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).

NOTE: This chapter shows how to compile and execute program **`simple`** in a way that works on all CM system configurations, using site defaults. For more information about compiler options and about the various CM execution environments, see the *CM Fortran User's Guide.*

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 *, ƒArray C contains: ƒ
PRINT *, C                      ! output of CM data

PRINT *, ƒThe largest of C is ƒ, MAXVAL(C)  ! intrinsic function

PRINT *, ƒThe average of C is ƒ, AVERAGE(C) ! user function

CALL CUBE( C,N )                            ! user subroutine
PRINT *, ƒArray C cubed contains: ƒ
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

   %
   ```

   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 reference. 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
```

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
```

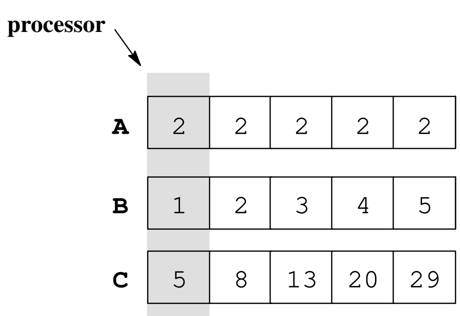
The first statement causes every element of **A** to receive a 2. In effect, 2 is treated as a five-element vector of 2's, and each element of **A** is assigned an element of that vector. (In fact, the front end "broadcasts" 2's to all the CM processors, where they are treated as immediate operands in 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 *, ƒArray C contains: ƒ
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 *, ƒThe third element of array C is ƒ, 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 *, ƒThe largest of C is ƒ, MAXVAL(C)  ! intrinsic
PRINT *, ƒThe average of C is ƒ, 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 *, ƒThe average of C is ƒ, 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.