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.

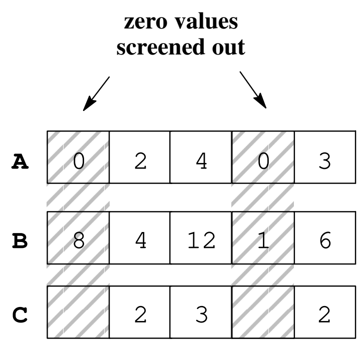
The **`FORALL`** statement and the array section syntax are especially useful for indicating data motion, such as shifting array elements in a grid pattern, arbitrary permutations and indexing, and parallel-prefix ("scan") operations.

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 and (beginning with Version 2.0) nested **`WHERE`** statements or constructs; it cannot contain 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)**.

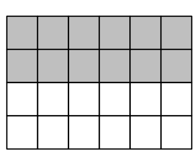
Triplet Examples

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

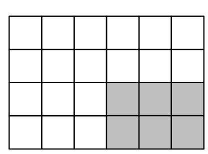
```
V(1:5)         ! first five elements
V(6:10)        ! last five elements
V(10:1:-1)     ! all ten elements in reverse order
V(1:10:2)      ! first, third, fifth, seventh, ninth
V(10:1:-2)     ! tenth, eighth, sixth, fourth, second
```

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)**:

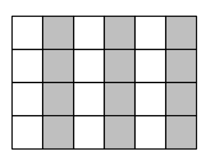
M(1:2,:)



M(3:4,4:6)



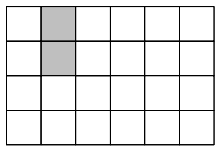
M(:,2:6:2)



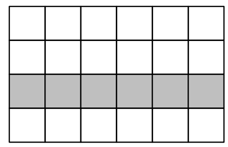
In these examples the section has the same rank (number of dimensions) as its *parent* array, although its size and shape are different. To get a section of lower rank than its parent, you can replace one or more of the triplets with a Fortran

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

M(1:2,2)



M(3,:)



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)
```

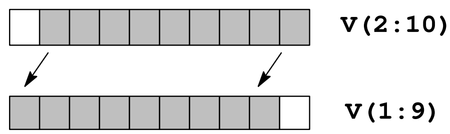
Any array reference that uses triplet notation is a Fortran 90 array reference and thus causes the parent array to be allocated on the CM.

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.)

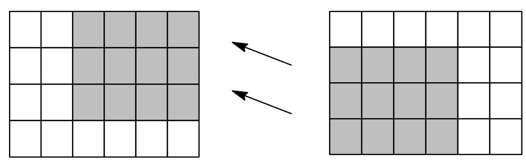
For example, to shift vector values to the left:

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



To shift data on more than one dimension:

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

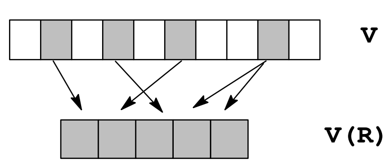


Vector-Valued Subscripts

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.)

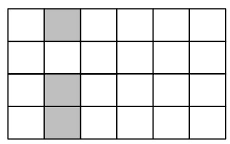
For example, if **`V`** is a vector of length 10 and **`P`** is a permutation of the integers from 1 to 10, then **`V = V(P)`** applies this permutation to the values in **`V`**. The statement **`V(P) = V`** applies the inverse permutation.

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:

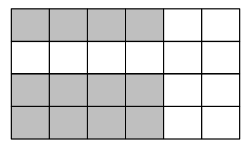


Vector-valued subscripts can be mixed with triplet subscripts and with Fortran 77 scalar subscripts in an array reference. The rank of the resulting array section is the number of dimensions indicated with either of the Fortran 90 subscripts, not counting any with scalar subscripts. For example, given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

`M(Q,2)`



`M(Q,1:4)`



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

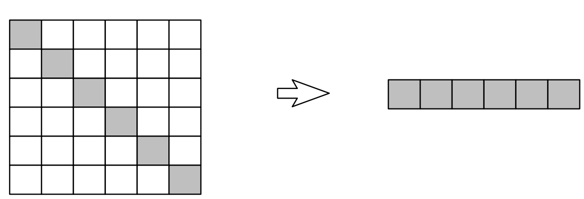
FORALL is a powerful feature for expressing data motion. It can mimic the behavior of other CM Fortran features, such as array sections and many of the intrinsic functions shown in the next chapter. Its real value, however, is in expressing patterns of data motion that are otherwise difficult to express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

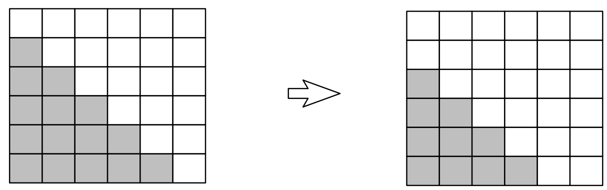
FORALL can also operate, in parallel, on irregularly shaped parts of an array. For example, to extract the diagonal elements of a matrix and assign them to a vector:

```
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 Utility Library Reference Manual*, allows you to work around any restrictions on the parallel execution of **FORALL** statements.