Chapter 10

The FORALL Statement

A **FORALL** statement describes a collection of assignments to designated array elements, to be executed in undefined order but as if simultaneously. This *elemental array assignment* is an extremely powerful way to express location-dependent action. It can be used for:

- Dynamically initializing arrays with sequences of values.
- Selecting subarrays, both by position and by value.
- Data motion in patterns that are otherwise difficult to express in CM Fortran as parallel operations. These include *parallel prefix* (cumulative) computations along array dimensions and certain operations on irregularly shaped data.

FORALL provides many of the same capabilities as **DO** constructs, including those with nested **DO**'s and those with embedded **IF** statements. However, since the individual assignments occur in an undefined order, **FORALL** statements can be executed in parallel on the CM system.

Although not a part of standard Fortran 90, **FORALL** is included in High Performance Fortran because of its particular expressive power. In some ways, **FORALL** is redundant with other CM Fortran features, such as assignment of array sections specified with triplet notation or vector-valued subscripts. However, **FORALL** is the only feature that can express certain very efficient CM operations, such as parallel prefix operations.

After introducing the syntax of the **FORALL** statement, this chapter examines its execution model — particularly the significance of *as if simultaneous* execution — and concludes with some examples of using **FORALL** to express various patterns of data motion.

Copyright © 1989–1994 Thinking Machines Corporation

10.1 Syntax

FORALL uses triplet notation to specify one or more sequences of integer values and associates each with a symbolic name. The names are then used in expressions in an assignment statement, where they can indicate either array indices or values. The basic format is:

FORALL (*triplet-spec*[*s*]) *assignment*

For example, to seed the array **IDENTITY(N)** with "self-addresses," that is, to assign each element its own index value:

```
FORALL ( I=1:N )  IDENTITY(I) = I
```

Similarly, to clear every tenth element of vector **V(N)**:

```
FORALL ( I=1:N:10 )  V(I) = 0
```

In simple examples like these, **FORALL** is redundant with other CM Fortran features. These statements can also be expressed as CM array assignments, using an array constructor in the first case and a triplet subscript in the second:

```
IDENTITY = [ 1:N ]

V( 1:N:10 ) = 0
```

However, **FORALL** is the feature of choice for more elaborate assignments, such as those involving multidimensional arrays and combinations of symbolic names. For example, the following statement initializes 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 )
```

Notice that the definition of the symbolic names in a **FORALL** statement resembles the definition of index variables in a **DO** construct. Where **DO** might define **I** as **I=1,N,10**, **FORALL** substitutes colons, **I=1:N:10**. This chapter adopts the term *index variable* to refer to a symbolic name defined in a **FORALL** triplet-spec.

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

10.1.1 The FORALL Assignment

The assignment in a **FORALL** statement is the same as the assignment statement in a **DO** construct. In CM Fortran, both **DO** and **FORALL** can assign individual array elements or array objects (sections).

FORALL assignments have certain restrictions: no left-hand-side element can be assigned more than one value, and, as with **DO** constructs, a function reference appearing anywhere in the assignment must not alter the value of an index variable.

A **FORALL** statement can have only one assignment. That is, there is no block **FORALL** construct and no embedding of **WHERE** or other CM Fortran statements.

The Target Elements

The left-hand side of a **FORALL** assignment uses the index variables to indicate the array indices of interest. It can do so either with scalar subscripts in the Fortran 77 manner,

```
FORALL ( I=1:N, J=1:M )  A(I,J) = expression
```

or with triplet subscripts in the Fortran 90 manner,

```
FORALL ( K=1:N )  A(K,:) = expression

FORALL ( K=1:N )  A(1:K, K:1:-1) = expression
```

When the left-hand side is specified with scalar subscripts, the target of each assignment is a single array element. When the left-hand side includes triplet subscripts, the target of each assignment is an array section (as described in Chapter 5).

In either case, the left-hand side must use *all* the index variables defined in the statement; array references without subscripts are not permitted. For example, the following assignments to vector **B** are *not* legal, the first because no index variables appear on the left and the second because **J** does not appear on the left:

```
FORALL ( I=1:N, J=1:M ) B = I+J        ! Error

FORALL ( I=1:N, J=1:M ) B(I) = J       ! Error
```


Copyright © 1989–1994 Thinking Machines Corporation

As in a **DO** construct, the index variables can be used either as primary expressions or as operands in an operation that specifies the target elements:

```
FORALL ( K=0:10 ) LOGS( 2**K:2**(K+1)-1 ) = K
```

The target elements specified need not constitute the whole array, of course. **FORALL** provides two ways to select a subarray by element position:

- By having the index variables refer to a sequence that covers only part of the corresponding array dimension:

```
DIMENSION A(100,100)
...
FORALL (I=1:50, J=1:50) A(I,J) = 0      ! Clear one quadrant

FORALL (K=2:100:2, L=5:100:5) A(K,L) = 0 ! Selected elements
```

- By indicating an array section on the left-hand side of the assignment:

```
DIMENSION B(10,10)
...
FORALL (M=1:10) B(M,:) = M ! Self-index the first dim.
```

The Source Values

The right-hand side of a **FORALL** assignment must provide no more than one value for each element specified on the left. The set of permissible expressions varies depending on whether the left-hand side is a series of single array elements (scalar subscripts) or a series of array sections (triplet subscripts).

If the target elements are specified with scalar subscripts, the right-hand side must be a scalar-valued expression. For example, given the three arrays shown, the following statements assign a scalar value to each successive element of vector **A**:

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

```
DIMENSION A(N),B(N)
DIMENSION C(N,M)

FORALL (I=1:N)  A(I) = 10    ! Constant expression
FORALL (I=1:N)  A(I) = I     ! Scalar variable
FORALL (I=1:N)  A(I) = B(I)  ! Array element
FORALL (I=1:N)  A(I) = SIN(I)  ! Elemental (scalar) function
FORALL (I=1:N)  A(I) = SUM(C(I,:))   ! Intrinsic (scalar)
FORALL (I=1:N)  A(I) = FUNC(B(I))    ! User func. (scalar)
```

If, on the other hand, the left-hand side of a **FORALL** assignment is specified with triplet subscripts, the target of each assignment is an array section rather than an individual element. In this case, the right-hand expression must conform to each of the target array sections. Thus, the right-hand side can be either:

- Any of the scalar-valued expressions listed above. Recall that a scalar is replicated to conform to any array.
- An array-valued expression of the same shape as a target array section.

For example, given the arrays shown, the following statements assign a vector to each successive row or column of matrix **A**. (Several of these options are illustrated below.)

```
DIMENSION A(N,M),B(N,M)
DIMENSION C(N)

FORALL (I=1:N)  A(I,:) = 10  ! Scalar extension (to rows)
FORALL (I=1:M)  A(:,I) = C   ! Array (to columns)
FORALL (I=1:N)  A(I,:) = B(I,:)   ! Array section
FORALL (I=1:N)  A(I,:) = [1:M]    ! Array constructor
FORALL (I=1:M)  A(:,I) = SIN(C)   ! Elemental intrinsic
FORALL (I=1:M)  A(:,I) = SUM(B,2) ! Intrinsic (array)
FORALL (I=1:N)  A(I,:) = FUNC(C)  ! User func. (array)
```


Copyright © 1989–1994 Thinking Machines Corporation

To illustrate some of these cases, assume that **A** is a 4 x 3 matrix. In the case of a scalar-valued right-hand expression,

```
FORALL ( I=1:3 ) A(:,I) = I
```

you can picture each of the three scalar values of **I** being promoted to a vector for (elemental) assignment to the corresponding column of array **A**. (Alternatively, you can picture the promotion of the scalars as spreading the whole vector sequence **I** on the first dimension of **A**.)

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

The effect is similar when the right-hand expression is an array constructor that conforms to the target section. In this example, a vector is assigned to each successive column. Since the right-hand side in this example is the same for every assignment, the effect is a spread operation on the second dimension of **A**.

```
FORALL ( I=1:3 ) A(:,I) = [ 1:4 ]
```

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

Similarly, consider an array-valued intrinsic function whose result is of the appropriate shape. The function reference **SUM(B,2)** returns a vector containing the sum of the elements in each row of matrix **B**. If **B**'s and **A**'s first dimensions (that is, their columns) are of the same length, the vector result of **SUM(B,2)** conforms to each of the columns of **A**.

```
FORALL (I=1:N)  A(:,I) = SUM( B, DIM=2 )
```

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{bmatrix} \Rightarrow A = \begin{bmatrix} 15 & 15 & 15 \\ 20 & 20 & 20 \\ 25 & 25 & 25 \\ 30 & 30 & 30 \end{bmatrix}$$

Again, the result is a spread operation because the right-hand expression is the same for each of the assignments. Some complicated patterns of data

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

movement occur when the right-hand side references the index variables and varies accordingly. See Section 10.3 for examples that use **FORALL** to specify these other patterns of data movement.

10.1.2 The FORALL Mask

A **FORALL** statement can contain a mask expression, which prevents certain elements from being assigned. The mask is a scalar-valued expression, placed inside the parentheses that enclose the triplet-specs:

FORALL **(** *triplet-spec*[*s*] [, *scalar-mask*] **)** *assignment*

The mask expression usually references one or more index variables, although it can be a constant expression. Its action is comparable to embedding an **IF** statement in a **DO** construct.

For example, to avoid division by zero in a **FORALL** statement:

```
FORALL ( I=1:N, J=1:M, A(I,J).NE.0.0 ) B(I,J) = 1.0/A(I,J)
```

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
```

Order of Execution

The mask is applied as the third or fourth step of a 4-step **FORALL** operation:

1. Compute the combinations of index variables.
2. Compute the mask value, true or false, for each combination of index variables.
3. Under the vector-units execution model, evaluate the right-hand expression. Under other models, evaluate the right-hand expression only for the combinations of index variables where the mask expression is true.
4. Perform the assignments (in unspecified order) only for the combinations of index variables where the mask expression is true.

Copyright © 1989–1994 Thinking Machines Corporation

Array-Valued Masks

The **FORALL** mask expression is always scalar-valued, even when the left-hand side of the assignment is an array section and the right is array-valued:

```
FORALL ( I=1:N, I.GT.5 ) B(I,:) = A(:,I) ! Legal: scalar mask

FORALL ( I=1:N, A.GT.5 ) B(I,:) = A(:,I) ! Error: array mask
```

However, array-valued masks can appear in intrinsic function references in the **FORALL** assignment. Notice that the two masks in the following statement are entirely independent of each other:

```
FORALL (I=1:N, I.NE.5) C(I) = SUM( D(:I,:I), MASK = D.GT.0 )
```

This example sums the positive elements in varying sections of array **D** and assigns the results to the successive elements of array **C**. The array-valued mask argument to **SUM** excludes the zero and negative elements of **D**; the scalar-valued **FORALL** mask excludes the fifth element of **C**.

10.2 Execution Model

This chapter has pointed out many ways in which a **FORALL** statement resembles a **DO** construct. The most significant difference between them is the order of execution of the individual assignments. Since the order is undefined, but as if simultaneous, the assignments can be simultaneous in fact when **FORALL** is executed by the CM.

10.2.1 As If Simultaneous

Any **FORALL** statement can be translated into a **DO** construct with equivalent effects. Consider a simple example:

```
FORALL ( I=1:N )  A(I) = I

DO I=1,N
    A(I) = I
END DO
```


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

Both these operations seed array **A** with its own index values. However, the semantics of the two operations differ in one crucial respect: the order of execution of the **DO** loop is index order, while the order of execution of the **FORALL** assignments is undefined. The **FORALL** statement, but not the **DO** loop, can therefore be executed in parallel by the individual CM processors.

The significance of as-if-simultaneous execution becomes clear when we use **FORALL** to assign an element whose own value is assigned to some other element. For example, consider a transposition of an **N** x **N** matrix **B**:

```
FORALL ( I=1:N, J=1:N )  B(I,J) = B(J,I)
```

Notice that the statement does not explicitly save the initial array values in a temporary location, as a **DO** construct would need to do. If all the element values are moved simultaneously, an element being moved cannot have already received the value of its opposite number. (In fact, when executed serially, **FORALL** creates any necessary temporaries transparently to the user.) Thus, the programmer need not take steps to ensure that only the original values are used in a **FORALL** operation. All the elemental assignments are executed *as if* at the same moment, when only the original values are available.

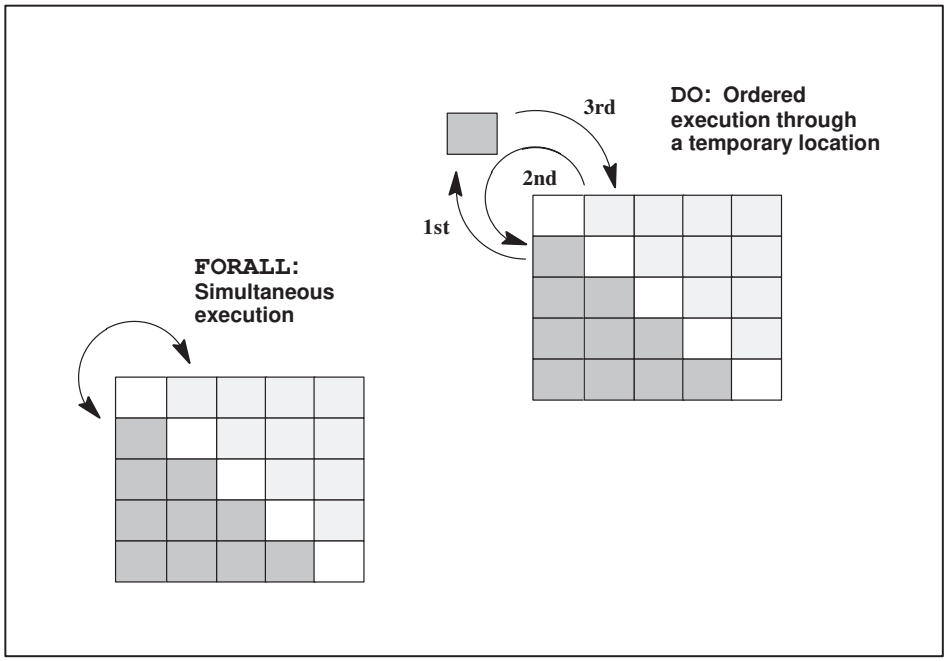


Figure 18. DO versus FORALL in transposing array elements (2,1) and (1,2).

Copyright © 1989–1994 Thinking Machines Corporation

10.2.2 Serial versus Parallel

The **FORALL** statement is not considered an array operation for the purpose of determining an array's home. **FORALL** can operate on either CM arrays or front-end arrays (or both at once), and it can execute either serially on the control processor or in parallel on the CM processors.

This section outlines the conditions under which **FORALL** executes serially or in parallel. The compiler generates a warning whenever it serializes a **FORALL** statement.

Determined by Array Home

Generally, the homes of the arrays referenced in the **FORALL** assignment determine whether **FORALL** executes serially or in parallel (the next section notes the exceptions). Specifically:

- **FORALL** assignments that reference only front-end arrays always execute serially:

```
FORALL (I=1:N)  FE_ARRAY(I) = I                ! Serial, FE
FORALL (I=1:N)  FE_ARRAY(I) = FE_ARRAY(I+1)    ! Serial, FE
```

- **FORALL** assignments that reference only CM arrays generally execute in parallel (with exceptions as noted below):

```
FORALL (I=1:N)  CM_ARRAY(I) = I         ! Parallel
FORALL (I=1:N)  CM_ARRAY(I) = CM_ARRAY(I+1) ! Parallel
```

- **FORALL** assignments that reference both front-end arrays and CM arrays always execute serially:

```
FORALL (I=1:N)  FE_ARRAY(I) = CM_ARRAY(I) ! Serial, FE
FORALL (I=1:N)  CM_ARRAY(I) = FE_ARRAY(I) ! Serial, FE
```

When a **FORALL** assignment references arrays with different homes, the system moves CM values one at a time to or from the front end to perform the serial operations. As with any mixed-home array operation, this data transfer degrades program performance considerably. See Chapter 2 for discussion of mixed-home operations and some suggestions for avoiding them.

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

Determined by Kind of Expression

Certain kinds of expressions in the **FORALL** assignment always cause the statement to be executed serially, even when only CM arrays are referenced. These are:

- any reference to a user-defined function, including a statement function:

```
FORALL (I=1:N)  A(I) = USER_FUNCTION(I) ! Serial, FE
```

- any reference to a character variable, such as `(STRING)I`
- any reference to an intrinsic function with a nonconstant **DIM** argument (that is, a dimension not known at compile time)

Determined by Temporary Restrictions

Please see the *CM Fortran Release Notes* for the current version for any restrictions on the efficient execution of **FORALL**.

Copyright © 1989–1994 Thinking Machines Corporation

10.3 Data Motion

The **FORALL** statement offers an alternative to other CM Fortran features in expressing various forms of data motion. The alternatives often vary in syntactical convenience and sometimes in performance. **FORALL** statements are often as fast as the fastest alternative means of expression. On serial dimensions, **FORALL** often gives faster performance than vector-valued subscripts and intrinsic functions.

In addition, **FORALL** can express several patterns of data motion that are otherwise difficult to express as parallel operations in CM Fortran. Among these are arbitrary permutations of multidimensional arrays, operations on nonrectangular or irregular data, parallel prefix operations along array dimensions, and send-with-combiner operations.

10.3.1 Compared with Other Features

FORALL can express all the array transformations and data motion operations described in Chapters 8 and 9, mimicking the behavior of array assignments that reference array sections, vector-valued subscripts, and transformational intrinsic functions. This section reviews the major categories of data motion already discussed and illustrates **FORALL** in expressing each. The sections that follow show patterns of movement for which **FORALL** is the only convenient choice.

Grid Communications

Grid communications, including dimensional shifts and other nearest-neighbor accesses, have in common the fact that element values all move the same distance in the same direction to reach their respective destinations. The following statements all shift matrix values by four positions on the first dimension.

```
DIMENSION A(N,M), B(N,M)
. . .
FORALL (I=1:N-4) B(I,:) = A(I+4,:)

B(1:N-4,:) = A(4:N,:)

B = CSHIFT( A, DIM=1, SHIFT=4 )
```


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

The semantics of the last statement differs slightly from the other two: the function **CSHIFT** wraps the elements that move off the edge of the array, whereas the other two statements explicitly prevent any elements from moving off the edge.

General Communications

General communications, including transpositions and arbitrary permutations (gathers/scatters), have in common the fact that element values move different distances or different directions to reach their respective destinations.

General communication operations, whether expressed with **FORALL**, with vector-valued subscripts, or with transformational intrinsic functions, typically generate **send** communication routines. An important exception is the use of **FORALL** for indirect addressing on a serial dimension; this operation generates only local memory accesses and uses the fast indirect addressing hardware for best performance.

Examples of optimized indirect addressing:

```
      INTEGER SOURCE(NS,NP), DEST(NP), INDEX(NP)
CMF$  LAYOUT SOURCE(:SERIAL,), DEST(), INDEX()
      ...
      FORALL (I=1:NP) DEST(I)=SOURCE((I),I)
```

and,

```
      INTEGER, DIMENSION(NS,NP) :: SOURCE, DEST, INDEX
CMF$  LAYOUT SOURCE(:SERIAL,),DEST(:SERIAL,)
CMF$  LAYOUT INDEX(:SERIAL,)
      ...
      FORALL (I=1:NS,J=1:NP)
     &   DEST(I,J)=SOURCE(INDEX(I,J),J)
```

This form of **FORALL** on serial dimensions gives performance comparable to the utility library procedures **CMF_AREF**/**ASET_1D**.

FORALL and vector-valued subscripts are the only features CM Fortran has for expressing an unordered set of indices. For example, the following statement uses two index vectors to access locations in a matrix. The lengths of the index vectors **V** and **R** are unrelated to the source array **C**, but they must equal the length of the corresponding dimensions of the destination array **D**:

Copyright © 1989–1994 Thinking Machines Corporation

```
DIMENSION C(M,M), D(N,N)
DIMENSION V(N), R(N)
...
D = C(V,R)
```

FORALL can mimic this behavior, albeit in a more cumbersome manner:

```
FORALL (I=1:N, J=1:N) D(I,J) = C( V(I), R(J) )
```

The real power of **FORALL**, however, is that it is not limited to vectors for indexing into multidimensional arrays. **FORALL** can perform a perfectly arbitrary permutation such as the following, which uses index arrays **X** and **Y**:

```
FORALL (I=1:N,J=1:M) D(I,J) = C( X(I,J), Y(I,J) )
```

Global Communication

A third form of communication involves cumulative computations along array dimensions. Examples of such *global communication* are the reduction intrinsics and the transformational function **SPREAD** (see Chapter NO TAG). Global communication is faster than general communication on the CM system.

FORALL can express a spread operation — that is, replicate an array along a specified dimension of another array of rank one higher — with about the same performance as the function **SPREAD**. For example, both the following statements spread a vector along the first dimension of a matrix:

```
DIMENSION A(N,M), V(M)
...
FORALL ( I=1:N ) A(I,:) = V
```

is functionally equivalent to

```
A = SPREAD( V, DIM=1, NCOPIES=N )
```

Similarly, **FORALL** can express a "reduce and spread" operation, where the result of a reduction operation along an array dimension is replicated across a dimension:

```
DIMENSION A(N,M)
...
FORALL ( I=1:M ) A(:,I) = SUM( A, DIM=2 )
```


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

is functionally equivalent to

```
A = SPREAD( SUM( A, DIM=2 ), DIM=2, NCOPIES=M )
```

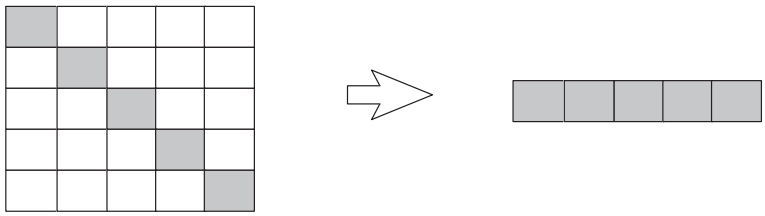
This example computes the sum of each row of **A** and returns the result as a vector. The vector is then replicated across the rows of **A**, with the result that each element gets the sum of all the elements on the second dimension:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix} \Rightarrow A = \begin{bmatrix} 10 & 10 & 10 & 10 \\ 14 & 14 & 14 & 14 \\ 18 & 18 & 18 & 18 \\ 22 & 22 & 22 & 22 \end{bmatrix}$$

10.3.2 Operations on Irregular Data

One of the major uses of **FORALL** is to express operations on irregularly shaped parts of an array. Such operations can be expressed serially with a **DO** construct, but **FORALL** is the only convenient way to express them for the CM.

A simple example of an operation on an irregularly shaped part of an array is extracting the diagonal elements of a matrix:

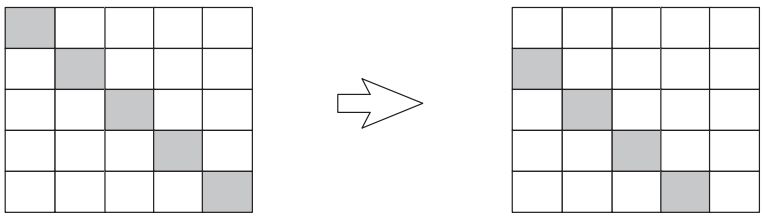


The **FORALL** expression of this operation is:

```
DIMENSION C(N,N)
DIMENSION D(N)
...
FORALL ( I=1:N ) D(I) = C(I,I)
```


Copyright © 1989–1994 Thinking Machines Corporation

Similarly, consider shifting the values in a matrix diagonal, say, one position to the left:



The elemental assignments that occur in this operation are

```
A(2,1) = A(2,2)
A(3,2) = A(3,3)
A(4,3) = A(4,4)
A(5,4) = A(5,5)
```

or, as expressed with **FORALL**,

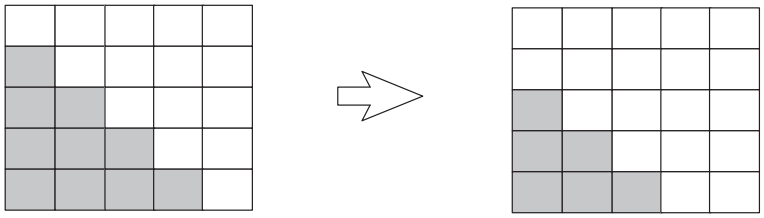
```
FORALL (I=2:5)  A(I,I-1) = A(I,I)
```

As with **DO**, the **FORALL** expressions can describe a number of possible shapes. For example, the following statement shifts the lower triangle of a matrix by one position:

```
FORALL (I=1:N, J=1:N, J.LT.I)  A(I,J) = A(I, J+1)
```

or

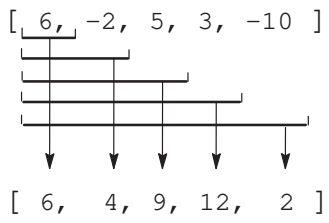
```
FORALL (I=1:N)  A(I,1:I-1) = A(I, 2:I)
```



Copyright © 1989–1994 Thinking Machines Corporation

10.3.3 Parallel Prefix Operations

Parallel prefix operations, sometimes called *scans*, apply some combining operation to the elements along an array dimension; that is, they compute for each element the combination of itself and all previous elements on that dimension. For example, a sum-prefix (or *add-scan)* is the familiar process of balancing a checkbook: each credit or debit is added to the sum of the previous ones to yield a running subtotal:



Parallel prefix operations are used by many scientific algorithms, such as in line-of-sight and convex-hull algorithms in computational geometry. Often, they are computed with **DO** loops using a serial approach. For instance, the example above could be expressed:

```
B(1) = A(1)
DO I=2,5
    B(I) = B(I-1) + A(I)
END DO
```

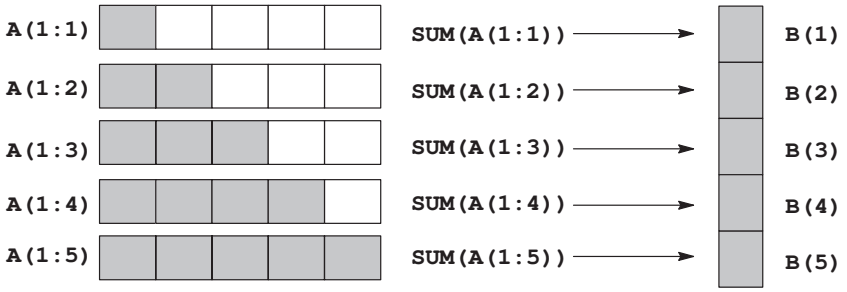
Despite the serial algorithm used in scanning with **DO**, scans are not inherently serial operations: they are computed efficiently by the CM using the global communication network. To express a scan as a parallel operation, use **FORALL** to specify the set of subarrays over which the combiner is applied and specify the combiner with a reduction function. The destination array can be any array that is conformable with the source array, including the source array itself.

For example, to express an add-scan with **FORALL**:

```
FORALL (I=1:5) B(I) = SUM( A(1:I) )
```

The reduction function **SUM** returns, for each value of **I**, the scalar sum of the values up to and including **A(I)**, which is then assigned to **B(I)**.

Copyright © 1989–1994 Thinking Machines Corporation



To reverse the direction of the scan, so that each element gets the sum of itself and those following, and the first element gets the cumulative sum:

```
FORALL (I=1:N)  B(I) = SUM( A( N-I+1 : N ) )
```

The scan operation is easily extended to a multidimensional array, although only one dimension can be scanned at a time:

```
FORALL (I=1:N)  B(I,:) = SUM( A(1:I,:) )
```

The combiner in a scan operation can be any operation for which CM Fortran provides a reduction function: **SUM**, **PRODUCT**, **MAXVAL**, **MINVAL**, **ANY**, **ALL**, or **COUNT**. For example, the following code uses a multiply-scan to compute factorials:

```
DIMENSION FACT(N)
FACT = [1:N]
FORALL (I=1:N) FACT(I) = PRODUCT( FACT(1:I) )
```

Following are a number of scan operations using **FORALL**.

Simple Scans

- upward inclusive 1D scan, axis 1:

```
FORALL(I=1:N) A(I) = SUM(B(1:I))
```

- upward exclusive 1D scan, axis 1:

```
FORALL(I=1:N) A(I) = SUM(B(1:I-1))
```

- upward inclusive 2D scan, axis 1:

```
FORALL(I=1:N,J=1:M) A(I,J) = SUM(B(1:I,J))
```


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

- upward inclusive 2D scan, axis 2:

```
FORALL(I=1:N,J=1:M) A(I,J) = SUM(B(I,1:J))
```

Segmented Scan

- upward inclusive segmented unconditional:

```
 FORALL (I=1:N) A(I) =
&   SUM(B(MAXVAL([1:I],SEGMENT(1:I)):I))
```

Scan of Scan

```
FORALL(I=1:N,J=1:M) A(I,J) = SUM(B(1:I,1:J))
```

Comparison with Utility Procedure CMF_SCAN_

The scan capabilities of **FORALL** are largely, but not entirely, the same as those of the utility library procedures **CMF_SCAN_***combiner*. The comparative combiners are:

CMF_SCAN_*combiner*	**FORALL** expression
ADD	**SUM**
MAX	**MAXVAL**
MIN	**MINVAL**
mult scan (not available)	**PRODUCT**
LOGICAL IOR	**ANY**
LOGICAL IAND	**ALL**
LOGICAL IEOR	**MOD(COUNT(),2)==1**
INTEGER IOR	(not available)
INTEGER IEOR	(not available)
INTEGER IAND	(not available)

NOTE: Performance of **FORALL** and **CMF_SCAN_** is comparable.

Copyright © 1989–1994 Thinking Machines Corporation

For example, a segmented add scan with **FORALL** can be written:

```
  FORALL(I=1:N)
&      A(I)  =  SUM(B(MAXVAL([1:I],SEGMENT(1:I)):I))
```

To do a copy scan, you can use a vector-valued subscript instead of the sum of a triplet expression. The triplet is **MAXVAL([1:I],SEGMENT(1:I))** as the lower bound and **I** is the upper bound.

The following examples show variations in direction, inclusion, and segment mode, as defined for **CMF_SCAN_***combiner*. Note that this form of **FORALL** can make use of two different masks:

- The reduction mask, the array-valued **MASK** argument to the reduction function, indicates which elements participate in the reduction.
- The **FORALL** mask, a scalar, indicates which elements are assigned.

Each example is shown first with a reduction mask implicitly **.TRUE.** and then with a reduction **MASK** argument.

- 1D upward, inclusive, none:

```
FORALL(I=1:N) A(I) = SUM(B(1:I))
FORALL(I=1:N) A(I) = SUM(B(1:I),MASK=MASK(1:I))
```

- 1D downward, inclusive, none:

```
FORALL(I=1:N) A(I) = SUM(B(I:N))
FORALL(I=1:N) A(I) = SUM(B(I:N),MASK=MASK(I:N))
```

- 1D upward, exclusive, none:

```
FORALL(I=1:N) A(I) = SUM(B(1:I-1))
FORALL(I=1:N) A(I) = SUM(B(1:I-1),MASK=MASK(1:I-1))
```

- 1D downward, exclusive, none:

```
FORALL(I=1:N) A(I) = SUM(B(I+1:N))
FORALL(I=1:N) A(I) = SUM(B(I+1:N),MASK=MASK(I+1:N))
```


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

- 1D upward, inclusive, segment:

  Note that the segment in this form must start with a **.TRUE.** element.

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAXVAL([1:I],MASK=SEGMENT(1:I)):I))
```

  In this form, the segment starts with an implicit **.TRUE.**.

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAX(1,MAXVAL([1:I],
&   MASK=SEGMENT(1:I))):I)))
```

  The intrinsic **MAXVAL** returns most-negative-integer when its mask is null. Therefore, where segment is **.FALSE.** and the **MAXVAL** expression returns **MOST_NEG_INT**, the **SUM** expression would be an error: **B(MOST_NEG_INT:I)**. The expression **MAX(1, MAXVAL(...))** causes the **SUM** expression to be **1:I** even if segment is **.FALSE.**.

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAX(1,
&   MAXVAL([1:I],MASK=SEGMENT(1:I))):I))
```

- 1D downward, inclusive, segment:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(I:MINVAL([I+1:N],MASK=SEGMENT(I+1:N))-1))
```

- 1D upward, exclusive, segment:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAXVAL([1:I],MASK=SEGMENT(1:I)):I-1))
```

- 1D downward, exclusive, segment:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(I+1:MINVAL([I+1:N],
&   MASK=SEGMENT(I+1:N))-1))
```

- 1D upward, inclusive, start:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAXVAL([1:I],MASK=SEGMENT(1:I)):I))
```


Copyright © 1989–1994 Thinking Machines Corporation

- 1D upward, exclusive, start:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(MAXVAL([1:I-1],MASK=SEGMENT(1:I-1)):I-1))
```

- 1D downward, inclusive, start:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(I:MINVAL([I:N],MASK=SEGMENT(I:N))))
```

- 1D downward, exclusive, start:

```
 FORALL(I=1:N) A(I) =
&   SUM(B(I:MINVAL([I:N],MASK=SEGMENT(I:N))-1)
```

10.3.4 Send-with-Combiner Operations

The **FORALL** statement generates parallel send-with-combiner instructions for *n*-to-*m*-dimensional scatters when the possibility of data collisions exists. For each dimension of the destination array, supply an index array that is conformable with the source array, and then use a reduction function to combine multiple values being sent to the same destination element.

For example, a 1D-to-1D send-with-add operation is written as

```
FORALL(I=1:8) A(I)=SUM(B(1:1000),MASK=V(1:1000).EQ.I)
```

where
A is **A(8)** of numeric type.
B is **B(1000)** of numeric type.
V is **V(1000)** of type integer.

A 1D-to-1D send-with-add operation that adds in the original destination value is written as

```
FORALL(I=1:N) A(I) = A(I) + SUM(B(:), MASK=V(:).EQ.I)
```

For a 2D-to-2D send-with-add, use an index array (conformable with the source array) for each dimension of the destination array:

```
 FORALL(I=1:N,J=1:M)
&       OUT(I,J) =
&       SUM(IN(:,:),
&           MASK=(X(:,:).EQ.I).AND.(Y(:,:).EQ.J))
```


Copyright © 1989–1994 Thinking Machines Corporation

A 1D-to-2D send-with-add operation is written as

```
 FORALL(I=1:N,J=1:M)
&       OUT(I,J)=
&       SUM(IN(:), MASK=X(:).EQ.I .AND. Y(:).EQ.J)
```

A restriction on this use of **FORALL** is that it generates parallel instructions *only if* the rank of **OUT** plus the rank of **IN** is 31 or less. For arrays of higher rank, use the utility library procedure **CMF_SEND_***combiner* for best performance. For arrays of lower rank, the performance of the two constructions is comparable.

Copyright © 1989–1994 Thinking Machines Corporation