Chapter 8

Data Movement

CM Fortran, like Fortran 90, provides new syntax and intrinsic functions for rearranging the elements of arrays:

- array sections
- vector-valued subscripts
- intrinsic functions **CSHIFT**, **EOSHIFT**, and **TRANSPOSE**

The **FORALL** statement, deferred until Chapter 10, also expresses data movement. Other intrinsic functions perform data movement implicitly in the course of some other kind of array transformation. These functions are presented in Chapter 9.

Dimensional shifts, permutations, and transpositions of array elements normally require the compiler to generate interprocessor communication routines on the Connection Machine system. However, some array section assignments on *serial dimensions* are handled more efficiently with local memory accesses instead of communication routines. A serial dimension is one that is constrained by a **LAYOUT** directive to be allocated entirely within the memory of each processor, as described in Chapter 12.

This chapter focuses on the semantics of the various CM Fortran constructions that express data movement. Consult the *CM-5 CM Fortran Performance Guide* to evaluate the comparative performance of these language constructions.

Copyright © 1989–1994 Thinking Machines Corporation

8.1 Array Sections

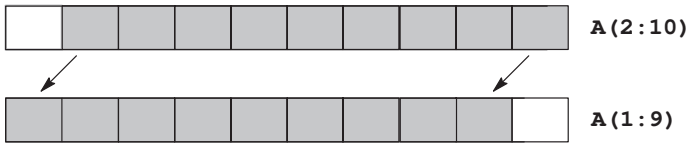
This section describes the use of array sections to reposition the values of arrays.

8.1.1 Dimensional Shifts

Array sections, in conjunction with assignments, are useful for moving data in regular grid patterns. An example is a "dimensional shift," where element values are all shifted some number of positions along an array dimension. The following assignment statement shifts the elements of a 10-element vector **A** by one position:

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

as illustrated by:



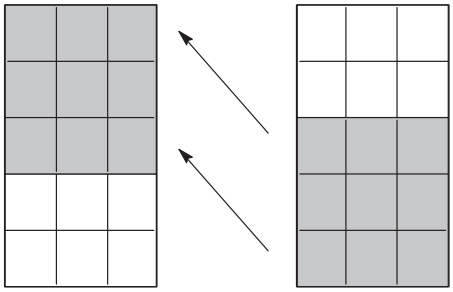
Notice that — as with any array assignment — the two array sections must be conformable. Thus, a number of elements at least equal to the shift offset must be excluded from the computation. In this example, a section that excludes the first element of the parent array is assigned to a section that excludes the last element.

Extending this example to a 2-dimensional array is straightforward. Suppose that **B** is a 5 x 3 matrix and you want to shift by two positions on the first dimension:

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


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

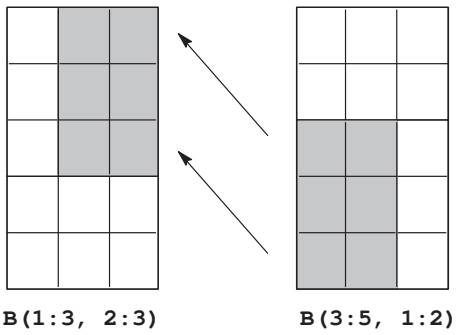
as illustrated by:



Similarly, to shift along more than one dimension at a time:

```
B(1:3, 2:3) = B(3:5, 1:2)
```

as illustrated by:



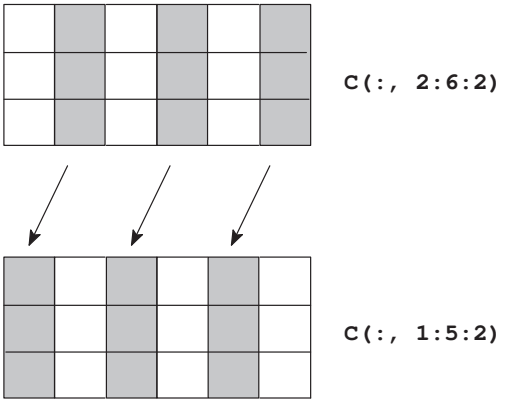
Copyright © 1989–1994 Thinking Machines Corporation

8.1.2 Shifting Noncontiguous Sections

Specifying a stride greater than 1 yields a noncontiguous array section, but the mechanics of shifting element values along a dimension are the same. For example, this statement copies the values from the even columns of a 3 x 6 array **C** into the elements in the odd columns in each row:

```
C(:, 1:5:2) = C (:, 2:6:2)
```

as illustrated by:



The before-and-after values in **C** might be:

$$C = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 3 & 1 & 3 & 1 & 3 \end{bmatrix} \Rightarrow C = \begin{bmatrix} 2 & 2 & 4 & 4 & 6 & 6 \\ 2 & 2 & 4 & 4 & 6 & 6 \\ 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

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

8.1.3 Permutations

Array sections can be used for data movement other than dimensional shifts by mixing positive and negative strides or by mixing unequal strides in the assignment statement. For example, to reverse the elements of a 10-element vector **D**:

```
D = D(10:1:-1)
```

Similarly, to reverse the columns of values in an *n* x 6 matrix **E**:

```
E  =  E(:, 6:1:-1)
```

And, to replace the left half of **E** with the values in the even columns:

```
E(:, 1:3)  =  E(:, 2:6:2)
```

For the dimensional shifts shown previously, the compile generates NEWS communication, which optimizes the special case of grid communication. The system cannot use NEWS communication in cases where the data motion specified is not a dimensional shift. In the examples shown here, the values move different distances to reach their destination positions. Any data motion where the pattern of movement is not the same for all selected elements must use the less-efficient **send** routines when the movement is across processors.

8.2 Vector-Valued Subscripts

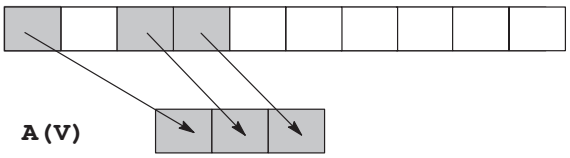
A special form of array section uses a *vector-valued subscript*, an integer vector that serves as a sequence of index values in an array reference. Since the index vector need not be ordered — that is, there is no fixed stride — this syntax permits an arbitrary selection of array values along a dimension. A vector-valued subscript is essentially an indirection vector; it is particularly useful for mapping unstructured problems onto a rectangular grid.

Like other array sections, array sections specified with vector-valued subscripts can be used in conjunction with assignment statements to perform data movement.

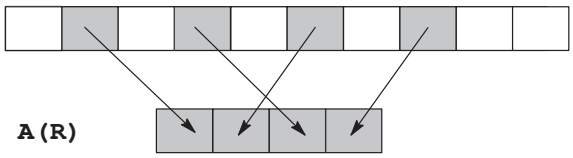
Copyright © 1989–1994 Thinking Machines Corporation

8.2.1 Examples

If **A** is a vector of length 10 and **V** is the vector [1,3,4], then **A(V)** is a vector of length 3 whose elements are **A(1)**, **A(3)**, and **A(4)**.



Similarly, if **R** is the vector [2,6,4,8], then **A(R)** is a vector of length 4 whose elements are **A(2)**, **A(6)**, **A(4)**, and **A(8)**.



Array sections specified with vector-valued subscripts always involve data movement, even in the absence of a statement assigning the values to different positions. That is, the system creates a temporary array of the same length as the index vector and moves the specified values of the parent array into the temporary before performing any operation on them. (Because the indices are not necessarily in their natural order, the system cannot simply ignore the elements whose indices are excluded, as it does with array sections.) As a result, even a statement like the following requires the **send** communication:

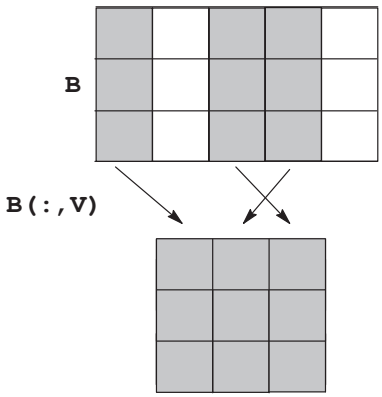
```
A(V) = A(V) + 1
```

You can avoid communication when performing indirect addressing on serial dimensions by using the **FORALL** statement, which is specially optimized for this case (see Chapter 10).

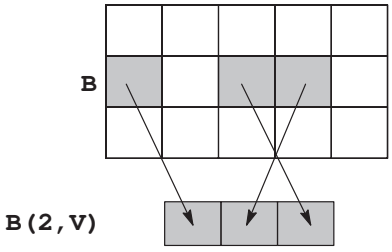
Vector-valued subscripts can be intermixed in a reference with triplet subscripts and scalar subscripts. For example, given a 3 x 5 array **B** and the

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

vector `V = [1,4,3]`, then `B(:, V)` specifies a 2-dimensional array section that consists of the first, fourth, and third columns of `B`, in that order:



The section `B(2,V)` specifies a 1-dimensional section of length 3 consisting of the first, fourth, and third values in the second row of `B`.



Notice that the rank of an array section equals the number of dimensions that are specified with either triplet subscripts or vector-valued subscripts, not counting any dimensions that are specified with scalar subscripts. Thus, `B(:,V)` in this example is smaller than the parent array `B` but still 2-dimensional. The section `B(2,V)`, however, has only one Fortran 90−style subscript and is therefore 1-dimensional.

Copyright © 1989−1994 Thinking Machines Corporation

8.2.2 Permutations

One use of vector-valued subscripts is to rearrange array elements. For example, if **A** and **B** are 10-element vectors, and vector **P** is a permutation of the numbers 1 to 10, the statement

```
A = B(P)
```

rearranges the values of **B** according to the permutation specified by **P** and assigns them to **A**. These actions are equivalent to the following Fortran 77 operation:

```
   DO 30 I = 1,10
       A(I) = B( P(I) )
30 CONTINUE
```

8.2.3 Assigning Permuted Sections

Like any array object, a section specified with a vector-valued subscript must be conformable with other array objects used in the same expression or in an assignment.

Thus, when the index vector is on the right-hand side of an assignment statement, it must be conformable with the destination array on the left. The source array can be any size or shape, since the index vector is specifying a section of the source array that is appropriately shaped for the assignment.

For example, assume the following vectors:

```
SOURCE = [ 10, 20, 30, 40, 50, 60, 70 ]
INDEX  = [ 3, 1, 4, 6 ]
```

Executing the statement

```
DEST = SOURCE(INDEX)
```

assigns the following values to **DEST**:

```
DEST   = [ 30, 10, 40, 60 ]
```

Similarly, when the index vector is on the left, as in the statement

```
DEST_2(INDEX_2) = SOURCE_2
```


Copyright © 1989–1994 Thinking Machines Corporation

the index vector must be conformable with the source array. The destination array can be any size or shape, since the index vector is specifying a section of it that is appropriate for the assignment.

8.2.4 Replicating Data

Vector-valued subscripts can go beyond simply rearranging data, since an index number may appear more than once in the index vector (as long as the vector appears on the right-hand side of the assignment). For example, assume the following vectors:

```
SOURCE_3 = [ 15, 20, 25, 30, 35, 40, 45 ]
INDEX_3  = [ 3, 1, 4, 4 ]
```

Notice that index 4 is specified twice in the index vector. Executing the statement

```
DEST_3 = SOURCE_3(INDEX_3)
```

assigns the following values to **DEST_3**:

```
[ 25, 15, 30, 30 ]
```

The index numbers may not be replicated when the index vector appears on the left-hand side of an assignment, since the language does not define the effect of assigning more than one value to an array element.

Copyright © 1989–1994 Thinking Machines Corporation

8.3 Intrinsic Shift Functions

CM Fortran provides two intrinsic functions for shifting array elements in regular patterns along array dimensions. Their actions differ from the dimensional shifts performed with array sections in that the functions deal with the boundary elements.

- **CSHIFT**, for "circular shift," causes values that move off the edge of the array to reappear at the opposite edge.

  ```
  CSHIFT( ARRAY, DIM, SHIFT )
  ```

  NOTE: When a program is compiled with the **-f90syntax** switch (non-default), the order of arguments to **CSHIFT** changes, making the function Fortran 90-compatible:

  ```
  CSHIFT( ARRAY, SHIFT, DIM )
  ```

  The use of keywords allows the arguments to appear in any order, regardless of the compilation switch.

- **EOSHIFT**, for "end-off shift," discards values that move off the edge of the array and moves some specified or default value into the positions vacated at the opposite edge.

  ```
  EOSHIFT( ARRAY, DIM, SHIFT [, BOUNDARY] )
  ```

  NOTE: When a program is compiled with the **-f90syntax** switch (non-default), the order of arguments to **EOSHIFT** changes, making the function Fortran 90-compatible:

  ```
  EOSHIFT( ARRAY, SHIFT, [BOUNDARY,] DIM )
  ```

  The use of keywords allows the arguments to appear in any order, regardless of the compilation switch.

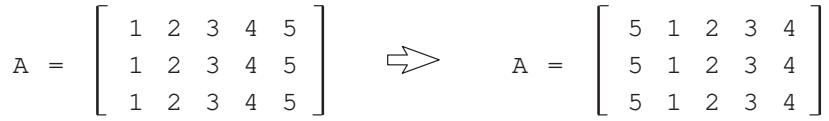
8.3.1 Using CSHIFT

Given a 3 x 5 array **A**, the statement

```
A = CSHIFT( A, DIM=2, SHIFT=-1 )
```

shifts the values in the second dimension of **A** (which you can picture as the columns, or as the values *in* the rows) by one column position in the negative direction, wrapping the right-most values around to the first column.

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



CSHIFT(ARRAY, DIM, SHIFT)

The negative direction — meaning that element **A(I,J)** gets the value of **A(I,J-1)** — is described here as "shifting to the right." An alternative view of the same action is that each element gets the value of its neighbor to the left, with wrapping at the edge.

Notice in the example above that CM Fortran supports the use of predefined keywords in intrinsic function calls. Using the keywords makes the arguments position-independent. We especially recommend using the keywords with the shift functions, since the order of arguments in CM Fortran reflects an earlier draft of Fortran 90, not the standard.

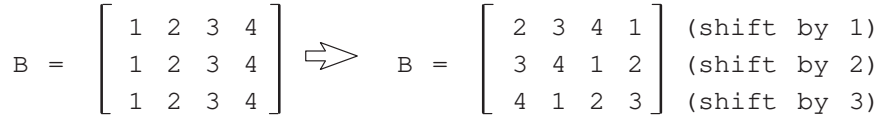
Calls to **CSHIFT** can be nested to access diagonal neighbors. For example, to have each element of a matrix get the sum of its four diagonal neighbors:

```
 A = CSHIFT( CSHIFT( A,DIM=1,SHIFT=1 ),  DIM=2,SHIFT=1) +
$    CSHIFT( CSHIFT( A,DIM=1,SHIFT=1 ),  DIM=2,SHIFT=-1)+
$    CSHIFT( CSHIFT( A,DIM=1,SHIFT=-1 ), DIM=2,SHIFT=1) +
$    CSHIFT( CSHIFT( A,DIM=1,SHIFT=-1 ), DIM=2,SHIFT=-1)
```

The **SHIFT** argument to **CSHIFT** can also be an array, indicating a possibly different shift distance for each row or column of the argument array. The **SHIFT** array must be of rank one less than the argument array. For 2-dimensional argument arrays, the **SHIFT** argument can be any vector, including an array constructor (see Chapter 3). For example, if **B** is a 2-dimensional array, then

```
B = CSHIFT( B, DIM=2, SHIFT=[1,2,3] )
```

has the following effect on **B**:



Copyright © 1989–1994 Thinking Machines Corporation

The **DIM** argument to **CSHIFT** can be either a constant expression or a variable — as long as **SHIFT** is a scalar. If **SHIFT** is an array, however, **DIM** must be a constant expression.

For example, all the following combinations of constant and variable **DIM** arguments and scalar and array **SHIFT** arguments are supported:

```
A = CSHIFT( A, DIM=2, SHIFT=1 )
A = CSHIFT( A, DIM=J, SHIFT=1 )
A = CSHIFT( A, DIM=2, SHIFT=[1,2,3] )
```

The illegal combination is variable **DIM** with array **SHIFT**:

```
A = CSHIFT( A, DIM=J, SHIFT=[1,2,3] ) ! Not supported
```

8.3.2 Using EOSHIFT

EOSHIFT is similar to **CSHIFT** except for its treatment of boundary elements. This function takes an optional **BOUNDARY** argument that specifies the value to move into the elements that are vacated by the shift operation. The default is zero or **.FALSE.**, depending on the type of array.

8.3.3 Efficiency Note

Assignments of array sections are often faster than the intrinsic shift functions, particularly on serial dimensions. On serial dimensions, array section assignments often translate to local memory accesses, whereas the shift functions always translate to run-time communication routines.

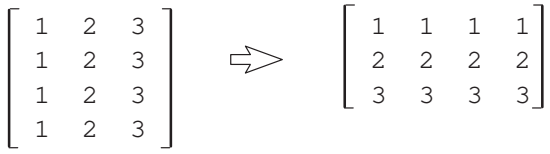
8.4 The TRANSPOSE Function

The intrinsic function **TRANSPOSE** transposes the axes of a matrix, creating a matrix of shape (**M,N**) from an argument matrix of shape (**N,M**). For example,

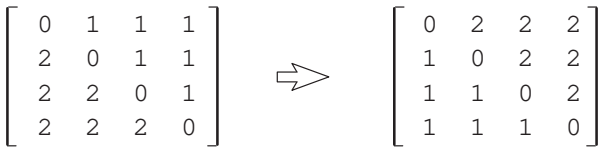
```
MATRIX_2 = TRANSPOSE( MATRIX_1 )
```


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

has the following effect when **MATRIX_1** is 4 x 3:



If the argument matrix is a square, the effect is to flip its values over the diagonal. For example, calling **TRANSPOSE** on a 4 x 4 matrix has the following effect:



Notice that the **TRANSPOSE** function moves the various element values different distances and directions. To perform this pattern of data movement the system necessarily uses **send** communication. The **TRANSPOSE** function therefore does not execute as fast as the dimensional shifts shown above; its speed is comparable to that of permutations.

Copyright © 1989–1994 Thinking Machines Corporation