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**:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 3 & 4 & 5 & 1 \\ 3 & 4 & 5 & 1 & 2 \\ 4 & 5 & 1 & 2 & 3 \end{bmatrix} \begin{array}{l} \texttt{(shift by 1)} \\ \texttt{(shift by 2)} \\ \texttt{(shift by 3)} \end{array}$$

(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**, **MIN–VAL**, **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`**:

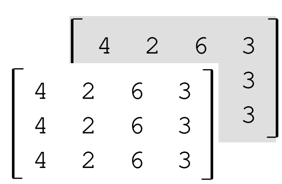
$$C = \begin{bmatrix} 4 & 4 & 4 \\ 2 & 2 & 2 \\ 6 & 6 & 6 \\ 3 & 3 & 3 \end{bmatrix}$$

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 =



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.