Chapter 5

Selecting Array Elements

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

- By value: Conditional (or *masked)* operations include or exclude elements depending on their values.
- By position: Operations on an *array section* affect only a subset of elements specified by their positions within the parent array.

Another method of selecting subarrays, the **FORALL** statement, can select elements both by value and by position. **FORALL** is described in Chapter 10. As with whole arrays, the language references all the elements of an array section or a masked array at once, and the Connection Machine system processes the elements in parallel.

5.1 Conditional Operations

Operations that are conditional on the element values of (CM) array objects include

- the **WHERE** statement and construct, which make an array assignment conditional
- CM Fortran intrinsic functions that take a **MASK** argument
- front-end (Fortran 77) conditionals or loop constructs that use a scalar value returned from the CM as a control specification

Copyright © 1989–1994 Thinking Machines Corporation

5.1.1 The WHERE Statement

The CM Fortran **WHERE** statement is similar to the Fortran 77 **IF** statement. The **WHERE** statement tests array elements (in parallel) and then performs an assignment of those that meet the specified condition. The format is

WHERE (*mask-expression* **)** *array-assignment*

For example, to avoid division by zero in an elemental array operation:

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

When executing this statement, the system creates a logical array containing the results of the relational operation **.NE.** It uses this array to mask the elements of **A** that contain 0 values, while the remaining elements are computed upon (see Figure 16). **WHERE** is thus called a *masked array assignment.*

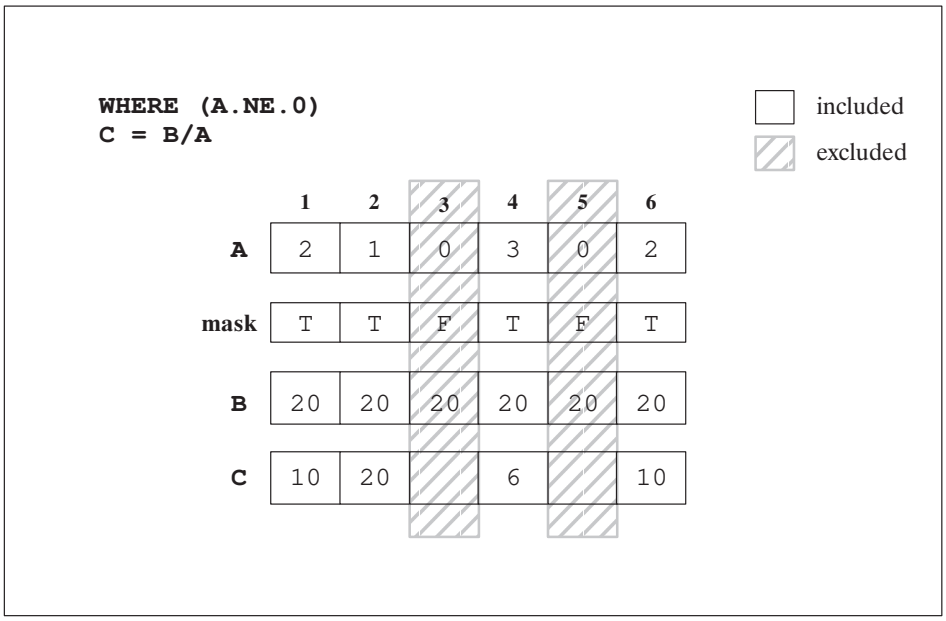


Figure 16. A masked array assignment.

Notice that arrays can be masked by an expression that does not reference them, but instead references another *conformable* array:

```
WHERE ( A.NE.0 ) C = B**2
```


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

5.1.2 The WHERE Construct

Like the **IF** statement, the **WHERE** statement is expanded into a construct with the **END WHERE** statement and an optional **ELSEWHERE**. The **WHERE** construct can contain array assignments and nested **WHERE** statements or constructs, but not **CALL** statements. A simple **WHERE** construct is

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

When executing this construct, the system computes a logical mask to screen out negative values of **A** from the **SQRT** operation. It then inverts the mask, reversing the true and false values, to include only the elements that have negative values for **A**. The corresponding elements of **B** are then assigned.

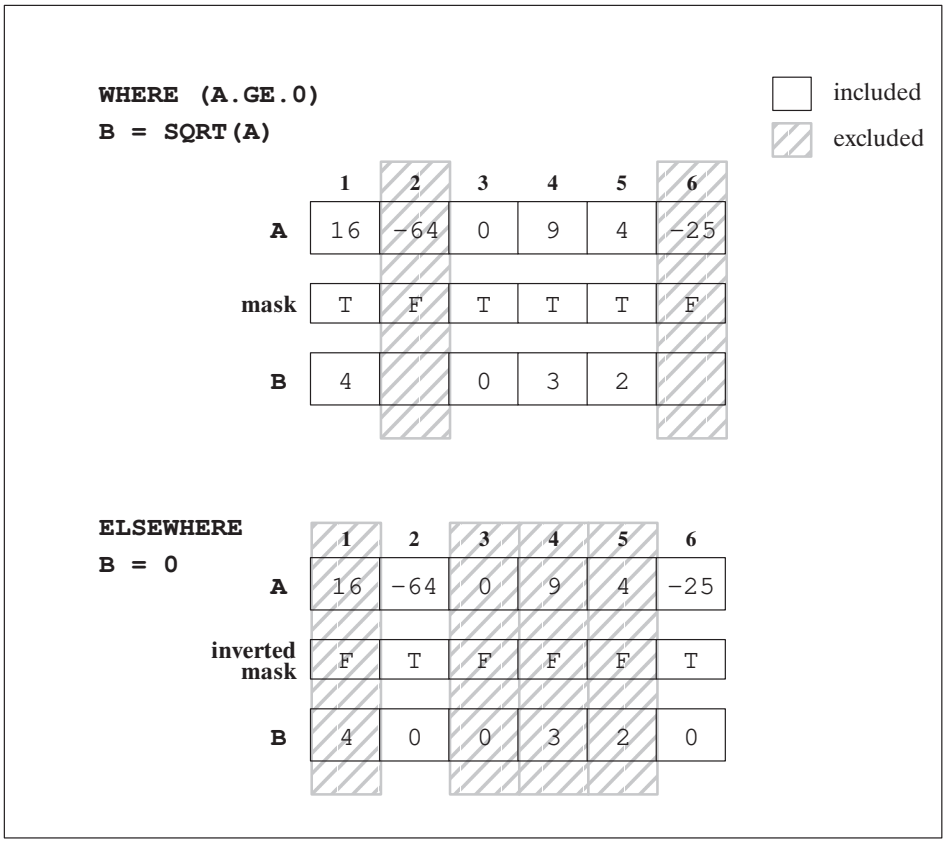


Figure 17. Inversion of the mask array in a WHERE–ELSEWHERE construct.

Copyright © 1989–1994 Thinking Machines Corporation

5.1.3 Nested WHERE Constructs

The body of a **WHERE** construct can contain

- **WHERE** assignments
- **WHERE** statements
- other **WHERE** constructs

When used with the **ELSEWHERE** statement, the capability of nesting parallel conditionals provides a great deal of flexibility in operating on different subsets of an array's elements during a series of assignments.

For example, the following form applies a second mask expression to the subset *excluded* by the first mask:

```
WHERE (mask-expr1)
    assignment-stmt
ELSEWHERE (mask-expr2)
    assignment-stmt
END WHERE
```

This form is evaluated as if it were written like this:

```
WHERE (mask-expr1)
    assignment-stmt
ELSEWHERE
    WHERE (mask-expr2)
        assignment-stmt
    END WHERE
END WHERE
```

Another form applies a second mask expression to the subset that is *selected* by the first mask:

```
WHERE (mask-expr1)
   assignment-stmt
   WHERE(mask-expr2)
      assignment-stmt
   END WHERE
   assignment-stmt
END WHERE
```


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

This form is evaluated as if it were written like this:

```
TEMP1 = mask-expr1
WHERE(TEMP1)
  assignment-stmt
END WHERE
TEMP2 = mask-expr2
WHERE(TEMP1 .AND. TEMP2)
  assignment-stmt
END WHERE
WHERE (TEMP1)
  assignment-stmt
END WHERE
```

Temporaries are used in this evaluation in case the mask expressions are changed by an assignment in the construct. The temporaries guarantee that the original mask expression values are used.

In cases where none of the assignments changes a mask expression, you can picture the evaluation of this form of nested **WHERE** statements more straightforwardly as follows. A construct written as

```
WHERE (mask-expr1)
    assignment-stmt
    WHERE (mask-expr2)
        assignment-stmt
    END WHERE
    assignment-stmt
END WHERE
```

is evaluated as if it were written like this:

```
WHERE (mask-expr1)
    assignment-stmt
END WHERE
WHERE (mask-expr1 .AND. mask-expr2)
    assignment-stmt
END WHERE
WHERE (mask-expr1)
    assignment-stmt
END WHERE
```


Copyright © 1989–1994 Thinking Machines Corporation

5.1.4 Conditional Intrinsic Functions

Many of the array-processing intrinsic functions (described in Chapter 9) perform a masking operation similar to the action of the **WHERE** statement. With these functions, the mask is supplied as an argument. For example, to take the product of the non-zero values of array **A**, invoke the function **PRODUCT** with a mask that eliminates the zero values:

```
SCALAR_PRODUCT_OF_A = PRODUCT( A, MASK=A.NE.0 )
```

If a masked intrinsic function is used within a **WHERE** statement, the two masks are completely separate. The **WHERE** mask has no effect on the evaluation of the function reference; the function's mask argument has no effect on the **WHERE** assignment. For example:

```
WHERE ( A.LE.10 ) B = A + PRODUCT( A, MASK=A.NE.0 )
```

The function reference in this example computes the product of **A**'s values, excluding any zero values but *not* excluding values over 10. This scalar product is then replicated to conform in shape to array **A** and added (elementally) to any elements that are less than or equal to 10 — including any zero values. Thus:

- An array element of 12 would be included in the computed product but *not* included in the addition and assignment operations; it is excluded by the **WHERE** mask but not by the function's mask argument.
- An array element of 0 would *not* be included in the product but would be included in the addition and assignment; it is excluded by the function's mask argument but not by the **WHERE** mask.

The behavior of masked intrinsic functions within a **WHERE** statement is a CM Fortran extension to the standard language. Fortran 90 permits *only* elemental functions, such as **SQRT**, in **WHERE** statements. None of the elemental functions takes a mask argument.

NOTE: The behavior of the intrinsic function **MERGE** is similar to that of the **WHERE** construct and may give better performance. See Chapter 9 for information on **MERGE**.

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

5.1.5 Front-End Conditionals

A scalar value derived from a CM array can be used in a serial (front-end) construction to control a conditional or iterative operation. The scalar can be either

- the result of scalar subscripting of a CM array, such as **CM_ARRAY(3,10)**
- the scalar result returned by an intrinsic reduction function, such as **ANY** or **SUM** (see Chapter 9)

For example, this **IF** construct uses the logical result of the reduction function **ANY** as its test condition. It branches according to whether any of the elements of the array object **A** is 0.

```
IF (ANY ( A.EQ.0) ) THEN
    PRINT *, 'Array A contains a 0.'
ELSE
    PRINT *, 'Array A does not contain a 0.'
END IF
```

A somewhat more elaborate operation combines a **WHERE** construct on the CM with a front-end control construct, such as **DO**, **DO WHILE**, **DO TIMES**, or **CASE**. The purpose is to perform a parallel operation repeatedly until a condition is met for each element. The **WHERE** construct masks out the elements that have met the condition, and the front-end loop construct checks the result of a reduction function after each iteration to determine whether any elements remain selected.

For example, the following program uses the scalar result of **ANY** to control a serial **DO WHILE** loop. In this program, the CM computes the floor of the base-2 logarithm of each element of array **A**. Each element of **A** is divided by 2 repeatedly until the element becomes 1; the number of divisions (the iteration count) accumulates in **B**. When the **A** value becomes 1, it is masked out of further computation by the **WHERE** construct. The serial **DO WHILE** loop continues until no elements are left unmasked.

Copyright © 1989–1994 Thinking Machines Corporation

```
      PROGRAM BASE2_LOG

      INTEGER N
      PARAMETER (N = 256)
      INTEGER, DIMENSION (N,N) :: A,B
      IMPLICIT NONE

C     Initialize A with non-zero random numbers

      CALL CMF_RANDOM( A, 1000)    ! A CMF utility routine
      A = A + 1                    ! All A.GT.0

C     Compute B = log_2(A)

      B = 0
      DO WHILE ( ANY( A.GT.1 ))    ! Front-end loop
         WHERE ( A.GT.1 )          ! CM conditional
            A = A/2
            B = B+1
         END WHERE
      END DO

      STOP
      END
```


Copyright © 1989–1994 Thinking Machines Corporation

5.2 Array Sections

Fortran 90 defines syntax for specifying *some* or *all* of an array's elements in a particular reference. This *triplet* notation resembles a **DO** loop control specification:

lower-bound : *upper-bound* : *stride*

For example, **A(1:4:1)** references every element of an array declared as **A(4)**, while **B(1:3:1, 1:5:1)** references every element of a matrix declared as **B(3,5)**. The lower and upper bounds default to the declared bounds of the array; the stride (increment) defaults to 1. Thus, these array references default to, simply, **A** and **B**.

As you might expect, you can specify bounds other than the array's declared bounds, or a stride other than 1, to indicate a subset of array elements. A subset of elements is called an *array section;* the array from which a section is specified is called the *parent* array.

5.2.1 Triplet Examples

Given a 10-element vector **A**, the expression

```
A(1:5)
```

refers to its first five elements (the stride defaults to 1); whereas the expression

```
A(1:10:2)
```

refers to elements 1, 3, 5, 7, and 9 in that order.

Negative Strides

Negative strides count down from the first value specified to the second. Thus, the expression

```
A(10:2:-2)
```

specifies the elements 10, 8, 6, 4, 2. However,

```
A(10:2:2)
```

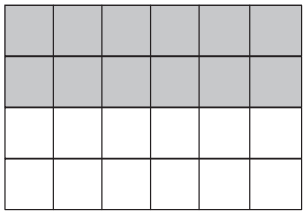

Copyright © 1989–1994 Thinking Machines Corporation

specifies a null sequence of subscript values (an empty array) because the first value is greater than the second one, but the stride is positive.

Multidimensional Parent Arrays

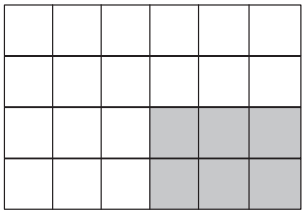
Sections of multidimensional parent arrays are specified with a triplet for each dimension, separated by commas. For example, to specify the upper half of a 4 x 6 matrix **B**:

```
B(1:2, 1:6)
```



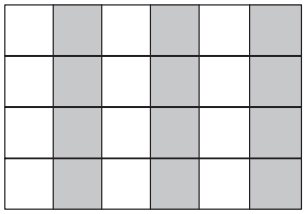
And, to specify the lower right quadrant:

```
B(3:4, 4:6)
```



Finally, to specify only the even columns:

```
B(1:4, 2:6:2)
```

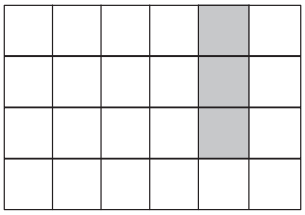


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

Triplets and Scalar Subscripts

Triplet subscripts can be intermixed in an array reference with Fortran 77–style scalar subscripts. For example, given the 4 x 6 matrix **B**, the following specifies a vector-shaped section that contains the first three elements in column 5:

```
B(1:3, 5)
```



Notice that the rank of this section is not the same as the rank of its parent array. The rank of an array section is the number of dimensions that are specified with a Fortran 90 subscript, not counting any that are specified with Fortran 77 (scalar) subscripts. As another example, the following specifies a 2-dimensional section of a 4-dimensional array **C**:

```
C(1:10, 1, 1:50:2, 3)
```

Default Triplet Forms

Any item of the triplet, or the triplet for any dimension, can be allowed to default, as long as placeholder colons and commas are retained to avoid ambiguity. (The stride in particular is often omitted.) For the sake of clarity, this manual usually specifies both bounds in a triplet, but the following sets of forms are equivalent when referencing sections of an array declared as **D(8,10)**:

```
D(1:4:1, 1:5:1)     <=>  D(1:4, 1:5)     <=>  D(:4, :5)

D(5:8:1, 1:10:1)    <=>  D(5:8, 1:10)    <=>  D(5:8, :)

D(1:3:1, 9)         <=>  D(1:3, 9)       <=>  D(:3, 9)
```


Copyright © 1989–1994 Thinking Machines Corporation

5.2.2 Using Array Sections

An array section can be used as an operand or argument in the same way that whole arrays are used, and the CM system operates on all the specified elements in parallel.

Conformable Sections

When an array section is used in an expression or assignment with other array sections or whole arrays, they must all be of the same shape. Scalars can, of course, be intermixed freely in such expressions.

For example, given two 8 x 10 matrices **D** and **E**, the following statement adds the corresponding elements in the upper left quadrant. Notice that the two sections specified are identical.

```
D(1:4, 1:5) = D(1:4, 1:5) + E(1:4, 1:5)
```

Sections and Communication

Operations on array sections do not require interprocessor communication if the array sections are made up of corresponding elements of conformable parent arrays. For example, the arrays

```
REAL, DIMENSION (20,20) :: A,B,C
```

are allocated among processors in the same fashion. When you select the same elements from all the arrays for an operation, the data items are already in the right processors for the operation to proceed. For example, this is an in-processor operation:

```
A(1:10, 2:20:2) = B(1:10, 2:20:2) + C(1:10, 2:20:2)
```

In contrast, operations on array sections that are of the same shape but are not the corresponding elements of their parent arrays do require interprocessor communication. For example:

```
A(1:10, :) = B(1:10, :) + C(11:20, :)
```

The three parent arrays are all distributed in the same way among processors. However, the elements selected from array **C** are not in the same processors as those taken from **A** and **B**. Before the system can perform the operation, it must allocate temporary storage for the section drawn from **C**

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

in the processors that contain the specified elements of **A** and **B**. Naturally, this (transparent) interprocessor communication adds to the program's execution time.

Array sections that are selected from parents of different shape are typically not located in the same set of processors because the parents were not distributed in the same fashion to begin with. The system must always move the specified array elements into the appropriate processors before it can perform an operation. For example, this assignment requires interprocessor communication:

```
REAL, DIMENSION (5) :: D
REAL, DIMENSION (10) :: E

E(6:10) = D
```

CM Fortran provides a compiler directive, **LAYOUT**, that you can use to minimize interprocessor communication when assigning arrays and array sections. See Chapter 12.

Copyright © 1989–1994 Thinking Machines Corporation