Chapter 9

Array Transformations

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

- *elemental* functions (Chapter 2), which apply Fortran 77 numeric and mathematical operations to all the elements of array arguments (as well as to scalars)
- *inquiry* functions (Chapter 6), which return information about an array (CM or front-end), such as its size or shape
- *movement* functions (Chapter 8), which reposition the elements of a CM array by performing a dimensional shift or a matrix transposition

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

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

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

Copyright © 1989–1994 Thinking Machines Corporation

9.1 Array Reduction

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

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

9.1.1 The ARRAY Argument

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

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

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

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

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

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

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

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

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


Copyright © 1989–1994 Thinking Machines Corporation

9.1.2 The MASK Argument

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

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

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

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

Then,

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

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

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

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

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


Copyright © 1989–1994 Thinking Machines Corporation

9.1.3 Logical Arrays

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

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

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

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

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

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

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

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

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


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

9.2 Element Location

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

9.2.1 Numeric Arrays

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

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

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

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

the numeric location functions behave as follows:

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

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

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

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

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


Copyright © 1989–1994 Thinking Machines Corporation

Special Cases

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

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

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

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

No DIM Argument

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

MASK Argument

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

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

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

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

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

Then,

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

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


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

9.2.2 Logical Arrays

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

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

For example,

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

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

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

Similarly,

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

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

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

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

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

the result is an array that conforms to the section:

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

Copyright © 1989–1994 Thinking Machines Corporation

DIM Argument

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

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

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

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

Similarly,

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

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

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

9.2.3 Element Location Example

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

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

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


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

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

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

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

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

9.3 Array Construction

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

- **RESHAPE** takes an array and constructs a new array with the same elements but a different shape.
- **DIAGONAL** takes a vector and constructs a matrix whose diagonal elements are those of the vector and whose other ("fill") elements are all a specified or default value.
- **MERGE** combines two conformable arrays into a new array by means of an 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.

Copyright © 1989–1994 Thinking Machines Corporation

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

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

9.3.1 Reshaping an Array

With the CM system's distributed memory, reshaping arrays is not as routine a practice as it is with linear memory machines. Reshaping a single large array should not be used as a substitute for separate declarations of smaller arrays, each in the shape needed. On the CM system, reshaping entails actual data movement — most often with **send** communication — 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, when compiled with the **-nof90syntax** switch (default) is:

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

When compiled with the **-f90syntax** switch, the keyword **SHAPE** is used instead of the equivalent **MOLD**, and the argument order is changed, in order to make **RESHAPE** Fortran 90-compatible:

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

The Target Array

The **MOLD** argument specifies the target shape; it is a vector of positive integers, each indicating the extent of a target dimension. (Since each vector element corresponds to an array dimension, the vector's length is limited to

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

31.) For example, the following calls reshape a matrix **A(100,100)** in various ways:

```
B = RESHAPE( MOLD=[10000], SOURCE=A ) !1 dimension
C = RESHAPE( MOLD=[10,10,100], SOURCE=A ) !3 dimensions
D = RESHAPE( MOLD=[10,10,10,10], SOURCE=A ) !4 dimensions
```

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

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

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

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

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

Padding the Target Array

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

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

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

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

Permuting the Axes

The optional **ORDER** argument is used to change the order in which the target dimensions are filled. The default order is the vector [1:*n*]; the

Copyright © 1989–1994 Thinking Machines Corporation

alternative order for a 2-dimensional mold would be [2,1], with the following effect:

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

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

NOTE: An alternative to **RESHAPE** that does not require data movement is the array aliasing package described in the *CM Fortran Libraries Reference Manual.* Aliasing is a powerful but potentially dangerous tool that should be used only if you know the exact layout of the array to be reshaped.

9.3.2 Merging Arrays

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

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

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

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

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

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

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

Copyright © 1989–1994 Thinking Machines Corporation

Alone among the transformational intrinsic functions, **MERGE** need not perform implicit data motion. If its three conformable arguments and the destination array are whole arrays, their corresponding elements are already lined up in the same processors (unless the program specifies otherwise with a compiler directive). The behavior of **MERGE** is the same as the behavior of a **WHERE** construct (described in Chapter 5):

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

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

9.3.3 Spreading an Array

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

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

For example, if the source array is the vector

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

Then,

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

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

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

Copyright © 1989–1994 Thinking Machines Corporation

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

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

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

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

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

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

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

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

9.4 Array Multiplication

The array multiplication functions are **DOT_PRODUCT** 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 = DOT_PRODUCT( [ 1,2,3 ], [ 2,3,4 ] )       ! I = 20
```

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

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

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

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 a separate volume.

NOTE: CM Fortran's **MATMUL** function gets much better performance if the program is linked with the CMSSL library:

```
% cmf myprogram.fcm -lcmssl
```


Copyright © 1989–1994 Thinking Machines Corporation