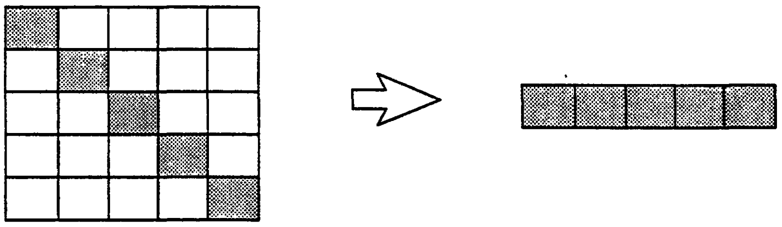
This example computes the sum of each row of **A** and returns the result as a vector. The vector is then replicated across the rows of **A**, with the result that each element gets the sum of all the elements on the second dimension:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix} \Rightarrow A = \begin{bmatrix} 10 & 10 & 10 & 10 \\ 14 & 14 & 14 & 14 \\ 18 & 18 & 18 & 18 \\ 22 & 22 & 22 & 22 \end{bmatrix}$$

8.3.2 Operations on Irregular Data

One of the major uses of **FORALL** is to express operations on irregularly shaped parts of an array. Such operations can be expressed serially with a **DO** construct, but **FORALL** is the only convenient way to express them for the CM.

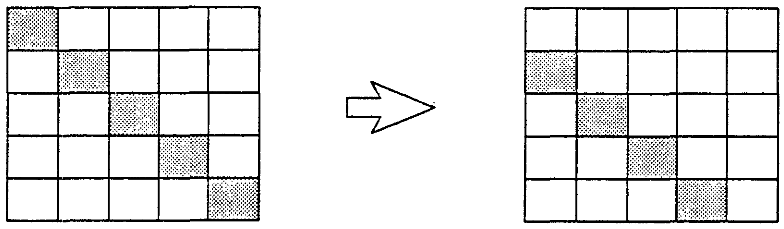
A simple example of an operation on an irregularly shaped part of an array is extracting the diagonal elements of a matrix:



The **FORALL** expression of this operation is:

```
DIMENSION C(N,N)
DIMENSION D(N)
...
FORALL ( I=1:N ) D(I) = C(I,I)
```

Similarly, consider shifting the values in a matrix diagonal, say, one position to the left:



The elemental assignments that occur in this operation are:

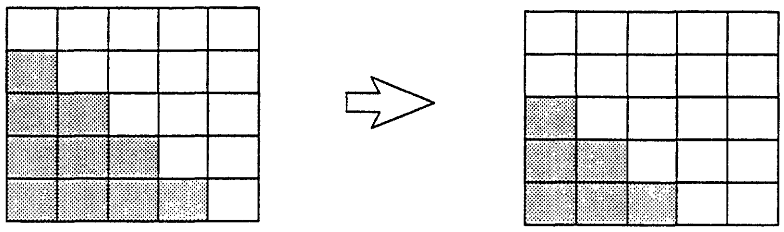
```
A(2,1) = A(2,2)
A(3,2) = A(3,3)
A(4,3) = A(4,4)
A(5,4) = A(5,5)
```

or, as expressed with **FORALL**:

```
FORALL (I=2:5)  A(I,I-1) = A(I,I)
```

As with **DO**, the **FORALL** expressions can describe a number of possible shapes. For example, the following statement shifts the lower triangle of a matrix by one position:

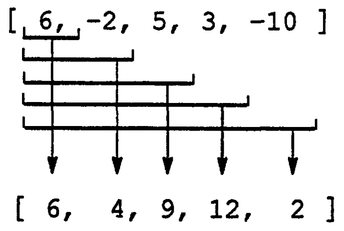
```
FORALL (I=1:N, J=1:I-1)  A(I,J) = A(I, J+1)
```



8.3.3 Parallel Prefix Operations

Parallel prefix operations, sometimes called *scans*, apply some combining operation to the elements along an array dimension; that is, they compute for each element the combination of itself and all previous elements on that dimension. For example, a sum-prefix (or *add-*

scan) is the familiar process of balancing a checkbook: each credit or debit is added to the sum of the previous ones to yield a running subtotal:



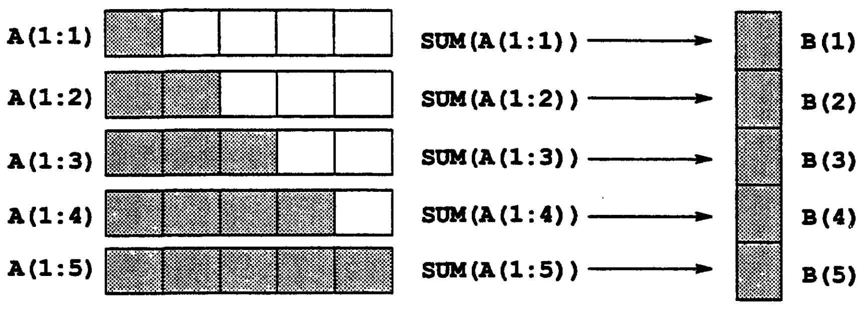
Parallel prefix operations are used by many scientific algorithms, such as in line-of-sight and convex-hull algorithms in computational geometry. Often, they are computed with **DO** loops using a serial approach. For instance, the example above could be expressed:

```
B(1) = A(1)
DO I=2,5
   B(I) = B(I-1) + A(I)
END DO
```

Despite the serial algorithm used in scanning with **DO**, scans are not inherently serial operations: they are computed efficiently by the CM using the global communication network. To express a scan as a parallel operation, use **FORALL** to specify the set of subarrays over which the combiner is applied and specify the combiner with a reduction function. The destination array can be any array that is conformable with the source array, including the source array itself. (However, recall the restriction that **FORALL** statements that reference a transformational intrinsic function are computed serially in Version 1.0.)

For example, to express an add-scan with **FORALL**:

```
FORALL (I=1:5) B(I) = SUM( A(1:I) )
```



The reduction function **SUM** returns, for each value of **I**, the scalar sum of the values up to and including **A(I)**, which is then assigned to **B(I)**.

To reverse the direction of the scan, so that each element gets the sum of itself and those following, and the first element gets the cumulative sum:

```
FORALL (I=1:N)  B(I) = SUM( A( N-I+1 : N ) )
```

The scan operation is easily extended to a multidimensional array, although only one dimension can be scanned at a time:

```
FORALL (I=1:N)  B(I,:) = SUM( A(1:I,:) )
```

The combiner in a scan operation can be any operation for which CM Fortran provides a reduction function: **SUM**, **PRODUCT**, **MAXVAL**, **MINVAL**, **ANY**, **ALL**, or **COUNT**. For example, the following code uses a multiply-scan to compute factorials:

```
DIMENSION FACT(N)
FACT = [1:N]

FORALL (I=1:N) FACT(I) = PRODUCT( FACT(1:I) )
```

NOTE

For cases, such as scan operations, where **FORALL** is temporarily restricted to execute serially, CM Fortran provides utility routines that generate the appropriate parallel instructions. These utility routines are described in the *CM Fortran User's Guide.*

Part III
Optimization

Chapter 9

Optimizing CM Array Layout

To achieve the best performance on the Connection Machine system, an application program must maximize processor use and streamline or eliminate interprocessor communication. Simply stated, this means:

- Use as many CM processors as possible in each operation. To the extent that a program leaves processors inactive, it reduces the advantages of parallel processing.
- Avoid interprocessor communication wherever possible. Operations within processors are faster than operations between processors.
- When communication is necessary, use the more efficient communication mechanisms and paths.

When the CM Fortran compiler allocates an array on the CM, it does so in a canonical layout that achieves these goals for a variety of array uses. For some uses, however, a different layout would yield greater efficiency. For these cases, CM Fortran provides two compiler directives, `LAYOUT` and `ALIGN`, which enable the programmer to call for the optimal CM array layout given the intended use of the arrays:

- `LAYOUT` causes an array to be laid out in CM memory in a way that either reduces interprocessor communication or optimizes the speed of communication along specified dimensions.
- `ALIGN` causes specified sections of two nonconformable arrays to be "aligned" in the same virtual processors, so that elemental operations on their corresponding elements do not require interprocessor communication.

This chapter introduces the directives `LAYOUT` and `ALIGN` and offers some guidelines for their use.

9.1 Specifying Directives

Compiler directives appear in the specification part of a main program, subroutine, or function, typically after the declaration of the arrays to which they apply. The general form of the directives is:

```
CMF$  ALIGN  arguments
CMF$  LAYOUT arguments
```

The `C` of `CMF$` must appear in column 1, and blanks must separate the items. Other than in column 1, column position is unimportant. See the *CM Fortran Reference Manual* for the complete syntax, including continuation lines and comments, of compiler directives.

9.1.1 Scope of Directives

An `ALIGN` or `LAYOUT` directive affects an array only within the program unit in which the directive appears. If an array is used in more than one program unit, the directive must be repeated in all the program units (possibly by means of an `INCLUDE` line). Inconsistent use of compiler directives across program units results in run-time errors or incorrect results.

9.1.2 Noncanonical Arrays as Arguments

If a directive changes the layout of an array—which is normally the intention—that array is referred to as a *noncanonical* array. When such an array is passed as an argument to a procedure, the dummy array in the procedure must have the same noncanonical layout as the actual argument. Like mismatched shape or home, mismatched array layout across procedure boundaries causes a run-time error or incorrect results.

The CM Fortran compiler can catch this error if you provide an interface block whenever you pass a noncanonical array as an argument. (Interface blocks are described in Chapter 5 and in the *CM Fortran Reference Manual*.) Thus, the directive line that specifies the array layout should appear in three places: in the specification part of the calling routine, in the specification part of the procedure, and in the interface block that duplicates the procedure interface within the calling routine.

9.2 The Virtual Machine Model

The discussion of elemental array operations in Chapter 2 mentioned that an array object is allocated in CM memory in a set of virtual processors that has been configured to reflect the shape of the array. Many different processor configurations, called *VP sets*, can coexist during program execution to accommodate array objects of different shapes.

Normally, the CM Fortran compiler creates VP sets and maps CM Fortran arrays onto them without any intervention from the programmer. It is worthwhile, however, to digress briefly into the nature of VP sets and the canonical array layout in order to understand how the system seeks to optimize processor use and communication.

Against this background, the later sections explore the ways the programmer can alter the canonical array layout by means of the compiler directives **LAYOUT** and **ALIGN**, and how these changes can improve the performance of the application program.

9.2.1 Two Execution Models

When compiling a CM Fortran program, the programmer chooses one of two execution models, Paris or slicewise (see Section 2.5). The execution models differ in the way the compiler maps CM arrays onto the underlying hardware. The Paris model can execute on any CM hardware configuration; the slicewise model requires a CM with the optional 64-bit floating-point accelerator.

This chapter does not fully explore the differences between the Paris and slicewise virtual machine models. (See the *CM Fortran Optimization Notes* for each of the models for this information.) Instead, this chapter focuses on the significance of the directives **LAYOUT** and **ALIGN**. For this purpose, the similarities between the two models are often more important than their differences.

All CMs are organized into *processing nodes* that consist of 32 bit-serial processors and other associated hardware. Systems equipped with a floating-point accelerator have one floating-point chip per node—that is, one chip for each 32 bit-serial processors. Programs compiled for the Paris model use the bit-serial CM processors as the basic physical processing-plus-memory units. Programs compiled for the slicewise model use the processing nodes as the basic physical units.

We introduce the term *processing element* (PE) to refer to the basic unit of either model. Thus, a 64K CM used in the Paris model has 64K PEs, but when used in the slicewise model, it has 64K / 32, or 2K, PEs.

9.2.2 Virtual Processing

The virtual processing mechanism enables a CM to simulate an arbitrarily large number of PEs by allocating more than one virtual processor per (physical) PE. Each PE executes its instructions as many times as there are virtual processors assigned to it, a process called *VP looping*. If the ratio of virtual to physical is 4, for instance, the PE loops over four instances, or *banks*, in its memory for each instruction.

Each set of virtual processors stores its data in the corresponding memory locations in *all* the PEs. Thus, a VP set cannot be smaller than the physical machine or machine section that executes the program. Moreover, all the PEs have exactly the same number of VPs from any given VP set. This number is the *VP ratio*, which is constant across the machine for any given VP set but which can vary from one VP set to another.

For example, suppose you have a VP set of 65,536 (64K) virtual processors. When the program containing this VP set executes in the Paris model on a 64K CM, the VP ratio is 1 and thus one bank of memory is allocated in each bit-serial processor (PE). When the same program runs on a 32K CM, two banks of memory are allocated in each PE for that VP set, and the PE executes each instruction twice (once for each instance of memory).

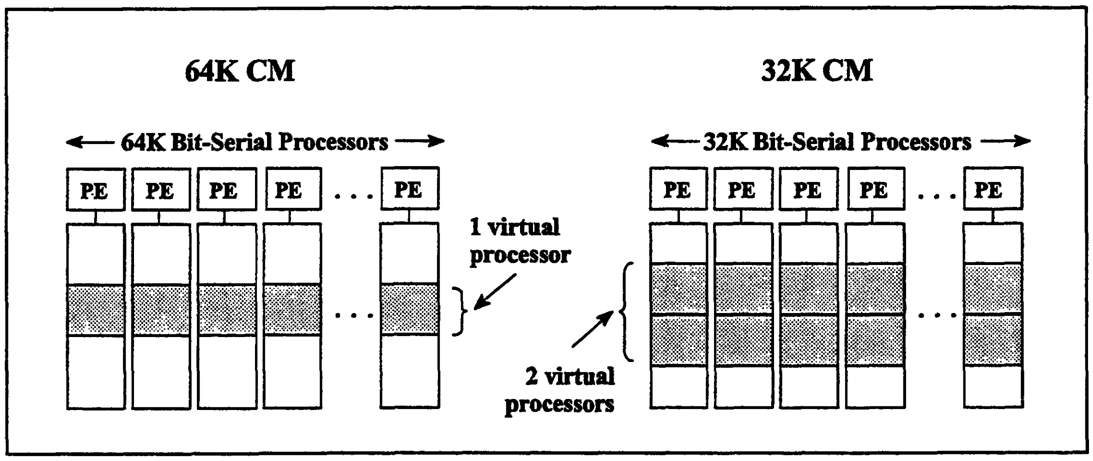


Figure 8. Memory banks allocated (VP ratio) in a 64K VP set: Paris model

In the slicewise model, the principle is the same, but the number of PEs, and thus the VP ratio, is different. Since a 64K CM has 2K processing nodes (PEs), the VP ratio for the 64K VP set is 32. For the same size VP set in a program executing on a 32K CM, the VP ratio is 64.

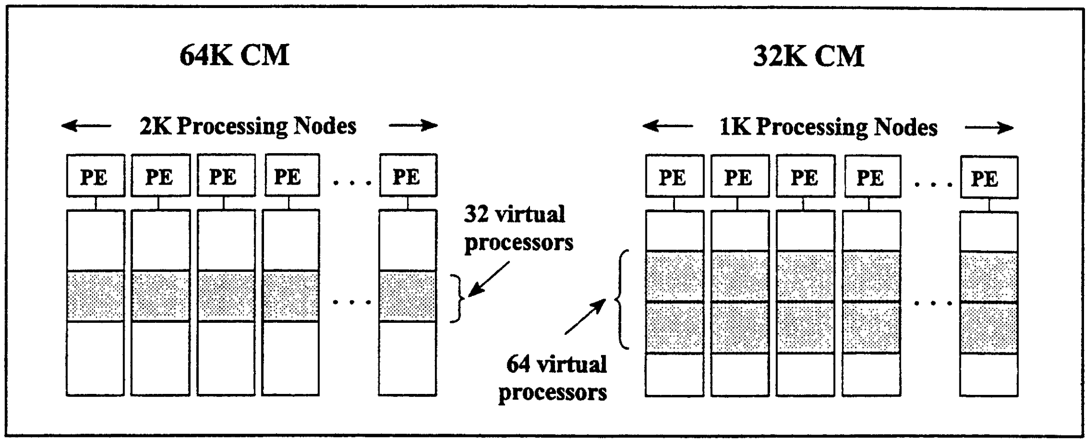


Figure 9. Memory banks allocated (VP ratio) in a 64K VP set: slicewise model

Since the PE loops over its virtual processors (including inactive processors) sequentially, the total execution time for an instruction increases with the VP ratio. The system maximizes PE use and minimizes VP looping by spreading VPs out across the whole physical machine—that is, by keeping the VP ratio the lowest possible that accommodates the data.

NOTE

The comparison of execution times for various VP ratios is valid *only* within an execution model, not across execution models. Because of the special optimizations of the slicewise model, it is likely that the above 64K VP set is processed in less total time under slicewise at VPR=32 than it is under Paris at VPR=1.

9.2.3 VP Geometries

The VPs in a VP set are logically configured into an *n*-dimensional rectangular grid. A VP set created by the CM Fortran compiler *accommodates* the shape of a Fortran array. This is to say that the VP set is at least the size and rank of the array.

The VP set may, however, be larger than the array because the exact size and shape—that is, the *geometry*—of a VP set must also meet some constraints set by the execution model. The details are:

- Under Paris, the axes of the VP grid must all be powers of 2 in length. It follows from this constraint, by the way, that VP ratios (total grid size divided by physical processors) are also powers of 2.
- Under slicewise, only the total size of the VP grid is constrained: it must be a multiple of 4 times the number of processing nodes, reflecting the fact that the 64-bit floating-point chips have a vector length of 4. The axes may be any set of lengths whose product is a legal total size.

The shape of a VP set does not influence the number of instances or banks of physical memory allocated for it. The 64K VP set shown in Figure 8 and Figure 9 could have been defined as 64K processors on one axis, or as 8K x 8, or as 64 x 32 x 4 x 8. The number of banks of memory (the VP ratio) is determined entirely by the total size of the VP set in relation to the number of PEs executing the program.

Axis Ordering

More relevant to physical memory layout is the *ordering* of VPs along a grid axis to suit the requirement that VPs with adjoining addresses be physically connected. In grid-based ordering, called *NEWS ordering,* VPs whose grid indices differ by 1 have a direct communication link. Thus, processor *i* is connected to processor *i*+1, which in turn is connected to processor *i*+2. This physical connection provides hardware support for grid-based communication—nearest-neighbor accesses, dimensional shifts, and cumulative computation along grid axes.

NOTE

Another processor ordering, called *send ordering*, reflects a second set of VP addresses that is independent of grid position. (Send-ordered VPs are connected if their send addresses differ by one bit.) Send ordering is used only in a few special cases, and is not covered in this manual. See *CM Fortran Reference Manual.*

Axis Weight

Grid axes in multidimensional VP sets can be assigned relative weights, which give the system even further information for selecting a particular virtual-to-physical mapping. VPs along the highest-weighted axis are given the fastest communication links, the next-highest weight gets the next-fastest connections, and so on. The effect of relative weights is to optimize the speed of communication along the higher-weighted axes at the expense of the lower-weighted ones.

9.3 Canonical Array Layout

This section describes the canonical layout of CM Fortran arrays—that is, the way the compiler lays out arrays in CM memory in the absence of directives. The following sections show how compiler directives can alter layout to optimize certain intended uses of arrays.

When an array is canonically allocated in CM memory:

- The compiler creates a VP set whose size and shape accommodates the array.

  - Under Paris, if the length of an array dimension is not a power of 2, the corresponding grid axis is the next higher power of 2. If the VP set is still smaller than the number of CM processors executing the program, extra virtual processors are added to the VP set along a hidden *(zeroth)* axis.

  - Under slicewise, if the array is not a legal size for a VP set (a multiple of 4 times the number of processing nodes), the compiler "pads" the VP set with extra processors until it reaches the next higher legal size. No hidden axis is added; the extra VPs are added to one or more of the existing axes.

  In general, slicewise VP sets are padded less often and less heavily than are Paris VP sets. The goal of the slicewise virtual-to-physical mapping is to minimize the amount of memory used, and the constraints on VP-set geometry are less stringent.

- The compiler allocates the array one element per virtual processor. The array element at the lower bounds of the dimensions, for instance, array element (1,1,...), is placed in processor (0,0,...).

  The "extra" processors that pad the VP set up to the next legal size or geometry are deactivated (or ignored) during operations on the array.

- All the grid axes are NEWS-ordered, and all are assigned the weight 1. Thus, the system uses the hardware to optimize grid communication, but none of the array dimensions is favored over the others.

  The zeroth axis created under Paris is given a lower relative weight (0), so that it is less favored for communication than the axes that contain array elements. The slicewise model has no axis 0.

When more than one array is canonically allocated in CM memory:

- The compiler places arrays of the same shape in the same VP set and places their corresponding elements in the same VP. Elemental operations between the arrays require no interprocessor communication, and dimensional shifts on sections taken from them can often be performed with NEWS communication.

- The compiler places arrays of different shapes in different VP sets and thus places their respective elements in different VPs. Most operations across VP sets, including elemental operations on conformable sections, require general (router) communication to copy an array into a temporary location before the operation can be performed.

The canonical layout optimizes performance in the same way that the virtual processor mechanism does: by spreading array elements across the whole physical machine (or machine section) to maximize PE use and minimize VP looping. In addition, the canonical layout uses the underlying hardware to optimize grid communications, but all the array dimensions are weighted equally.

Notice, however, that an array too small to fill the physical machine nevertheless ties up memory across the whole machine. As a result, some number of PEs are left idle during operations on the array. (An array fails to fill the machine under Paris if it is smaller than the number of CM processors; under slicewise, an array needs to be 4 times the number of processing nodes to fill the machine.) If the algorithm permits, small arrays are often better handled on the front end.

9.4 The LAYOUT Directive

The LAYOUT directive allows the user to specify the axis ordering and weights of the VP set in which an array is to be allocated. CM Fortran itself provides another possible ordering, *serial ordering*, which calls for very different features in the underlying VP set than does the canonical layout.

While parallel (NEWS-ordered) dimensions are allocated across VPs, one element per VP, serial dimensions are allocated within the memory of the VPs. The subscript values in a serial dimension reference element data at different *offsets* into VP memory, rather than in different VPs.

For example, consider a 4 x 6 array. In the canonical layout, where both dimensions are parallel, the array is laid out across 24 virtual processors, with one array element in the memory of each processor. When the first dimension is serial, however, the array is laid out across 6 virtual processors, with one *column* in the memory of each processor. Figure 10 indicates these alternative layouts by highlighting the memory of one virtual processor.

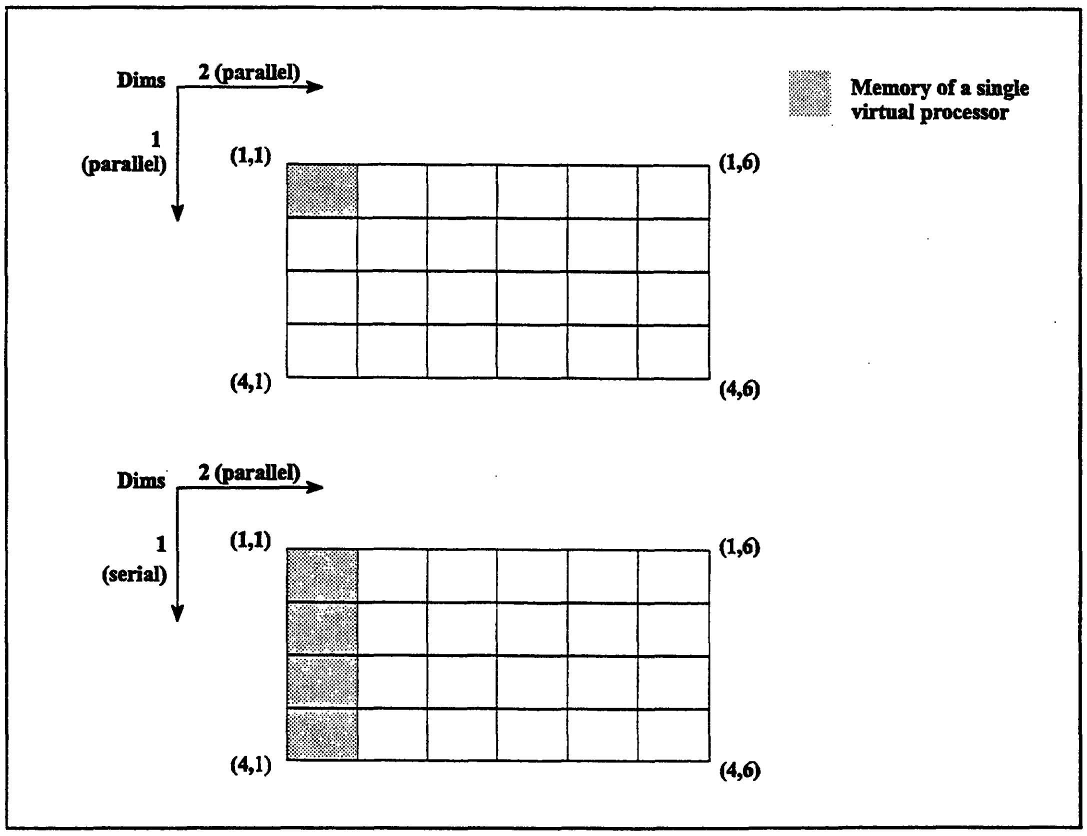


Figure 10. Layouts of a 4 x 6 array with first dimension parallel and first dimension serial

For simplicity, this figure and subsequent figures do not show the actual size of the VP set in which the array is allocated. Small arrays like the ones shown would be heavily padded in both Paris and slicewise models to meet the minimum legal sizes for VP sets.

Declaring an array dimension as serial makes no difference in how you reference it or in the array operations you can perform on it. For instance, to manipulate the elements on the serial dimension of the 4 x 6 array shown above:

```
          A(1,:) = A(2,:) + A(3,:)**2 + SIN( A(4,:) )
```

The major difference between this array and a canonical array is that these operations occur entirely within each of the VPs instead of across VPs, with a consequent increase in speed. Each PE need only index within its own memory to locate or move the element of interest.

A second difference between this array and a canonical array is in the way sections are passed as arguments. Recall from Section 5.4.1 that array sections are normally copied to a temporary location before being passed as arguments to a procedure. However, a section specified with a scalar subscript for a serial dimension—such as **A(2,:)**—is passed in place. Passing a section in place is of course faster than first copying it to another location.

9.4.1 Syntax

Only one **LAYOUT** directive can be applied to an array in a program, and the directive must be repeated in every program unit where that array is used. The directive has the form:

CMF$ **LAYOUT** *array-name* **(** *axis-1-spec, axis-2-spec*, . . . **)**

Each array dimension must have exactly one *axis-spec*, which specifies the ordering and weight of the dimension. The ordering can be either **:SERIAL** or **:NEWS**, and each **:NEWS** keyword can be preceded by a literal or named integer constant that indicates the relative weight. For best performance, all serial dimensions should be specified to the left of the first parallel dimension.

For example:

```
      DIMENSION A( 100,100,100 )
CMF$  LAYOUT A( :SERIAL, 2:NEWS, :NEWS )
```

This directive specifies that array **A** is to be laid out with one serial dimension and two NEWS-ordered dimensions, and that dimension 2, with a weight of 2, is to be favored for interprocessor communication over dimension 3, which gets the default weight 1.

Since NEWS is the default axis ordering, the keyword may be omitted and just a weight supplied; or, both ordering and weight may be omitted as long as placeholder commas remain. Thus, the following directive is equivalent to that above:

```
      DIMENSION A( 100,100,100 )
CMF$  LAYOUT    A( :SERIAL, 2, )
```

Increasing the relative dimension weight does not guarantee a proportional increase in communication speed for a dimension, but it does help the system in selecting from among the many possible ways of mapping virtual processors to hardware.

9.4.2 Array Homes

A **LAYOUT** directive determines the home of the array in the program unit. In the absence of a directive, an array's home is determined by how it is used (as shown in Chapter 2). However, a **LAYOUT** directive that specifies at least one parallel dimension causes the array to be allocated on the CM regardless of how the array is referenced in the program unit.

It is sometimes useful, particularly for arrays that will be passed as arguments to external procedures, to supply a **LAYOUT** directive simply to control the home of an array without necessarily altering its layout. For example, this directive, which specifies the default CM layout, guarantees that array **B** will be allocated on the CM:

```
      DIMENSION B( 1000,1024 )
CMF$  LAYOUT    B( :NEWS, :NEWS )
```

Conversely, this directive

```
      DIMENSION C( 1000,1024 )
CMF$  LAYOUT    C( :SERIAL, :SERIAL )
```

guarantees that array **C** will be allocated entirely in front-end memory, laid out in the normal linear, column-major fashion. Such an array may not be used in a Fortran 90 array operation. (Unlike a CM array, an array with all serial dimensions is not padded to reach some minimum size or geometry.)

When you use the **LAYOUT** directive to control array homes, avoid using the compiler switch **-nodirective**, which disables all directives in a program. By disabling **LAYOUT** directives, this switch can change array homes and thus cause a valid program to fail.

9.4.3 Serial Ordering

In general, you declare an array dimension serial when the operations you intend to perform on it are inherently serial.

There are two different, but equally valid, ways to view an array with mixed parallel and serial dimensions: as a collection of *local* (per-virtual-processor) arrays or as a collection of parallel arrays. The layout and the means of referencing the array are identical in the two views. The views reflect differences in the intended use of the array and therefore in the particular way that the layout enhances program performance.

Per-Processor Arrays

One way to view an array with mixed parallel and serial dimensions is as a parallel array of per-processor arrays. Each per-processor array is stored in its own virtual processor as a linear array of elements. This is the view depicted above in Figure 10.

In this view, the array represents a collection of subarrays, or *local* arrays, that will all be subject to the same operations—the basic rationale for data parallel processing. The parallel dimension provides the parallelism—the replication of the operation—while the serial dimension optimizes the speed of each instance of the operation by eliminating interprocessor communication. The operation shown at the beginning of this section illustrates the benefit of using serial dimensions in this way.

```
          A(1,:) = A(2,:) + A(3,:)**2 + SIN( A(4,:) )
```

Either the parallel array or the local arrays can be multidimensional. For example, Figure 11 shows an array with one serial dimension and two parallel dimensions. This array can be viewed as a matrix of local vectors, where each local vector resides in the memory of a unique virtual processor. This array would be declared as:

```
          DIMENSION   A( 4,6,4 )
    CMF$  LAYOUT      A( :SERIAL, :NEWS, :NEWS )
```

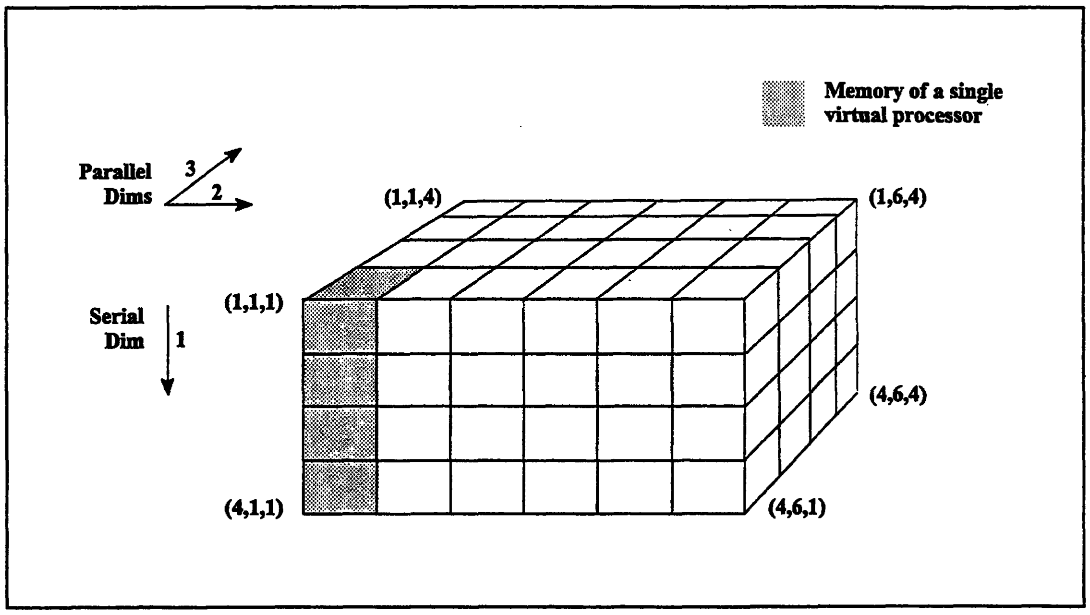


Figure 11. A 4 x 6 x 4 array with first dimension serial, viewed as local vectors

Figure 12 shows the layout of an array with two serial dimensions and one parallel dimension, which can be viewed as a vector of local matrices. This array would be declared as:

```
      DIMENSION   B( 3,2,4 )
CMF$  LAYOUT      B( :SERIAL, :SERIAL, :NEWS )
```

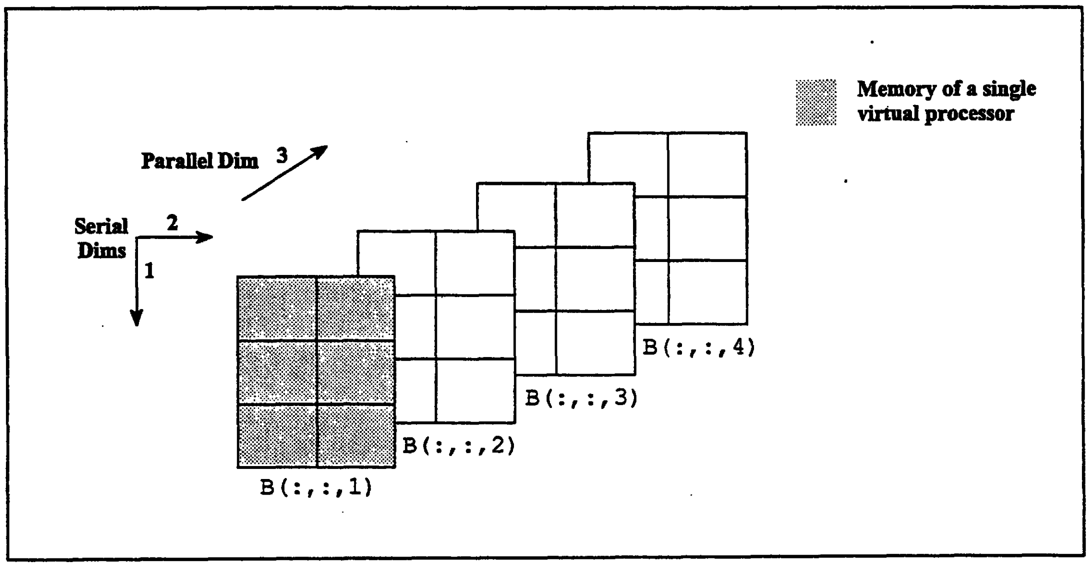


Figure 12. A 3 x 2 x 4 array with first two dimensions serial, viewed as local matrices

In all these layouts, operations within the local arrays require no interprocessor communication, as they would in the canonical array layout. This increases the speed of operations such as the following one, which averages the four lower elements of array **B** (in Figure 12) and places the result in the upper left corner:

```
B(1,1,:) = ( B(2,1,:) + B(3,1,:) + B(2,2,:) + B(3,2,:) ) / 4
```

NOTE

When referring to a section of an array with serial dimensions, programs get best performance when any scalar subscripts to a serial dimension are to the left of all Fortran 90–style subscripts to any dimension, serial or parallel. For example, given array **B** above, an operation on `B(1,:,:)` or on `B(1,1,:)` is faster than an operation on `B(:,1,:)` or on `B(:,:,1)`.

Serial dimensions extract no trade-off for the increased speed as long as the parallel part of the array—that is, the product of the parallel dimensions—is large enough to reach the minimum legal size for a VP set. If the parallel array dimensions do not reach legal minimum VP-set size, the system must allocate—and then deactivate—the extra VPs. The result is idle PEs and thus reduced performance.

The problem of unused resources is more acute under Paris than under slicewise, because of the somewhat greater likelihood under Paris that an array with serial dimensions might not fill the physical machine. For example, imagine that you have a 4 x 8K array in a program executing under Paris on a 32K CM. In the canonical layout, this array exactly fills the machine and no resources are wasted. However, if the first dimension is serial, the parallel portion of the array is only 8K and thus fills only one-quarter of the machine. The compiler allocates memory in the additional virtual processors on the zeroth axis, but these processors are never used.

Notice that no resources are wasted when this array is allocated under the slicewise model. Since a 32K CM has 1K PEs, the 8K parallel portion of the array fills the machine at a VP ratio of 8, twice the legal minimum size.

Sets of Parallel Arrays

A second way to view an array with mixed parallel and serial dimensions is as an indexed collection of parallel arrays. Such an array can be seen as a serial array of parallel sections, perhaps as time slices in the evolution of a system. Figure 7 illustrates this view of a 3-dimensional array that is serial in its first dimension.

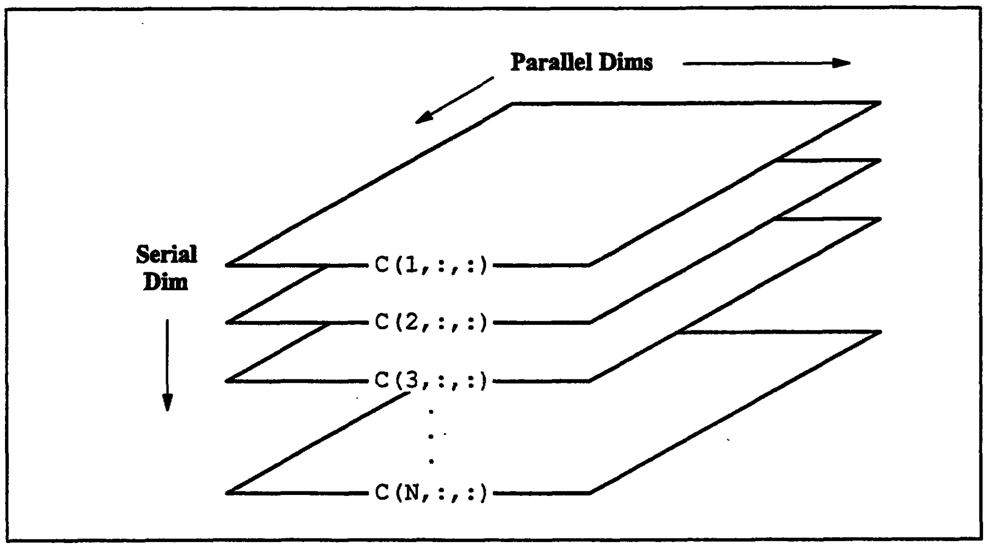


Figure 13. A 3-dimensional array with first dimension serial, viewed as parallel sections

The array in Figure 7 might have been declared as follows:

```
      DIMENSION   C( N,M,P )
CMF$  LAYOUT      C( :SERIAL, :NEWS, :NEWS )
```

Similarly, Figure 8 illustrates this viewpoint in relation to a 4-dimensional array that is serial in its first two dimensions. This array might be declared as:

```
      DIMENSION   D( N,M,P,Q )
CMF$  LAYOUT      D( :SERIAL, :SERIAL, :NEWS, :NEWS )
```

These arrays are declared and laid out in exactly the same way as are per-processor (local) arrays shown in Figure 11 and Figure 12. What distinguishes the two viewpoints is the intended use of the arrays: instead of parallel operations on local arrays, this use of serial dimensions is to operate on one or a few "slices" or parallel planes of the array at a time.

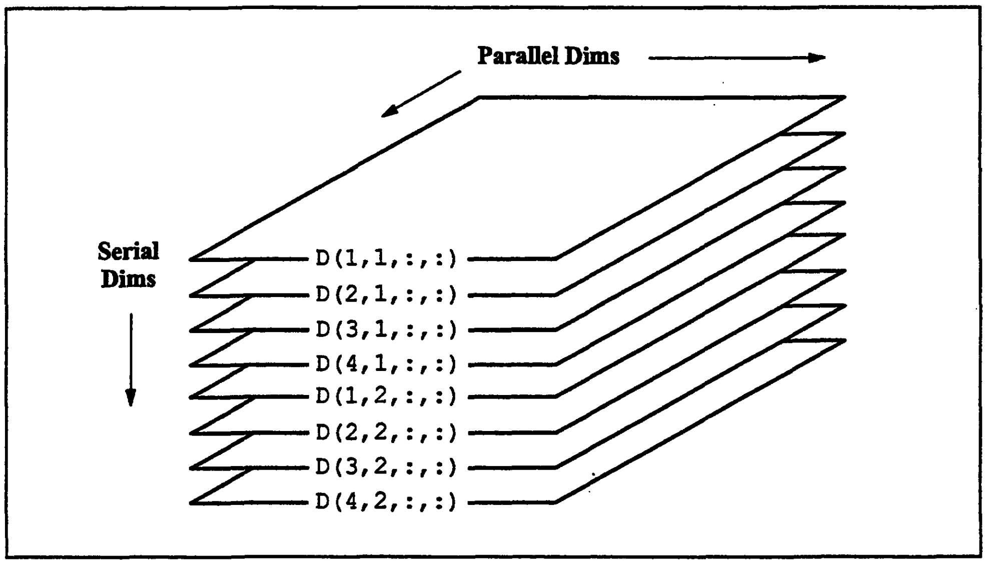


Figure 14. A 4-dimensional array with first two dimensions serial, viewed as parallel sections

For example, to compute the sum of all the elements on the first plane of the 3-dimensional array in Figure 7:

```
      TOP_SUM = SUM( C(1,:,:) )
```

Similarly, to increment the values in the third slice of the 4-dimensional array in Figure 8:

```
      D(3,1,:,:) = D(3,1,:,:) + 1
```

The purpose of declaring serial dimensions, in this view, is to minimize VP looping when manipulating a parallel section of the array. Recall from Chapter 4 that selecting a section of an array deactivates the processors that contain the non-selected elements. If the above arrays were allocated canonically, the physical processors would need to loop over the VPs for all the planes, if only to discover that most of them are inactive. With the left-most dimensions laid out within VPs, however, the PE need only access the location of the desired parallel section, simply ignoring the other locations. Thus, no VP looping is required to operate on any one parallel section of the array and execution time is greatly reduced.

9.4.4 Aligning Multiple Arrays

Arrays of different shapes are normally allocated in different VP sets. However, an array with mixed serial and parallel dimensions is allocated in the same VP set as another array with the same *parallel* dimensions (assuming equal axis weights). Programs can take advantage of this fact to eliminate interprocessor communication in operations on non-conformable arrays that are frequently used together.

For example, consider two arrays declared as follows:

```
      DIMENSION A( 10, 256, 256 )
      DIMENSION B( 256, 256 )
CMF$  LAYOUT    A( :SERIAL, :NEWS, :NEWS )
CMF$  LAYOUT    B( :NEWS, :NEWS )
```

Because of the **LAYOUT** directive, the compiler places these two arrays in the same 256 x 256 VP set. Each of the VPs contains the ten serial elements of array **A** *and* one element of array **B**. As a result, an operation that involves the parallel planes of array **A** and the corresponding elements of array **B** requires no interprocessor communication. For example:

```
   B = A(3,:,:)          ! No interprocessor communication
```

or,

```
   DO (I = 1,10)
      A(I,:,:) = B + I   ! No interprocessor communication
   END DO
```

On the other hand, an array that is the same shape as **A** but canonically laid out (or subject to a different **LAYOUT** directive) is in a different VP set from **A**. In this case, operations on the corresponding elements of the two arrays take place across VP sets, with a consequent increase in execution time.

For example, because the arrays **A** and **C** have different axis orderings, the assignment statement below involves interprocessor communication:

```
      DIMENSION A( 10, 256, 256 )
      DIMENSION C( 10, 256, 256 )
CMF$  LAYOUT    A( :SERIAL, :NEWS, :NEWS )
CMF$  LAYOUT    C( :NEWS, :NEWS, :NEWS )
      . . .
      A = C                 ! Interprocessor communication
```

9.5 The ALIGN Directive

The **ALIGN** directive describes the layout of an array in terms of specified axes of another array whose layout is already determined. This directive causes the elements of the source array to be placed in the same virtual processors as certain sections of the target array.

The **ALIGN** directive can speed up an application program by eliminating unnecessary communication of data when operations are performed on multiple arrays that would not normally be aligned. For example, if a vector is used repeatedly in array operations with rows of a matrix, it may be worthwhile to align the vector with one of the rows of the matrix.

9.5.1 Syntax

Like the **LAYOUT** directive, the **ALIGN** directive can be applied only once to an array in a program, and it must be repeated in every program unit where that array is used. Its format is:

CMF$ **ALIGN** *source-array* **(** *axis-specs* **)** **WITH** *target-array* **(** *axis-specs* **)**

The *axis-specs* of the source array assign a symbolic name to each of its dimensions. The same symbolic names are then used in the *axis-specs* of the target array to indicate the pairs of source-and-target dimensions that are to be aligned in the same VPs. Any target dimensions that are not related to the source array are identified with a scalar index value (or offset).

For example, to align a 5-element vector **V** with the first row of the 5 x 5 matrix **A**:

```
      DIMENSION V(5), A(5,5)
CMF$  ALIGN V(I) WITH A(1,I)
```

The effect is to place **V** in the same VP set as **A**, in the same VPs as **A**'s first row:

```
VVVVV         AAAAA
.....         AAAAA
.....         AAAAA
.....         AAAAA
.....         AAAAA
```

Similarly, to align **V** with the last row of **A**:

```
      CMF$  ALIGN V(I) WITH A(5,I)
```

as illustrated by:

```
            .....          AAAAA
            .....          AAAAA
            .....          AAAAA
            .....          AAAAA
            VVVVV          AAAAA
```

To align **V** with the first column of **A**:

```
      CMF$  ALIGN V(I) WITH A(I,1)
```

as illustrated by:

```
            V....          AAAAA
            V....          AAAAA
            V....          AAAAA
            V....          AAAAA
            V....          AAAAA
```

Multiple Source Arrays

More than one source array can be aligned (in separate directive lines) with a single target. For example, to align **V** and another 5-element vector **Q** with different sections of **A**:

```
      CMF$  ALIGN V(I) WITH A(I,1)
      CMF$  ALIGN Q(I) WITH A(I,3)
```

as illustrated by:

```
            V.Q..          AAAAA
            V.Q..          AAAAA
            V.Q..          AAAAA
            V.Q..          AAAAA
            V.Q..          AAAAA
```

No interprocessor communication is required when **V** is used together with the first column of **A** or when **Q** is used together with the third column of **A**. However, **V** and **Q**, although in the same VP set, are in different VPs, and operations on the two vectors therefore do require interprocessor communication.

Multidimensional Source Arrays

The source array can of course be multidimensional. The target array must be of the same or higher rank. For example, to align a 2-dimensional array with the first two dimensions of a target 3-dimensional array on the target's third "row":

```
      DIMENSION B(5,5), C(5,5,10)
CMF$  ALIGN B(I,J) WITH C(I,J,3)
```

The **ALIGN** directive does not support permutation of array dimensions. The dimension names in the source axis-specs must appear in the same order in the target axis-specs (although the target's axis-specs may intersperse dimension subscripts among the symbolic names).

```
      DIMENSION B(5,5), D(5,5)
CMF$  ALIGN B(I,J) WITH D(J,I)    [ not supported ]
```

Aligning Different-Sized Dimensions

A source array dimension must fit entirely within the corresponding dimension of the target array. Unless otherwise specified, the lower bound of the source dimension is aligned with the lower bound of the target dimension. For example, the directive

```
      DIMENSION  R(5), Q(20)
CMF$  ALIGN R(I) WITH Q(I)
```

aligns the lower bounds of **R** and **Q**:

```
RRRRRR...............
QQQQQQQQQQQQQQQQQQQQQ
```

You can supply a (positive) offset in the target's axis-spec to indicate another placement. For example, to align **R** with the section **Q(6:10)**, which is offset by five elements from **Q**'s lower bound:

```
      CMF$  ALIGN R(I) WITH Q(I+5)
```

as illustrated by:

```
            .....RRRRRR..........

            QQQQQQQQQQQQQQQQQQQQQ
```

Extending this example to a 2-dimensional target array is straightforward. For example,

```
            DIMENSION V(5), B(10,10)
      CMF$  ALIGN V(I) WITH B(2,I+3)
```

causes **V** to be allocated in the same VPs that contain the section **B(2,4:8)**:

```
            ..........  BBBBBBBBBB
            ...VVVVV..  BBBBBBBBBB
            ..........  BBBBBBBBBB
            ..........  BBBBBBBBBB
            ..........  BBBBBBBBBB
            ..........  BBBBBBBBBB
            ..........  BBBBBBBBBB
            ..........  BBBBBBBBBB
            ..........  BBBBBBBBBB
            ..........  BBBBBBBBBB
```

Similarly, for a 2-dimensional source array,

```
            DIMENSION A(5,5), B(10,10)
      CMF$  ALIGN A(I,J) WITH B(I+5,J+2)
```

causes **A** to be allocated in the same VPs that contain **B(6:10,3:7)**, as illustrated by:

```
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..AAAAA...  BBBBBBBBBB
..AAAAA...  BBBBBBBBBB
..AAAAA...  BBBBBBBBBB
..AAAAA...  BBBBBBBBBB
..AAAAA...  BBBBBBBBBB
```

9.5.2 Benefits and Costs

The **ALIGN** directive can greatly enhance program performance when two arrays that are not conformable are often used together. By aligning the two arrays in the same virtual processors, the program uses elemental operations instead of router communication to operate on corresponding elements of the two arrays.

There are, however, certain potential trade-offs that you should consider when you evaluate the benefits of using the **ALIGN** directive in an application program. These trade-offs are greater memory use, more VP looping, and interprocessor communication in cases where it would not normally be needed.

Memory Use

A potential penalty for the increase in speed that **ALIGN** gives is greater memory use. The layout of the source, or "aligned," array is determined by the target array. In cases where the source array is much smaller or of lower rank than the target array, a certain amount of the memory allocated for it in their shared VP set is never used. If the target array has serial dimensions, the potential for wasted memory is larger.

To pick an extreme example, imagine aligning a vector with the serial dimension of a matrix:

```
      DIMENSION V(4), A( 4, 32768 )
CMF$  LAYOUT A( :SERIAL, :NEWS )
CMF$  ALIGN V(I) WITH A(I,1)
```

Array **A** requires a VP set of size 32K, with the 4 elements of its first dimension allocated within each VP. The 4-element vector **V** requires the same amount of memory, since it is aligned with **A**'s serial dimension. If **V** were not aligned with **A**, it would be spread across

4 VPs in a VP set of 8K x 4. That is, **V** would use up only one-fourth as much memory if it were not aligned in this way with **A**.

VP Looping

The source array shares the VP ratio of the target array. If VP looping is required to cover all the VPs allocated for the target array, the same amount of VP looping is required for the source array, even when it is used alone. For example:

```
      DIMENSION C(128), D(128, 512)
CMF$  LAYOUT D( :NEWS, :NEWS )
CMF$  ALIGN  C(I) WITH D(I,1)
```

Array **D** requires a VP set of 64K. If the program is executed under the Paris model on a 32K machine, each physical processor loops over two VPs; on a 16K machine, it loops four times. (The VP looping is greater under the slicewise model, although the total execution time is likely to be less.) The same amount of VP looping, and its consequent execution time, is required for array **C** in the following operation:

```
      C = 10
```

Array **C**, if canonically laid out under the Paris model, would fit in any size CM with a VP ratio of 1. When aligned with array **D**, however, its VP ratio is higher than 1 for any size CM under 64K. Although the unused processors are deactivated for the assignment statement above, the PEs always loop over all the VPs assigned to them (at least to determine which ones are active).

Communication Costs

An aligned array may no longer be in the same VP set as other arrays of its own shape. Array **C** in the above example is conformable with, but not in the same VP set as, another 128-element vector that is canonically allocated.

```
      DIMENSION C(128), D(128, 512), E(128)
CMF$  LAYOUT D( :NEWS, :NEWS )
CMF$  ALIGN  C(I) WITH D(I,1)
```

If you use vectors **C** and **E** together, the operation will require interprocessor communication and thus greater execution time than if **C** had been allocated canonically.

Appendix

Appendix A

Sample Programs

This appendix presents several programming examples in CM Fortran (V0.7 or later).

- A histogram program
- A convolution program
- A prime number sieve
- A solver for Laplace's equation

For comparison, several of these programs are presented in Fortran 77 versions as well.

A.1 Histogram

This subroutine for computing a histogram is shown in both serial and parallel versions.

A.1.1 Histogram: Fortran 77 Version

```
      SUBROUTINE HISTOGRAM(NPOINTS, POINTS, NBARS,
     $                     LOW, HIGH, BARS, ERR)
      INTEGER  NPOINTS, NBARS, BARS(NBARS), I, J
      REAL     POINTS(NPOINTS), LOW, HIGH, BARWIDTH
      LOGICAL  ERR
```

```
C
C     Initialization
C
      ERR = .FALSE.
      BARWIDTH = (HIGH-LOW)/NBARS
      DO 1 I=1,NBARS
         BARS(I) = 0
  1   CONTINUE
C
C     Iterate over POINTS, updating BARS and checking for errors
C
      DO 2 I=1,NPOINTS
         IF (POINTS(I) .LT. LOW .OR. POINTS(I) .GT. HIGH) THEN
            ERR = .TRUE.
         ELSE
            J = IFIX((POINTS(I)-LOW)/BARWIDTH)+1
            BARS(J) = BARS(J) + 1
         ENDIF
  2   CONTINUE

      RETURN
      END
```

A.1.2 Histogram: CM Fortran Version

```
      SUBROUTINE HISTOGRAM(NPOINTS, POINTS, NBARS,
     $                     LOW, HIGH, BARS, ERR)
      IMPLICIT NONE
      INTEGER NPOINTS, NBARS, BARS(NBARS), BAR, PT
      REAL POINTS(NPOINTS), LOW, HIGH, BARWIDTH
      LOGICAL ERR
      INTEGER TEMP1(NPOINTS, NBARS), TEMP2(NPOINTS, NBARS)
C
C     Initialization
C
      ERR = .FALSE.
      BARWIDTH = (HIGH-LOW)/NBARS
C
```

```
C     First, we fill TEMP1 with values so that TEMP1(PT,BAR) =
C     BAR, then we fill TEMP2 such that TEMP2(PT,BAR) contains
C     the bar number in which POINTS(PT) falls. Finally, we
C     count the "intersections" of TEMP1 and TEMP2 for each bar.
C
      FORALL (BAR=1:NBARS)  TEMP1(:,BAR) = BAR
      FORALL (PT=1:NPOINTS, BAR=1:NBARS)
     $   TEMP2(PT,BAR) = IFIX( (POINTS(PT)-LOW)/BARWIDTH )+1
      BARS = COUNT( TEMP1 .EQ. TEMP2, DIM=1 )
C
C     Check for errors
C
      ERR = ANY(POINTS .LT. LOW .OR. POINTS .GT. HIGH)

      RETURN
      END


C   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      PROGRAM HISTTEST
      INTEGER NPOINTS, NBARS, I, J, K
      PARAMETER (NBARS = 10, NPOINTS = NBARS*(NBARS+1)/2)
      LOGICAL ERR
      INTEGER BARS(NBARS)
      REAL POINTS(NPOINTS)

      BARS = 0
      K = 1

      DO I=1,NBARS
         POINTS(K:K+I-1) = REAL(I)
         K = K + I
      END DO

      ERR = .FALSE.
      CALL HISTOGRAM(NPOINTS, POINTS, NBARS,
     $               0.9, 10.1, BARS, ERR)
      DO I=1,NBARS
         PRINT *,I,BARS(I)
      END DO
      IF (ERR) PRINT *,"An error was detected"
      END
```

A.2 Convolution

This program illustrates 1-dimensional convolution over a 2-dimensional data set (done in column direction). It is shown in both serial and parallel versions.

A.2.1 Convolution: Fortran 77 Version

```
      PROGRAM CONVOLVE
      INTEGER NT,NX,NFP
      PARAMETER (NX=2048)
      PARAMETER (NT=128)
      PARAMETER (NFP=16)
      REAL F(NFP)
      REAL P(NT,NX), Q(NT,NX)
C     Initialize P and Q

      DO I=1,NX
        DO J=1,NT
          P(J,I) = FLOAT(J)
          Q(J,I) = 0.0
        END DO
      END DO

C     Initialize F

      DO I=1,NFP
        F(I) = I
      END DO

C     Shift the array P by one over the first dimension
C     Compute Q after the shift on P

      DO IFP=1,NFP
        DO I=1,NX
          DO J=NT-1,1,-1
            P(J+1,I) = P(J,I)
          END DO
          P(1,I) = 0.0
        END DO
```

```
          DO I=1,NX
            DO J=1,NT
              Q(J,I) = Q(J,I) + P(J,I) * F(IFP)
            END DO
          END DO
        END DO

C       Show results

        PRINT *, ( Q(I,1), I = 1,NT )

        STOP
        END
```

A.2.2 Convolution: CM Fortran Version

```
        PROGRAM CONVOLVE
        IMPLICIT NONE
        INTEGER, PARAMETER :: NX=16, NT=16, NFP=16
        INTEGER I, IFP
        REAL F(NFP)
        REAL, ARRAY(NT,NX) :: P, Q
C
C       Initialize P and Q on the CM.
C
        Q = 0.0
        FORALL(I=1:NT)  P(I,:) = FLOAT(I)
C
C       Initialize F on the front end.
C
        DO IFP = 1, NFP
           F(IFP) = IFP
        END DO
C
C       Shift the array P by one over the first dimension.
C       Compute Q after the shift on P.
C
```

```
      DO IFP = 1, NFP
         P = EOSHIFT(P, DIM=1, SHIFT=-1)
         Q = Q + P * F(IFP)
      END DO
C
C     Show results
C
      PRINT *, ( Q(I,1), I = 1, NT)
      END
```

A.3 Prime Number Sieve

This program for finding the prime numbers in a set of numbers is presented in three versions: a serial version and two alternative parallel versions.

A.3.1 Primes: Fortran 77 Version

```
      PROGRAM FINDPRIMES
      INTEGER I, J, N
      PARAMETER (N = 999)
      LOGICAL PRIMES(N), CANDID(N)
C
C     Initialization
C
      DO 1 I=1,N
         PRIMES(I) = .FALSE.
         CANDID(I) = .TRUE.
 1    CONTINUE
      CANDID(1) = .FALSE.
C
C     Loop: Find next valid candidate, mark it as prime,
C           invalidate all multiples as candidates, repeat.
C
 2    DO 4 I=1,SQRT(REAL(N))
         IF (CANDID(I)) THEN
            PRIMES(I) = .TRUE.
            DO 3 J=I,N,I
               CANDID(J) = .FALSE.
 3          CONTINUE
         ENDIF
 4    CONTINUE
C
C     At this point, all valid candidates are prime
C
      DO 5 I=SQRT(REAL(N))+1,N
         PRIMES(I) = CANDID(I)
 5    CONTINUE
```

```
C
C     Print results
C

      DO I=1,N
         IF (PRIMES(I)) PRINT *,I
      END DO
      END
```

A.3.2 Primes: First CM Fortran Version

This first parallel version is a straightforward translation of the serial program Although it is much faster than the serial program, it is not the fastest possible implementation.

```
      PROGRAM FINDPRIMES
      IMPLICIT NONE
      INTEGER I, N, NEXTPRIME
      PARAMETER (N = 999)
      LOGICAL PRIMES(N), CANDID(N)
C
C     Initialization
C
      PRIMES = .FALSE.
      CANDID = .TRUE.
      CANDID(1) = .FALSE.
C
C     Loop: Find next valid candidate, mark it as prime,
C          invalidate all multiples as candidates, repeat.
C
      NEXTPRIME = 2
      DO WHILE ( NEXTPRIME .LE. SQRT( REAL(N) ) )
         PRIMES( NEXTPRIME ) = .TRUE.
         CANDID( NEXTPRIME:N:NEXTPRIME ) = .FALSE.
         NEXTPRIME = MINVAL( [1:N], DIM=1, MASK=CANDID )
      END DO
C
C     At this point, all valid candidates are prime.
```

```
C
      PRIMES( NEXTPRIME:N ) = CANDID( NEXTPRIME:N )
C
C     Print results
C
      PRINT *, "Number of primes:", COUNT(PRIMES)
      DO I=1, N
         IF (PRIMES(I)) PRINT *, I
      ENDDO
      END
```

A.3.3 Primes: Second CM Fortran Version

This parallel version uses a different approach from the two programs just shown. This program is very fast because it uses more processors than the first parallel version. (The speed advantage is less at higher VP ratios.)

```
      PROGRAM FINDPRIMES
      IMPLICIT NONE
      INTEGER I, N, NN
      PARAMETER (N = 500)
      INTEGER TEMP1(N,N), TEMP2(N,N)
      LOGICAL CANDID(N,N), PRIMES(N)
C
C     Initialization
C
      CANDID = .FALSE.
      FORALL (I=1:N)  TEMP1(I,:) = 2*I+1
      FORALL (I=1:N)  TEMP2(:,I) = 2*I+1
C
C     At this point, the temporary 2-dimensional arrays are
C     modulated element by element. If an element in TEMP1 is
C     not a multiple of the corresponding element of TEMP2, or
C     (for the sake of an easily generated argument to the up-
C     coming ALL instrinsic) the element in TEMP1 is greater
C     than or equal to the corresponding element in TEMP2, then
C     the corresponding element in the CANDID array is set.
```

```
C
      WHERE ( ((MOD( TEMP1,TEMP2 ) .NE. 0)
     $             .AND. (TEMP1 .GT. TEMP2))
     $             .OR.   (TEMP1 .LE. TEMP2) )
     $       CANDID = .TRUE.
      END WHERE
C
C     The following statement performs an AND across the second
C     dimension of CANDID and stores the results in PRIMES.
C
      PRIMES = ALL(CANDID, DIM=2)
C
C     Print results
C
      PRINT *, "Number of primes:", COUNT(PRIMES)+1
      PRINT *, 2
      DO I=1,N
         IF (PRIMES(I)) PRINT *, 2*I+1
      ENDDO
      END
```

A.4 Laplace Solver

This program solves Laplace's equation

$$\nabla^2 f = 0$$

on the unit square (`[0,1] x [0,1]`), subject to the boundary condition that $f = 1$ at $y = 1$ and $f = 2$ along the rest of the boundary. This program uses the 5-point Jacobi relaxation method, with f initially set to 0 on the interior.

```
      PROGRAM LAPLACE
      PARAMETER (MAXX=32)
      PARAMETER (MAXY=MAXX)
      REAL F(MAXX,MAXY),DF(MAXX,MAXY)
      LOGICAL CMASK(MAXX,MAXY)
      REAL RMS_ERROR,MAX_ERROR
      INTEGER ITERATION
C
C     Initialize the mask for the interior points
C
      CMASK = .FALSE.
      CMASK(2:MAXX-1,2:MAXY-1) = .TRUE.
C
C     Initialize F
C
      F = 2.
      F(:,MAXY) = 1.
      WHERE (CMASK) F = 0.
C
C     Set a dummy value for MAX_ERROR
C
      MAX_ERROR = 1.
      ITERATION = 0
```

```
C
C     Iterate until MAX_ERROR < 1.E-3
C
      DO WHILE (MAX_ERROR.GT.1.E-3)
         ITERATION = ITERATION + 1
C
C        Compute DF, the change at each iteration, and update
C
         DF = 0.
         WHERE (CMASK)
            DF = 0.25*(CSHIFT(F,1,1)+CSHIFT(F,1,-1)+
     S           CSHIFT(F,2,1) + CSHIFT(F,2,-1)) - F
            F = F + DF
         ENDWHERE
C
C     Compute the RMS and Maximum errors.
C
         RMS_ERROR = SQRT(SUM(DF*DF)/((MAXX-2)*(MAXY-2)))
         MAX_ERROR = MAXVAL(DF,MASK=CMASK)
C
C     See if we should print things out
C
         IF (MOD(ITERATION,10).EQ.0) THEN
            WRITE (6,*) ITERATION,RMS_ERROR,MAX_ERROR
         ENDIF
      ENDDO
C
C     Write the final iteration count
C
      WRITE (6,*) ITERATION,RMS_ERROR,MAX_ERROR

      END
```

Index

Index