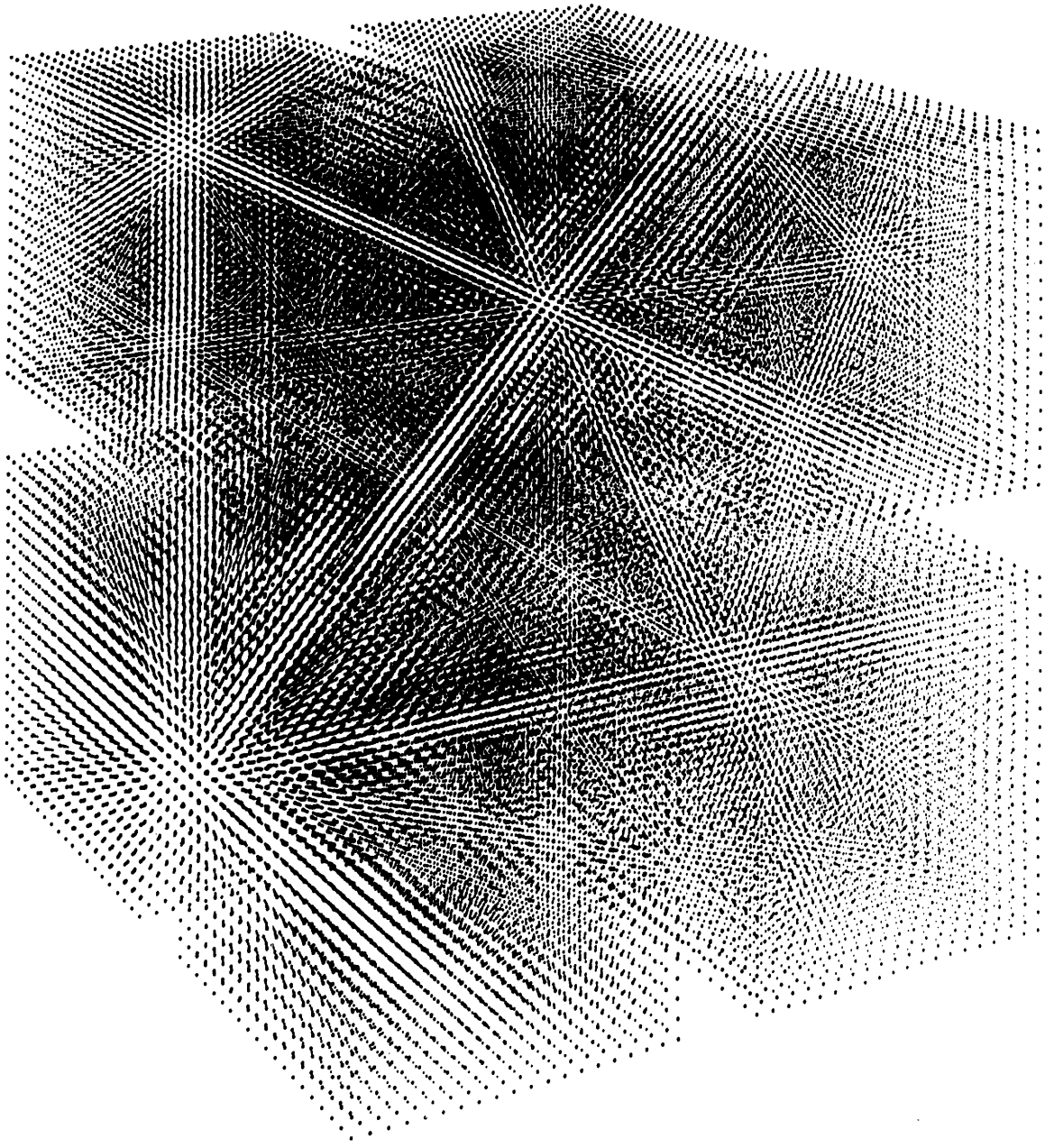
Thinking Machines Corporation

Getting Started in CM Fortran



The
Connection Machine
System

Getting Started in CM Fortran

November 1991

Thinking Machines Corporation
Cambridge, Massachusetts

Revised Edition, November 1991

****

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

****

Connection Machine® is a registered trademark of Thinking Machines Corporation.
CM, CM-1, CM-2, CM-200, CM-5, and DataVault are trademarks of Thinking Machines Corporation.
CMOST and Prism are trademarks of Thinking Machines Corporation.
C*® is a registered trademark of Thinking Machines Corporation.
Paris, *Lisp, and CM Fortran are trademarks of Thinking Machines Corporation.
C/Paris, Lisp/Paris, and Fortran/Paris are trademarks of Thinking Machines Corporation.
Thinking Machines is a trademark of Thinking Machines Corporation.
UNIX is a registered trademark of AT&T Bell Laboratories.

Copyright © 1991 by Thinking Machines Corporation. All rights reserved.

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

Contents

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 backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an Applications Engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

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

Chapter 1

What Is CM Fortran?

The CM Fortran language is an implementation of Fortran 77 supplemented with array-processing extensions from the ISO and ANSI (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 hundreds or thousands of processors to bear on large arrays, processing array elements in unison

This manual introduces CM Fortran as implemented for all Connection Machine models: CM-2, CM-200, and CM-5. The language itself is completely portable among Connection Machine models, although some features of the underlying system architecture and operating system differ from one model to another.

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 assignment or passed as an argument to any Fortran intrinsic function, and the operation is performed on every element of the array.

```
REAL A(40,40,40)

A = 8.0              ! Set all 64,000 elements to 8.0.
A = A * 2.0          ! All 64,000 elements contain 16.0.
A = SQRT( A )        ! All 64,000 elements contain 4.0.
```

Fortran 90 Array References

When an array is referenced in the Fortran 77 manner (with subscripts), its elements are treated as individual scalars. To operate on such an array, you need to step through its elements in a loop, nested as deeply as the number of dimensions. Fortran 90 array operations dispense with the individual subscript references and the **DO** loops.

Fortran 77 style ⟺ **Fortran 90 style**

```
   INTEGER B(100)
   ...
   DO 30 I=1,100
     B(I) = B(I)+1
30 CONTINUE
```

⟺

```
B = B + 1
```

```
   INTEGER C(100,100)
   ...
   DO 30 I=1,100
     DO 40 J=1,100
       C(I,J)=C(I,J)+1
40   CONTINUE
30 CONTINUE
```

⟺

```
C = C + 1
```

The results are the same, but the semantics are slightly different. The Fortran 77 constructs are evaluated in the order specified by the nested loops, whereas Fortran 90 allows the elements of **B** and **C** to be processed in any order, including simultaneously. The CM system takes advantage of this standard feature to process the array elements in parallel.

Data Parallel 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. The CM system stores the array elements in the memories of separate processors and operates on multiple elements at once.

For example, given a 40 x 40 x 40 array **A**, consider the statement:

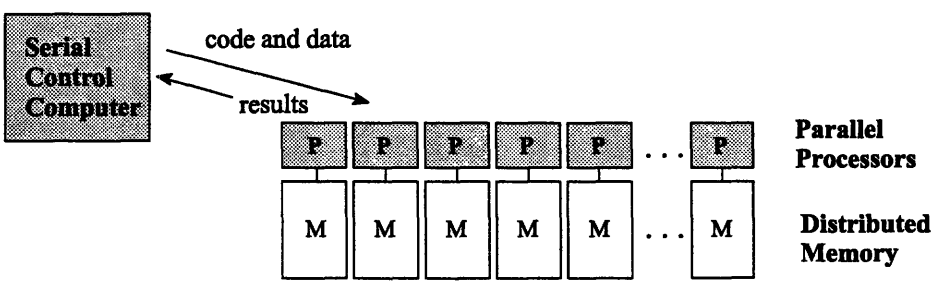
```
A = SQRT(A)/2
```

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

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. This fact is the essential distinction between *data parallel processing* (the execution model for CM Fortran and the other CM languages) and *host-node processing* (or *message passing*).

1.2 The Connection Machine System

The CM program execution environment consists of a set of processors, each with its own memory, all acting under the direction of a serial control processor. (In the CM-5, the control processor is called the *partition manager;* in the CM-2 and CM-200, it is called the *front end.)* To accommodate a data set that is larger than the set of physical processors, the processors subdivide their memory among multiple data elements and process each in turn. As a programming model, you can imagine the parallel processing unit as a set of *virtual processors*, one for each data element.



When the CM system executes a data parallel program, the serial processor directs the flow of control and also operates on scalar data stored in its own memory. For parallel operations, the serial processor directs all the parallel processors (or some selected subset of them) to execute instructions on user data stored in their own memories.

Few useful problems decompose into completely independent subproblems, of course; most require the parallel processors to interact. Fortran 90 defines numerous intrinsic functions and other features that transform arrays or move their elements. These features map naturally onto the CM's mechanisms for interprocessor communication.

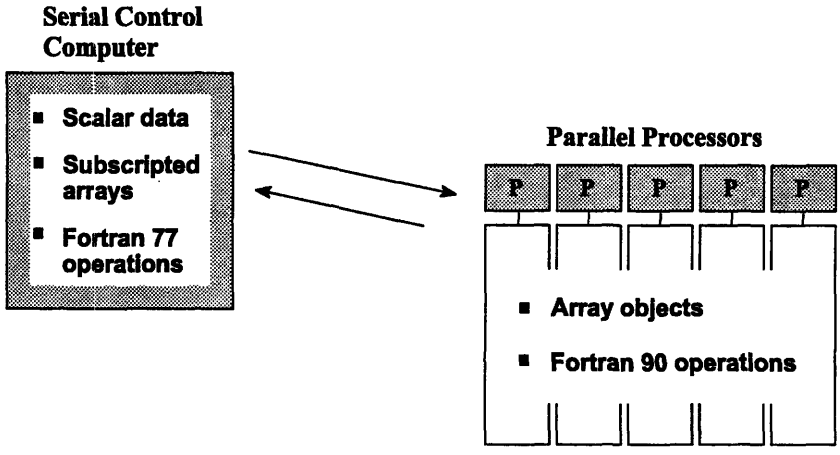
1.3 CM Fortran on the CM System

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 be stored — that is, its *home* can be — either in the centralized memory of the serial processor or in the distributed memory of the parallel processors.

Memory Management

No new data structure is needed to express parallelism, and the programmer need not take any special action to invoke the parallel processing unit. Within each program unit — main program, subroutine, or function — the CM Fortran compiler allocates arrays on one system component or the other *depending on how they are used.*

- Arrays that are used *only* in Fortran 77 constructions in a program unit, and all scalar data, reside on the serial control processor (called the *front end* in CM Fortran). Essentially, the front end executes all of CM Fortran that is Fortran 77.
- Arrays that are used in array operations anywhere in a program unit reside on the parallel processing unit (called the *CM* in CM Fortran). Essentially, the CM executes all of CM Fortran that is Fortran 90.



CM Fortran programmers usually avoid using an array *both* as an array object and as a subscripted array. Such an array has a CM home, but the system moves it to the front end, one element at a time, to perform the serial operation. Naturally, this transfer exacts a performance cost.

CM Array Operations

CM Fortran supports all Fortran 77 features for operating on scalar data and subscripted arrays. For operating on CM arrays (array objects), CM Fortran includes these Fortran 90 array-processing features:

- *Elemental operations.* Most Fortran 77 features — operators, statements, and intrinsic functions — can be applied to CM arrays. These operations are called *elemental* because they behave as if they had been applied separately to each element of the array (in undefined order).

  Because of the CM's distributed memory, certain Fortran 77 features with storage-order dependencies — most notably, the **EQUIVALENCE** statement — cannot be used with CM arrays.

- *Conditionals and subarrays.* Fortran 90 syntax and statements can select subarrays for array operations. The elements of interest can be selected either by their values or by their positions in the array.

- *Array intrinsic functions.* The Fortran 90 intrinsic functions inquire about arrays' properties, shift or transpose elements, determine the location of certain elements (such as the maximum value), perform vector and matrix multiplication, reduce arrays to scalars, or construct new arrays from the information in argument arrays.

CM Fortran also includes some features that are specific to the CM system. These features are particularly useful for data parallel programming:

- *CM Fortran extensions.* CM Fortran offers the **FORALL** statement, which performs an elemental array assignment where the particular values assigned are location-dependent, and several intrinsic functions beyond those defined in Fortran 90.
- *Compiler directives.* Several compiler directives control the layout of arrays in CM memory, which can have major effects on program performance. Another directive controls whether arrays in common blocks are stored on the CM or the front end.
- *Utility Library routines.* These procedures serve a number of purposes, such as:
  - Providing Fortran 90 capabilities that are not yet implemented in CM Fortran, such as generating random numbers
  - Performing actions that are specific to the CM system, such as moving an array *en masse* between front-end memory and CM memory
  - Improving performance in cases where the CM Fortran compiler does not yet translate language syntax into the optimal parallel instruction

CM Fortran Documentation

- *CM Fortran Programming Guide* expands this *Getting Started* guide to describe all the major language features in a task-oriented way.
- *CM Fortran Reference Manual* defines the language and the compiler directives.
- *CM Fortran User's Guide* (published in separate editions for the CM-2/200 and for the CM-5) describes the compiler and its switches, the debugger and other development facilities, and the library of utility routines.
- *CM Fortran Optimization Notes* describe the mapping of CM arrays onto CM memory and provide some hints about efficient CM programming practices.

Chapter 2

A Simple Program

This chapter examines a simple program to illustrate the operations that are fundamental to any array-processing program:

- Declaring arrays
- Moving data into arrays
- Computations on arrays
- Retrieving the results of computations
- Compiling and executing a program

Program **`simple`**, shown on the next page, declares three arrays and uses them in various Fortran 90 array operations. The program also includes a subroutine and a function, which illustrate CM arrays as arguments.

The remainder of this chapter steps through this program, pointing out the essentials of programming in CM Fortran.

The later chapters introduce the methods of operating on selected elements of an array (Chapter 3) and the functions that perform array transformations (Chapter 4).

The Source Program

```
PROGRAM SIMPLE
INTEGER A, B, C, N
INTEGER AVERAGE
PARAMETER (N=5)
DIMENSION A(N), B(N), C(N)

A = 2                          ! a CM array assignment
B = [1:5]                      ! an array constructor
C = A**2 + B**2                ! array-valued expressions

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

PRINT *, 'The largest of C is ', MAXVAL(C)  ! intrinsic function

PRINT *, 'The average of C is ', AVERAGE(C) ! user function

CALL CUBE( C,N )                            ! user subroutine
PRINT *, 'Array C cubed contains: '
PRINT *, C

STOP
END

~~~~~~~~SUBROUTINE~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

SUBROUTINE CUBE( ARRAY, SIZE )
INTEGER SIZE, ARRAY( SIZE )

ARRAY = ARRAY*ARRAY*ARRAY
END

~~~~~~~~FUNCTION~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

INTEGER FUNCTION AVERAGE( ARRAY )
INTEGER ARRAY(:)

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

Compiling and Executing

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

cmf [Connection Machine Fortran version]
compiling simple.fcm... done.
Linking... done.

%
```

You might want to supply a switch that specifies the CM model for which to compile: **-cm5**, **-cm2**, or **-cm200**. The default is determined locally at installation time.

3. Execute the program from a CM partition manager (CM-5) or front end (CM-2/200). Type the executable name after the prompt.

```
% simple

Array C contains:
      5           8          13          20          29
The largest of C is          29
The average of C is          15
Array C cubed contains:
    125         512        2197        8000       24389

 FORTRAN STOP
Detaching... done.

%
```

On the CM-5, this command line causes the program to execute on the partition manager and on whatever number of parallel processors the partition manager currently controls. On the CM-2/200, this command line attaches the front end to some default number of processors for the duration of program execution.

2.1 Declarations

The specification part of program **simple.fcm** is familiar Fortran 77. It could also have used the **DATA** statement and the **COMMON** statement. All the Fortran 77 data types are supported, plus **DOUBLE COMPLEX**.

At this point in the program, there is no distinction between front-end arrays (subscripted arrays) and CM arrays (array objects): **A**, **B**, and **C** could be either, depending on how they are later used in the executable part of the program unit. Only in certain cases does the specification part of the program determine an array's home:

- All character arrays have a front-end home and must be used in the Fortran 77 manner. (Since the CM system does not support parallel processing of character arrays, these arrays cannot be used in Fortran 90 array operations.)
- Common arrays have a CM home by default. The compiler assumes that common arrays are intended for use in array operations unless the user specifies otherwise with a compiler directive or switch (see Section 2.4, below).

2.2 Array Operations

An array operation is any reference to an array object — that is, any use of the array name without subscripts — in an expression, assignment, or intrinsic function call. The various forms of array operation are all illustrated in program **simple.fcm**:

```
A = 2                     ! a CM array assignment
C = A**2 + B**2           ! array-valued expressions
PRINT *, ... MAXVAL(C)    ! intrinsic function call
```

These statements cause the three arrays to be allocated on the CM, where the operations are carried out in parallel.

The function **MAXVAL** is an example of the array-processing intrinsic functions that CM Fortran adds to Fortran 77. Most of the new array-processing intrinsics take only array objects as arguments (not scalars), and they always execute on the CM. The Fortran 77 intrinsics are extended in CM Fortran to take either scalars or array objects as arguments.

Array Constructors

Notice the assignment of initial values to array **B** in program **simple.fcm**:

```
B = [1:5]
```

The construction on the right is a Fortran 90 feature called an *array constructor.* An array constructor is a sequence of values enclosed in square brackets; it specifies an unnamed, one-dimensional array containing those values. In CM Fortran, an array constructor is always treated as a CM array; it can be used in an array assignment or passed as an argument to an intrinsic function.

Array constructors can specify values in several ways:

```
ARRAY = [ 1,2,3,4,5,6,7,8,9,10 ]   ! List the values
ARRAY = [ 1:20:2 ]                 ! Specify a sequence
ARRAY = [ 5[0], 5[1] ]   ! Specify one or more repeat counts
```

In the first form, the values can be any type other than character. If you list more than one type, the constructed array is the same type as the first value listed. In the second two forms, the values specified must be integers, but you can coerce them to another type by means of any of the Fortran 77 type-conversion functions.

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. Scalars can be used freely in array assignments and array-valued expressions, since Fortran 90 defines a scalar as conformable with any array.

```
A = 2
C = A**2 + B**2
```

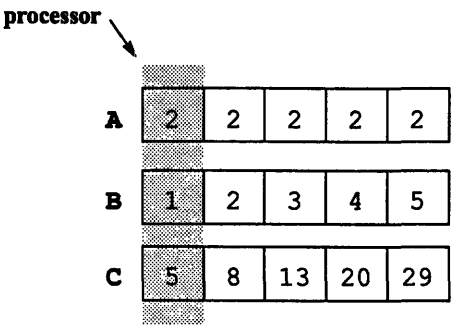
The first statement causes every element of **A** to receive a 2. In effect, 2 is treated as a five-element vector of 2's, and each element of **A** is assigned an element of that vector. (In fact, the front end "broadcasts" 2's to all the CM processors, where they are treated as immediate operands in the assignment.) In the second statement, every element of **C** receives the sum of the squares of the corresponding elements of **A** and **B**.

Fortran 90 does not define the effect of mixing array objects of different sizes and shapes in an expression or assignment:

```
REAL C(5), D(10,10)
...
C = D                          ! ERROR: Nonconformable arrays
```

This assignment of **D** to **C** becomes meaningful only if you select a one-dimensional, five-element subarray, or *array section*, from **D**. The syntax for specifying an array section is shown later (in Chapter 3).

CM Fortran implements operations on conformable arrays by configuring a set of processors into a logical grid of the appropriate shape for the arrays. Arrays of many different sizes and shapes can coexist in CM memory, but conformable arrays are always stored in the same set of processors in the same order. Thus, elements **A(1)**, **B(1)**, and **C(1)** all reside in the local memory of the same processor, as do **A(2)**, **B(2)**, and **C(2)**, and so on. Each processor executes the operations on its own set of array elements; no data motion occurs between processors.



Notice, though, that if you assign a section of the two-dimensional array **D** to the vector **C**, the system must move data into the appropriate processors before it can proceed with the assignment.

This fact suggests one of the basic principles of CM programming: operations on the corresponding elements of conformable arrays are the most efficient use of the system. Given the CM's distributed memory, the common Fortran 77 practice of declaring one or a few large arrays and selecting pieces of them as needed often forces the system to move data into the appropriate processors before acting upon it. It is better, wherever possible, to declare multiple arrays of the same shape and to operate on their corresponding elements. When you do this, the data does not need to move to the appropriate processors — it is already there.

2.3 Input-Output

Program **simple.fcm** uses the familiar Fortran syntax to retrieve the results of the array operations. CM Fortran supports all Fortran I/O operations — the **READ**, **WRITE**, and **PRINT** statements — for CM data as well as for front-end data. These statements cause CM data to be displayed from the front end or placed in (or retrieved from) the UNIX file system.

The **PRINT** statement lets you view all the results stored in array **C**:

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

For large vectors or for matrices, you can use a **FORMAT** statement to improve the readability of the output:

```
      INTEGER MATRIX(4,4)
     . . .
     PRINT 10, MATRIX
10   FORMAT (4I9)
```

You can also retrieve a scalar value from the CM by subscripting a CM array in the Fortran 77 fashion. Notice that this is a deliberate use of a "mixed-home" construction: the array element that is referenced with a Fortran 77 subscript is automatically moved to the front end, where you can view it or use it like any other scalar value:

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

Finally, you can derive a scalar value (and thus a front-end value) by applying an intrinsic reduction or inquiry function to a CM array. The reduction functions, such as **MAXVAL** and **SUM**, perform a combining operation on an array's elements and return the scalar result to the front end. The inquiry functions, such as **DSIZE** and **DUBOUND**, return the requested array property as a scalar. Program **simple.fcm** displays these scalar results with **PRINT** statements:

```
INTEGER AVERAGE
. . .
PRINT *, 'The largest of C is ', MAXVAL(C)  ! intrinsic
PRINT *, 'The average of C is ', AVERAGE(C) ! user function
. . .
AVERAGE = SUM( ARRAY ) / DSIZE( ARRAY )
```

In addition to the **PRINT** statement, you can also use the Fortran **READ** and **WRITE** statements in exactly the same way as you use them with front-end data. Data retrieved in this way passes through front-end memory on its way between CM memory and the UNIX file system.

For large data sets, it is more efficient to bypass the front end and move data directly, in multiple streams, between CM memory and a file. To perform this *parallel I/O*, you use the CM Fortran Utility Library routines.

2.4 Procedures

Procedures are defined and invoked in Fortran 90 in much the same way as in Fortran 77, but there is — again — a crucial difference in semantics when the argument is a CM array object. Like the CM array objects referenced in array operations, an array object passed as an argument is the whole array.

For example, consider the invocation of the two user-defined procedures in program **simple.fcm**:

```
DIMENSION C(N)

PRINT *, 'The average of C is ', AVERAGE(C) ! function
CALL CUBE( C,N )                            ! subroutine
```

These procedure calls look just like procedure calls in Fortran 77. However, since array **C** has been established in the main program as a CM array object, the references to it as an argument specify all **N** elements of **C**, not just the first element. This difference in the semantics of a procedure call has certain implications for defining and invoking procedures in CM Fortran.

Declaring Dummy Arrays

As in Fortran 77, the type of an actual argument must match the type of the corresponding dummy argument. In addition, in CM Fortran the *shape* of the actual and dummy arrays must match. That is, a dummy array argument must be declared in such a way that its rank and the length of each dimension are the same as those of the actual array argument passed.

Notice the declaration of the dummy array argument in subroutine **CUBE**:

```
SUBROUTINE CUBE( ARRAY, SIZE )
INTEGER SIZE, ARRAY( SIZE )

ARRAY = ARRAY*ARRAY*ARRAY
END
```

The parameter **N**, which is the length of array **C** in the main program, is passed as an argument to subroutine **CUBE**, where it specifies the length of dummy array **ARRAY**. Because **C** is to be the actual argument, the dummy argument *must* be of rank one and length **N**. In CM Fortran, it is an error to resize or reshape an array object across procedure boundaries.

You are not restricted, however, to declaring a dummy array to match some particular actual array. A dummy argument can also be *assumed-shape,* which means that it *assumes* the shape of the actual argument. An assumed-shape array is declared without explicit dimension bounds; you simply specify a colon for each dimension, with commas between.

For example, notice the declaration of the dummy **ARRAY** in function **AVERAGE**. The dummy is of rank one, but it can be of any length. When the function is invoked with array argument **C**, the dummy assumes the length of **C**.

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

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

This particular function returns a scalar result; in fact, since the intrinsic functions **SUM** and **DSIZE** return scalars to the front end, the division operation is executed on the front end. User-defined functions can also be computed entirely on the CM and return array-valued results (as described in the CM Fortran documentation set). The behavior of such a function is like that of a subroutine that takes an array as its first argument and stores its results there.

Passing CM Array Arguments

The use that a procedure makes of a dummy array determines the home — CM or front end — of that array. Both subroutine **CUBE** and function **AVERAGE** use their dummy arrays in Fortran 90–style array operations, and the arrays are therefore assumed to be allocated on the CM:

```
ARRAY = ARRAY*ARRAY*ARRAY                ! from subroutine CUBE

AVERAGE = SUM(ARRAY) / DSIZE(ARRAY)      ! from function AVERAGE
```

Actual array arguments must match the corresponding dummies in *home*, as well as in type and shape. It is an error to pass a front-end array to a procedure that expects a CM array, or to pass a CM array to a procedure that expects a front-end array.

When a procedure contains array operations, the programmer must see to it that the actual argument is allocated on the CM. One way to do this is to use the array in a Fortran 90–style array operation in the calling procedure. Recall that an array operation is any unsubscripted reference to the array in an expression, assignment, or intrinsic function call. Another way is to declare the dummy argument as an assume-shape array. Assumed-shape arrays are always taken to be CM arrays, no matter how they are used in the procedure.

Finally, a third way to force an array onto the CM is to use the compiler directive **LAYOUT**. This directive is intended to control the particular way an array is laid out across (or within) CM processors. It can, for instance, direct the compiler to lay out the elements of a specified dimension all within the same processor (thus creating a *serial dimension),* while laying out the other dimensions across processors in the usual way. A subsidiary effect of **LAYOUT** is that it also controls an array's home.

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 array **C** onto the CM:

```
      DIMENSION C(N)
CMF$  LAYOUT C (:NEWS)
```

CMF$ must start in column 1, to indicate that this structured comment is a compiler directive. Any array dimension can be laid out in-processor (instead of cross-processor) by labeling it **:SERIAL**. If all dimensions are made serial, the array is allocated on the front end and cannot be used in array operations.

Declaring Local Arrays

Like Fortran 90, CM Fortran permits the dynamic allocation of local arrays in a procedure. For example, compare the two arrays declared in this subroutine:

```
SUBROUTINE X( ARRAY, SIZE )
INTEGER SIZE
REAL ARRAY( SIZE ), TEMP( SIZE )

TEMP = ARRAY            ! store ARRAY's initial values

. . .      various array operations on ARRAY . . .

ARRAY = ARRAY + TEMP   ! compute initial values back into ARRAY
END
```

ARRAY is a straightforward dummy array that stands for an actual passed in at run time. The array **TEMP**, however, is a Fortran 90 *automatic array.* Storage for **TEMP** is allocated upon entry to the procedure and deallocated upon exit from the procedure. Its size might be passed in at run time, as in this example, or it might be specified with a constant. (Automatic arrays are always allocated on the CM, regardless of how they are used in the procedure.)

Using Common Arrays

Common arrays can reside either on the front end or on the CM. Since common arrays are normally used in several program units, the compiler cannot determine the proper home from their use in the program unit being compiled. It assumes, therefore, that common arrays are intended for use in array operations and allocates them on the CM unless directed otherwise.

You can override the compiler's default allocation of common arrays in either of two ways:

- Use the compiler directive **COMMON** to give particular common blocks one home or the other. For example:

```
      REAL A(N), B(N), C(N), D(N)
      COMMON /BLOCK_1/  A,B
      COMMON /BLOCK_2/  C,D

CMF$  COMMON CMONLY /BLOCK_1/    ! redundant with default
CMF$  COMMON FEONLY /BLOCK_2/    ! C and D are on front end
```

The arrays in **BLOCK_1**, like all CM arrays, can be used in array operations on the CM or (at some performance cost) in serial operations on the front end. The arrays in **BLOCK_2**, like all front-end arrays, can be used *only* in serial operations on the front end.

- Compile with the switch **-fecommon**, which causes common arrays to be allocated on the front end.

  When this switch is used, *no* common array can be used in an array operation *except* those that are constrained to a CM home by a compiler directive. The compiler directives **LAYOUT** and **COMMON** override the effect of the compiler switch **-fecommon** for the particular arrays to which they apply.

In CM Fortran all the CM common arrays used anywhere in a program must be declared in the main program. Common arrays that are constrained to the front end need not be declared in the main program unless they are referenced there.

Chapter 3

Selecting Array Elements

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

- By value: conditional (or *masked)* operations include or exclude array 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 original array.

A CM Fortran extension, the **FORALL** statement, allows you to select array elements both by value and by position for performing position-dependent actions on an array or array section.

The **FORALL** statement and the array section syntax are especially useful for indicating data motion, such as shifting array elements in a grid pattern, arbitrary permutations and indexing, and parallel-prefix ("scan") operations.

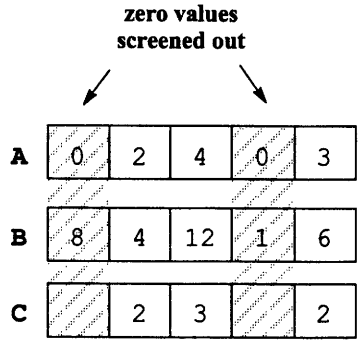
3.1 Conditional Operations

An array assignment can be made conditional on an array's values by enclosing the assignment in a **WHERE** statement. **WHERE** is the array-processing extension of the Fortran **IF** statement: it specifies a logical array (or array-valued expression) as the test, followed by an array assignment. The arrays in the assignment and the

mask array must all be conformable. For example, to avoid division by zero in an array operation:

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

You can imagine the system overlaying all the conformable arrays with a mask to screen out the elements in positions that correspond to the false values of the test expression. The remaining elements participate in the assignment statement.



Like the Fortran **IF**, the **WHERE** statement can be expanded into a construct with the **END WHERE** statement and an optional **ELSEWHERE**. The body of a **WHERE** construct can contain multiple array assignments; it cannot contain nested **WHERE** statements or external procedure calls. For example:

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

In a **WHERE–ELSEWHERE** construct, you can imagine the system segregating the elements into disjoint sets and then performing the appropriate array operations (in parallel) on each of the sets.

3.2 Array Sections

Fortran 90 defines *triplet subscripts*, a new syntax for specifying a sequence of subscripts in an array reference. The subscript sequence indicates the subset, or *section*, of the array to be operated upon. Not surprisingly, since a triplet replaces an explicit DO loop, it has the feel of a DO control specification.

array-name (*first* : *last* : *stride*)

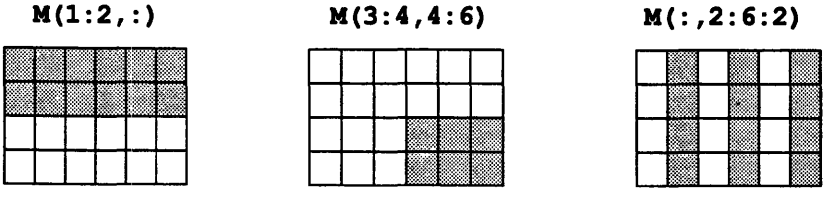
A triplet indicates, for one array dimension, the beginning and terminal indices and the increment. The *first* and *last* subscript values default to the declared bounds of the dimension; *stride* defaults to 1. If the entire triplet is allowed to default for every dimension, we are left with simply the array name. The array names A, B, and C used in the sample program in the previous chapter are the defaulted forms of A(1:5:1), B(1:5:1), and C(1:5:1).

Triplet Examples

The following array references are sections of vector V(10):

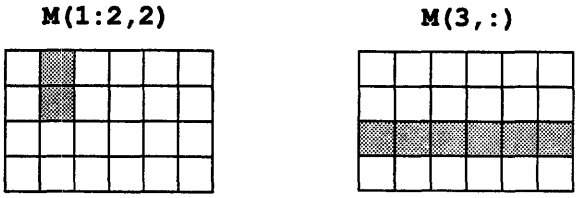
```
V(1:5)        ! first five elements
V(6:10)       ! last five elements
V(10:1:-1)    ! all ten elements in reverse order
V(1:10:2)     ! first, third, fifth, seventh, ninth
V(10:1:-2)    ! tenth, eighth, sixth, fourth, second
```

To take a section of a multidimensional array, you specify a triplet for each dimension, with commas separating them. Any of the triplets can be allowed to default to the whole dimension, as long as the placeholder commas and colons are retained to avoid ambiguity. For example, here are some sections of matrix M(4,6):



In these examples the section has the same rank (number of dimensions) as its *parent* array, although its size and shape are different. To get a section of lower rank than its parent, you can replace one or more of the triplets with a Fortran 77

scalar subscript. The scalar subscript indicates a single row or column, rather than a sequence of rows or columns:



Using Array Sections

Array sections can be used anywhere that whole arrays are used. As with whole arrays, sections that are used together in an expression or assignment must be conformable. For example, here are some array operations on sections of vectors **A(5)**, **B(5)**, and **V(10)**:

```
V(1:5) = A**2 + B**2
B = A(5:1:-1)
PRINT *, MAXVAL( B(1:4) )
A(2:5) = [ 10,20,30,40 ]
WHERE ( A(1:3).NE.0 )  V(1:3) = B(1:3) / A(1:3)
```

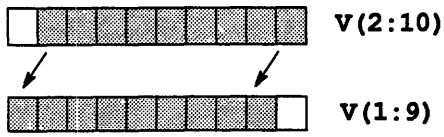
Any array reference that uses triplet notation is a Fortran 90 array reference and thus causes the parent array to be allocated on the CM.

Array Sections and Data Motion

Array sections are particularly useful for moving data in regular grid patterns in applications such as convolutions or image rotation. You move data by assigning one section of an array to another section of the same array or another array. As in all array operations, the sections must be conformable. (The shape of the parent arrays does not affect the behavior.)

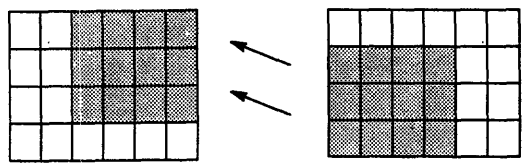
For example, to shift vector values to the left:

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



To shift data on more than one dimension:

```
M(1:3,3:6) = M(2:4,1:4)
```

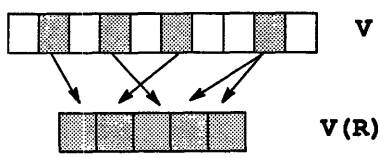


Vector-Valued Subscripts

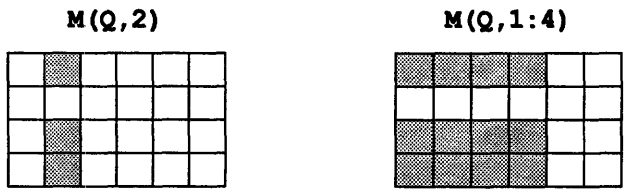
A vector-valued subscript is a form of array section that uses a vector of index values as a subscript. Since the index values need not be ordered — that is, there is no fixed stride — this syntax can specify any arbitrary selection of array values along a dimension. Vector-valued subscripts are useful for vector permutations and for indexing into a vector or array dimension. (For array permutations, see the discussion of the **FORALL** statement, Section 3.3, below.)

For example, if `V` is a vector of length 10 and `P` is a permutation of the integers from 1 to 10, then `V=V(P)` applies this permutation to the values in `V`. The statement `V(P) = V` applies the inverse permutation.

The index values can be repeated, which causes element values to be repeated in the section. For example, if `R` is the vector `[2,6,4,9,9]`, then `V(R)` is a five-element vector whose values are `V(2)`, `V(6)`, `V(4)`, `V(9)`, and `V(9)`, in that order:



Vector-valued subscripts can be mixed with triplet subscripts and with Fortran 77 scalar subscripts in an array reference. The rank of the resulting array section is the number of dimensions indicated with either of the Fortran 90 subscripts, not counting any with scalar subscripts. For example, given the matrix `M(4,6)` and the vector `Q = [1,3,4]`, consider these two array sections:



The reference on the left indicates a one-dimensional section of length 3 and the reference on the right indicates a two-dimensional section of shape `[3,4]`. The values in the sections are the shaded elements of the parent matrix. (The values would be reordered if the indices in `Q` were, say, `[1,4,3]`.)

Array Sections as Arguments

Array sections, like any array object, can be passed as arguments to intrinsic functions and user-defined procedures. Array sections are always CM arrays, because of the Fortran 90 syntax in the array reference. The dummy array argument in the procedure must therefore be a CM array, and the actual must match it in type and shape. For example:

```
REAL A(100), B(100,100)
. . .
CALL SELECT_AND_SORT( A(1:50) )
CALL SELECT_AND_SORT( A(51:100) )
CALL SELECT_AND_SORT( B(1,1:50) )
. . .
END


SUBROUTINE SELECT_AND_SORT( ARRAY )
REAL ARRAY(50)

. . .     various array operations on ARRAY . . .

END
```

The dummy argument in this subroutine is a vector of 50 real elements, used in CM array operations. The actual arguments in the three calls to the subroutine are also 50-element real vectors, allocated on the CM because of the triplet syntax in the array reference.

You could also declare the dummy argument as an assumed-shape array:

```
SUBROUTINE SELECT_AND_SORT( ARRAY )
REAL ARRAY(:)
```

In this case, the actual arguments would have to be real vectors allocated on the CM, but they could be of any length.

3.3 The FORALL Statement

FORALL, a CM Fortran extension to Fortran 90, is the array-processing feature that looks most like a **DO** loop. A **FORALL** statement defines one or more index variables and uses them in an assignment statement, thus indicating action that depends on the positions of the target array elements.

For example, to give each element of vector **V** its own index value:

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

Similarly, to initialize a matrix with sequential integers in Fortran array-element order:

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

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

Or, to initialize 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)
```

There is one crucial semantic difference between **FORALL** statements and **DO** constructs. The individual assignments in a **FORALL** statement are executed in undefined order but *as if simultaneously.*

- Since the individual assignments need not be sequential, they can be performed in parallel on the CM.
- Since the assignments are *as if simultaneous*, even if performed serially on the front end, the program need not take action to save the initial value of an element that is both a target of one assignment and the source of another. For example, you can transpose matrix elements without explicitly passing them through a temporary location:

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

FORALL is not considered an array operation for the purpose of determining the homes of arrays. The assignment can involve either front-end arrays or CM arrays, and the homes of the arrays determine whether the statement executes on the front end or on the CM. It is possible — but usually not wise, for performance reasons — to mix front-end and CM arrays in a **FORALL** assignment, since in this case the CM arrays are moved element by element to the front end.

Array Assignments in FORALL

In CM Fortran, the assignment in a **FORALL** statement (and in a **DO** construct) can involve whole arrays and array sections as well as individual elements. For example, to spread a vector **V** along the first dimension of a matrix **H**:

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

As in any array operation, the arrays in a **FORALL** array assignment must be conformable. In this example, the vector of length **M** is assigned to each row of the matrix. The rows are also of length **M**.

While **FORALL** normally does not determine an array's home, the two arrays in this example are referenced with triplet subscripts (implicitly in the case of **V**). These Fortran 90–style references are sufficient to cause the two arrays to be allocated on the CM, and the **FORALL** statement therefore executes in parallel on the CM.

Conditional FORALL Statements

A **FORALL** statement can contain a mask expression, which prevents certain elements from being assigned. The effect is similar to embedding an **IF** statement in a **DO** construct.

The mask is always a *scalar-valued* expression, even when the assignment references whole arrays or array sections. For example, to avoid division by zero in a **FORALL** statement:

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

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
```

Data Motion with FORALL

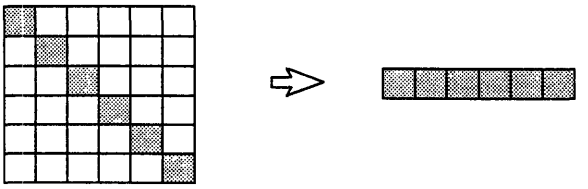
FORALL is a powerful feature for expressing data motion. It can mimic the behavior of other CM Fortran features, such as array sections and many of the intrinsic functions shown in the next chapter. Its real value, however, is in expressing patterns of data motion that are otherwise difficult to express as parallel operations in CM Fortran.

For example, **FORALL** can perform, in parallel, arbitrary permutations of multidimensional arrays. The following statement indexes into matrix **H**, using index arrays **X** and **Y**:

```
FORALL ( I=1:N,J=1:M )  G(I,J) = H( X(I,J), Y(I,J) )
```

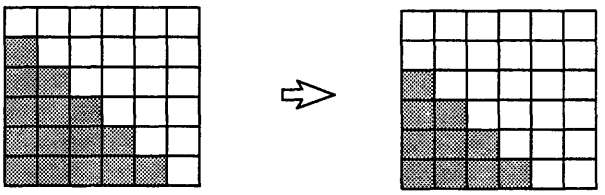
FORALL can also operate, in parallel, on irregularly shaped parts of an array. For example, to extract the diagonal elements of a matrix and assign them to a vector:

```
FORALL (I=1:N)  V(I) = H(I,I)
```



To shift the lower triangle of a matrix by one position:

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



Finally, **FORALL** can express parallel prefix, or *scan*, operations along an array dimension. Scan operations apply some combinator cumulatively along a dimension, giving each element the combination of itself and all previous elements (like computing the running balance of a checkbook). For example, to express an add-scan of array **A**, compute the sum-reductions of progressively larger sections of **A**:

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

Parallel vs. Serial Execution

When **FORALL** is applied to CM arrays, it usually executes in parallel on the CM. There are exceptions, however, as noted in the *CM Fortran Release Notes* for the current release. As new capabilities are added to **FORALL**, they sometimes execute serially in the early implementations, but execute in parallel in later releases. A set of utility routines, described in the *CM Fortran User's Guide*, allows you to work around any restrictions on the parallel execution of **FORALL** statements.

Chapter 4

Array Transformations

CM Fortran supplies a rich set of intrinsic functions for manipulating, describing, and transforming arrays. The intrinsic functions are of four different kinds:

- *Elemental intrinsics.* These functions are the Fortran 77 intrinsics (excluding the character functions); they can take either a scalar value or an array as an argument.
- *Inquiry intrinsics.* These functions, illustrated earlier in the discussion of procedures, return information about the properties of an array or of a specified array dimension. They include **DSIZE**, **DSHAPE**, **RANK**, **DUBOUND**, and **DLBOUND**.
- *Location intrinsics.* These functions determine the location of certain array elements, such as the maximum numeric value or the first true value. They include **MAXLOC**, **MINLOC**, **FIRSTLOC**, **LASTLOC**, and **PROJECT**.
- *Transformational intrinsics.* These functions take array arguments and apply some transformation to them, returning scalars in some cases and arrays in others. The result arrays are often a different shape from the argument arrays.

All these functions are described in full in the CM Fortran documentation set. This chapter gives a brief overview of the transformational intrinsics to suggest the power and convenience of array-processing in CM Fortran. The array transformations you can perform are:

- Data movement
- Array reduction
- Array construction
- Array multiplication

4.1 Data Movement Functions

Data movement functions reposition the elements of an array. They include **CSHIFT** ("circular shift"), **EOSHIFT** ("end-off shift"), and **TRANSPOSE**. This section describes **CSHIFT** to illustrate the data movement functions.

CSHIFT takes as arguments an array object, an integer indicating the dimension on which to shift element values, and another integer *or* array indicating the shift offset:

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

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

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

shifts the values in the rows (the second dimension) of **A** one column position in the negative direction (to the right), wrapping the right-most values around to the first column. (Notice that CM Fortran allows you to identify the arguments in intrinsic function calls with predefined keywords). The effect of the statement above on array **A** is:

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

The **SHIFT** argument to **CSHIFT** can also be an array, indicating a possibly different shift distance for each row, column, or plane of the argument array. The **SHIFT** array must be of rank one less than the argument array. For example,

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

has the following effect on **A**:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 3 & 4 & 5 & 1 \\ 3 & 4 & 5 & 1 & 2 \\ 4 & 5 & 1 & 2 & 3 \end{bmatrix} \begin{matrix} \text{(shift by 1)} \\ \text{(shift by 2)} \\ \text{(shift by 3)} \end{matrix}$$

(Notice the use of an array constructor as a CM array argument to an intrinsic function.)

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)
```

4.2 Array Reduction Functions

The reduction functions take an array and "summarize" it by applying some combining operation across its elements, thus reducing the array either to a scalar or to an array of lower rank. They include the numeric functions **MAXVAL**, **MINVAL**, **SUM**, **PRODUCT** and the logical functions **ANY**, **ALL**, and **COUNT**. This section describes **MAXVAL** and **ANY** to illustrate the behavior of the reduction functions.

The MAXVAL Function

Like the other numeric reduction functions, **MAXVAL** takes an array object, an optional integer that specifies the dimension to reduce, and an optional array mask.

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

For example, consider a two-dimensional array **A**:

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

When you supply only the array argument (either a whole array or an array 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 a dimension argument is supplied, the result is a CM array whose rank is one less than the argument array. A reduction on the first dimension of **A** yields a

4-element vector containing the maximum of each of the columns; a reduction on the second dimension yields a 3-element vector containing the maximum of each of the rows.

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

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

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

The mask argument indicates which elements to compute into 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 ]
```

The ANY Function

Like the other logical reduction functions, **ANY** takes an array object and an optional integer that specifies the dimension to reduce. The logical array argument is identified with the keyword **MASK**; these functions work only on the elements that are **.TRUE.**

```
ANY( 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 **.TRUE.** if any of the elements in the array (or section) is true. If a dimension is specified, the result is a CM 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 ]
```

4.3 Array Construction Functions

The construction functions construct new CM arrays by using the elements in existing arrays in specified ways.

- **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 element-wise 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 by using 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** and **SPREAD**.

The RESHAPE Function

With the CM system's distributed memory, reshaping arrays is not as routine a practice as it is with centralized-memory machines. Reshaping a single large array should not be used as a substitute for the separate declarations of smaller arrays, since reshaping entails actual data movement in CM memory 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 mold argument specifies the target shape; it is a vector of positive integers, each indicating the extent of a target dimension. Unless the call specifies otherwise, the source array elements are placed in the target array in array-index order. For example, assuming the existence of a 3 x 4 array **X**, the statement

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

reshapes the source vector and places the following values in **X**:

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

(Note again that CM array arguments to the intrinsic functions can be specified with array constructors.)

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

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

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

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

The 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 two-dimensional mold would be **[2,1]**, with the following effect:

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

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

The SPREAD Function

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 two-dimensional, the result array is three-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**:

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

4.4 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 (CMSSL). These are described in the CM documentation set.

Chapter 5

Sample Programs

This chapter presents several very simple programming examples in CM Fortran.

- A prime number sieve
- A solver for Laplace's equation

5.1 Prime Number Sieve

Beginning on the following page is a program that finds the prime numbers in a set of numbers. Three versions of the program are presented for comparison: a serial version and two alternative parallel versions.

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
         END IF
 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
```

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_1
      IMPLICIT NONE         ! suppress implicit typing rules
      INTEGER I, N, NEXTPRIME
      PARAMETER (N = 999)
      LOGICAL PRIMES(N), CANDID(N)

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           Notice support for DO WHILE and END DO.
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
      END DO
      END
```

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.

```
      PROGRAM FINDPRIMES_2
      IMPLICIT NONE
      INTEGER I, N, NN
      PARAMETER (N = 500)
      INTEGER TEMP1(N,N), TEMP2(N,N)
      LOGICAL CANDID(N,N), PRIMES(N)

      CANDID = .FALSE.
      FORALL (I=1:N)  TEMP1(I,:) = 2*I+1
      FORALL (I=1:N)  TEMP2(:,I) = 2*I+1
C
C     The temporary 2-dimensional arrays are now modulated
C     element by element. If an element in TEMP1 is not a
C     multiple of the corresponding element of TEMP2, or (for
C     the sake of an easily generated argument to the upcoming
C     ALL instrinsic) the element in TEMP1 is greater than or
C     equal to the corresponding element in TEMP2, then the
C     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     Perform an AND across the second dimension of CANDID.
      PRIMES = ALL(CANDID, DIM=2)
C
C     Print results
      PRINT *, 'Number of primes:', COUNT(PRIMES)+1
      PRINT *, 2
      DO I=1,N
         IF (PRIMES(I)) PRINT *, 2*I+1
      END DO
      END
```

5.2 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 mask and F for boundaries and interior
C
      CMASK = .FALSE.
      CMASK( 2:MAXX-1, 2:MAXY-1 ) = .TRUE.

      F = 2.
      F( :,MAXY ) = 1.
      WHERE (CMASK) F = 0.

      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
         END WHERE
```

```
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
           END IF
         END DO
C
C       Write the final iteration count
C
         WRITE (6,*) ITERATION,RMS_ERROR,MAX_ERROR

         END
```

Index

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1214
(617) 234-1000