Chapter 10

Assignment

When an object is given a valid value during program execution, it is said to become *defined*. This is accomplished by execution of an assignment statement or an input statement. If a variable ceases to have a predictable value it is said to be *undefined*. The actions that cause a variable to become defined or undefined are given in Chapter 16.

Arrays of one or more elements, character strings of length greater than one, and variables of type complex, may be *partially* defined. For example, an array section may refer to a subset of the elements of a parent array, and that subset may be referenced or defined independently of the other elements. Similarly, a substring of a string may be referenced or defined independently of the remaining characters of the string. Through storage association, either the real part or the imaginary part of a complex number may be assigned or referenced independently of the other.

Zero-sized arrays are always defined. The following variables are defined initially:

- variables specified to have initial values by `DATA` statements,
- variables specified to have initial values by type declaration statements with the `DATA` attribute, and
- zero-sized array variables.

All other variables are initially undefined.

Execution of an assignment causes the value of a variable to become defined or changed. An assignment is either an assignment statement (Section 10.1) or a masked array assignment (Section 10.2).

Copyright © 1989–1994 Thinking Machines Corporation

10.1 Scalar and Array Assignment

An *assignment statement* (R10-1) causes a variable to become defined with the value of an expression.

An *assignment-stmt* is: R10-1
variable = *expr*

An assignment statement is a *numeric assignment statement* if the variable and expression being assigned are of numeric type; *character assignment statements* and *logical assignment statements* are defined similarly.

An *array assignment statement* is an assignment statement in which the variable being assigned is an array. CM Fortran does not support array assignment to arrays of type character.

Examples of scalar assignment statements:

```
REAL A, B, X, Y
...
A = 3.5 + X*Y        ! scalar assignment
B = INT( A )         !   ditto
```

Examples of array assignment statements:

```
REAL, DIMENSION(100) :: A, B, X, Y
INTEGER MASK(5)
...
A = 3.5 + X*Y        ! array assignment
B = INT( A )         !   ditto
MASK = [1,3,5,7,9]   ! array assignment of constructor
```

10.1.1 Assignment Conformance

For any assignment statement, *variable* and *expr* must conform in shape, and if *expr* is an array, *variable* must also be an array. For a numeric assignment statement, *variable* and *expr* may have different numeric types, in which case the value of *expr* is converted to the type of *variable* according to the rules of Table 9, p. 119. For a logical assignment, both the variable and the expression must be of type logical.

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

Table 9. Numeric conversion for the assignment statement *variable* = *expr*.

The functions INT, REAL, and CMPLX are the generic intrinsic functions defined in Chapter 17.

Type of *variable*	Value Assigned
integer	INT(*expr*)
real	REAL(*expr*, MOLD=*variable*)
complex	CMPLX(*expr*, MOLD=*variable*)

NOTE: When a program is compiled with the `-f90syntax` switch, the keyword KIND replaces the equivalent MOLD, making the function CMPLX Fortran 90-compatible.

For a character assignment statement, *variable* and *expr* may have different lengths, in which case the conversion of *expr* to the length of *variable* is performed as follows:

- If the length of *variable* is less than that of *expr*, the value of *expr* is truncated from the right until it is the same length as *variable*;
- If the length of *variable* is greater than that of *expr*, the value of *expr* is extended to the right with blanks until it is the same length as *variable*.

10.1.2 Interpretation of Scalar and Array Assignment

Execution of an assignment statement (R10-1) effectively proceeds in three steps:

1. the expression *expr* and all expressions within the *variable* are evaluated (as noted in Section 7.4, it may be unnecessary to evaluate all subexpressions of *expr* or *variable* in order to determine the value and location at which to store)
2. if necessary, the *expr* is converted to the type of *variable* (also, if *expr* is of type character, it may be truncated or extended to match the length of *variable*), and
3. the *variable* is defined with the resulting value.

The execution of the assignment has the same effect as if the evaluation of all operations in *expr*, and any operations in the subscripts or section subscripts of *variable* occurs before any portion of *variable* is defined by the assignment. The evaluation of expressions within *variable* does not affect, nor is it affected by the evaluation of *expr*.

Copyright © 1989–1994 Thinking Machines Corporation

Both *variable* and *expr* may contain references to any portion of *variable*. For example, in the array assignment statement below

```
INTEGER DIGITS(0:9) = [0:9]
DIGITS(3:7) = -DIGITS(5:9)
```

the value of DIGITS after the assignment is [0,1,2,-5,-6,-7,-8,-9,8,9]. As another example, in the character assignment statement

```
STRING(2:5) = STRING(1:4)
```

the assignment of the first character of STRING to the second character does not affect the evaluation of STRING(1:4). That is, if the value of STRING prior to the assignment is 'ABCDEF', the value following the assignment is 'AABCDF'.

10.1.2.1 Elemental Array Assignment

When the variable being assigned a value in an assignment statement is an array, the values are assigned element-by-element in arbitrary order. Such assignment is called *elemental array assignment*. (Elemental assignment of arrays is performed in parallel on the CM using a single machine instruction.)

If *expr* in an assignment is a scalar and *variable* is an array, the *expr* is treated as if it were an array of the same shape as *variable* with every element equal to the scalar value of *expr*.

When the *variable* in an assignment statement is an array, the assignment is performed element-by-element on corresponding array elements of *variable* and *expr*. For example, if A and B are arrays of the same shape, the array assignment statement

```
A = B
```

assigns the corresponding elements of B to those of A; that is, the first element of B is assigned to the first element of A, the second element of B is assigned to the second element of A, etc.

The following program fragment causes the values of the elements of array X to be reversed:

```
REAL X(10)
...
X(1:10) = X(10:1:-1)
```

When the *variable* of an assignment is a component of an object, the assignment does not affect the value of other parts of the object. For example, if *variable* is an array section, the assignment does not affect the value of the elements of the parent array not specified by the array section.

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

10.2 Masked Array Assignment: WHERE

A *masked array assignment* is an elemental array assignment performed by a WHERE statement or a WHERE construct. Masked array assignment evaluates selected elements of an array expression under control of a logical array mask expression and assigns the resulting values to an array variable.

10.2.1 WHERE Statement

The WHERE statement (R10-2) qualifies the evaluation of expressions and assignment of resulting values in a single array assignment statement.

A *where-stmt* is: R10-2
WHERE (*mask-expr*) *where-assignment*

A *mask-expr* is: R10-3
logical-expr

A *where-assignment* is: R10-4
array-*variable* = *expr*

- The *variable* being defined (assigned to) in a *where-assignment* must be an array of the same shape as *mask-expr*. The *where-assignment* is a form of *assignment-stmt* (R10-1).

The *mask-expr* (R10-3) is evaluated, then the evaluation of expressions and execution of the assignment of individual array elements takes place for all elements for which the *mask-expr* is true, as described in Section 10.2.3.

Examples of WHERE statements:

```
! Reduce temperatures above 98.6 by a specified amount
WHERE ( TEMP > 98.6 ) TEMP = TEMP - TEMP_REDUCTION

WHERE ( E(N:N+59) .LE. 100 ) D(1:60) = SUM( C(1:100) )
```

The second WHERE statement assigns the sum of the first hundred elements of C to selected elements of a 60-element section of D. The only elements of D that are updated are those corresponding in position to elements of a section of a mask array E that are not greater than 100.

Copyright © 1989–1994 Thinking Machines Corporation

```
DO I = MIN,MAX,STEP
  WHERE ( I <= DATA .AND. DATA < I+STEP ) TEMP = DATA
  HISTO = SUM( TEMP, 1 )
  PRINT *, I, I+STEP, HISTO
END DO
```

The WHERE statement in the loop above selects a subset of values from the array DATA that lie within a specified range and assigns the result to the array TEMP to be totaled and printed.

10.2.2 WHERE Construct

The WHERE construct (R10-5) qualifies the evaluation of expressions and assignments of values in several array assignment statements.

A *where-construct* is: R10-5

WHERE (*mask-expr*)
 [where-body]...
[ELSEWHERE
 [where-body]...]
[ELSEWHERE (*mask-expr*)
 [where-body]...]
END WHERE

A *where-body* is one of: R10-6

- *where-stmt* (R10-2)
- *where-assignment* (R10-4)
- *where-construct* (R10-5)

- The *variable* being defined (assigned to) in a *where-assignment* must be an array of the same shape as *mask-expr* (R10-3). The *where-assignment* is a form of *assignment-stmt* (R10-1).

- All *mask-expr*s within a *where-construct* must be logical arrays of the same shape.

The *mask-expr* is evaluated, and the assignment statements following the WHERE keyword are executed for each array element corresponding to a true value in the mask expression.

If the keyword ELSEWHERE is present and is not followed by a mask expression, then the assignment statements following it are executed for each array element for which the *mask-expr* after the WHERE keyword is false.

If the keyword ELSEWHERE is present and *is* followed by a mask expression, then the assignment statements following it are executed for each array element corresponding to

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

a false value in the mask expression specified in the WHERE statement and a true value in the mask expression specified in the ELSEWHERE statement.

If a *where-body* consists of another WHERE construct, the assignment statements within the nested WHERE construct are executed for each array element corresponding to true values in the *mask-expr*s for *both* the outer and inner WHERE construct statements. In cases in which a WHERE assignment has modified a *mask-expr*, the original values or *mask-expr*s are used.

The execution of the *where-construct* is described in detail in the next section.

Examples of the WHERE construct:

```
WHERE ( PRESSURE < 1. )
  PRESSURE = PRESSURE + INC_PRESSURE
  TEMP = TEMP - 5.
END WHERE


WHERE ( PRESSURE < 1. )
  PRESSURE = PRESSURE + INC_PRESSURE
  TEMP = TEMP - 5.
ELSEWHERE
  RAINING = .TRUE.
END WHERE


WHERE ( PRESSURE < 1. )
  PRESSURE = PRESSURE + INC_PRESSURE
  TEMP = TEMP - 5.
ELSEWHERE ( TEMP > 0. )
  RAINING = .TRUE.
END WHERE
```

In the second and third examples, the arrays PRESSURE, TEMP, and RAINING must all be arrays of the same shape. All may be modified by the construct.

10.2.3 Interpretation of Masked Array Assignments

The execution of a masked array assignment causes the expression *mask-expr* to be evaluated. When the *assignment-stmt* in a *where-stmt* is executed, the expression on the right-hand side of the *assignment-stmt* is evaluated and the results are assigned to the cor-

Copyright © 1989–1994 Thinking Machines Corporation

responding elements of the variable on the left-hand side of the assignment statement. Depending on the CM execution model, the evaluation of the right-hand side expression may be of all elements or of only those whose corresponding *mask-expr* element is true. In either case, the assignment occurs only for the elements for which *mask-expr* is true.

When a *where-construct* is executed, the *mask-expr* is evaluated and the result saved. This value of *mask-expr* governs the masking in the execution of the `WHERE` construct; subsequent changes to variables referenced in *mask-expr* have no effect on the masking behavior. Each *assignment-stmt* following the `WHERE` keyword is evaluated, in the order in which it appears, as if it were

`WHERE ( ` *mask-expr* ` )` *assignment-stmt*

and then each assignment following the `ELSEWHERE` keyword, if present, is evaluated, in the order in which it appears, as if it were

`WHERE ( .NOT.` *mask-expr* ` )` *assignment-stmt*

Each of the array assignment statements following the `WHERE` and `ELSEWHERE` keywords are executed in normal execution sequence. A reference to any array may appear subsequent to its definition in the same `WHERE` construct.

If a *non*elemental function reference occurs in the expression on the right-hand side of an *assignment-stmt*, the function is evaluated without any masked control; that is, *all* its argument expressions are fully evaluated and the function is fully evaluated. If the result is an array, elements corresponding to true values in *mask-expr* (elements corresponding to false values in assignment statements following `ELSEWHERE`) are selected for use in evaluating the expression of the *assignment-stmt*.

If an elemental intrinsic function reference occurs in the expression on the right-hand side of an *assignment-stmt* and that function reference is not itself an actual argument of a nonelemental function reference, the function is evaluated only for the elements corresponding to true values in *mask-expr* (and only for elements corresponding to false values in assignment statements following `ELSEWHERE`).

Of the statements comprising a `WHERE` construct, only the `WHERE` statement and the `END WHERE` statement may be branch target statements. Execution of an `END WHERE` has no effect.

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

10.2.4 Nested WHERE Constructs

The WHERE construct selects certain elements of an array for an array operation. A mask expression qualifies the evaluation of expressions and the assignment of values in the array operation. A WHERE body can contain:

- a WHERE assignment
- a WHERE statement
- another WHERE construct

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:

```
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
```


Copyright © 1989–1994 Thinking Machines Corporation

This form is evaluated as if it were written:

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

In cases where an assignment statement changes a mask expression, the original mask expression values are used. In the following evaluation, temporaries are used to illustrate this. 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:

```
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
```


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

In the following example, there is no assignment statement immediately after the initial WHERE:

```
WHERE ( mask-expr1 )
 WHERE ( mask-expr2 )
  assignment-stmt-1
 ELSEWHERE ( mask-expr3 )
  assignment-stmt-2
 END WHERE
ELSEWHERE
 assignment-stmt-3
END WHERE
```

This construct is evaluated as if it were written:

```
WHERE ( mask-expr1 .AND. mask-expr2 )
 assignment-stmt-1
ELSE
 WHERE ( mask-expr1 .AND. mask-expr3 )
  assignment-stmt-2
 END WHERE
ELSEWHERE
 assignment-stmt-3
END WHERE
```

10.3 FORALL Elemental Array Assignment

The FORALL statement (R10-7) is an elemental array assignment statement used to specify an array assignment in terms of array elements or array sections. The element array may be masked with a scalar logical expression. The FORALL statement effectively describes a collection of assignments to be executed elementally.

A *forall-stmt* is: R10-7
FORALL (*forall-triplet-spec(s) [* , scalar-mask-*expr]*) *forall-assignment*

A *forall-triplet-spec* is: R10-8
index-variable-*name* = *subscript* : *subscript [* : *stride]*

An *index-variable-name* is: R10-9
scalar-integer-variable-*name*

- The index-variable-*name* must be a scalar-*name* of type integer.

Copyright © 1989–1994 Thinking Machines Corporation

- A *subscript* or a *stride* in a *forall-triplet-spec* must not contain a reference to any index-variable-*name* in the *forall-triplet-spec(s)* list.

A *forall-assignment* is one of: R10-10

- *array-element-designator* = *expr*
- *array-section* = *expr*

- The *array-section* or *array-element-designator* in a *forall-assignment* must reference all the index-variable-*name*s specified in the *forall-triplet-spec(s)* list. The *forall-assignment* is a form of *assignment-stmt* (R10-1).

For each index variable name in the *forall-assignment*, the set of permitted values over which that subscript may range is determined at the beginning of statement execution and is

m_1 + (*k* − 1)*m_3

where *k* = 1,2,...,INT((m_2 − m_1 + m_3)/m_3), and where m_1, m_2, and m_3 are the values of the first subscript, the second subscript, and the stride, respectively, in the *forall-triplet-spec*. If the *stride* is missing, the effect is as if it were present with a value of 1. The expression *stride* must not have the value 0. If for some index variable name the value of the expression above is less than or equal to zero, the *forall-assignment* is not executed.

Execution of an elemental array assignment entails the evaluation in any order of the *subscript* and *stride* expressions in the *forall-triplet-spec(s)* list, the evaluation of the scalar-mask-*expr*, and the evaluation of the *expr* in the *forall-assignment* for all valid combinations of values of index variables for which the scalar mask expression is true, followed by the assignment of the resulting *expr* values to the corresponding elements of the array being assigned to. If the scalar mask is omitted, the effect is as if it were present with value true.

NOTE: The *forall-assignment* must not cause any element of the array being assigned to be assigned a value more than once.

The scope of the index variable name is the FORALL statement itself. A function reference appearing in any expression in the *forall-assignment* must not alter the value of any index variable name in the *forall-assignment*.

NOTE

The FORALL statement does not in itself cause a referenced front-end array to become a CM array (i.e., the statement is not classified as an array operation for purposes of determining the

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

home of an array). A FORALL statement that assigns to a front-end array is executed serially. A FORALL statement that assigns to a CM array is executed either in parallel or serially depending on the form of the *forall-assignment*.

To better understand the workings of the FORALL assignment statement, consider a statement of the form

```
  FORALL ( forall-triplet-spec1, forall-triplet-spec2, ..., forall-triplet-speck,
$          [ , scalar-mask-expr ] ) forall-assignment
```

where each *forall-triplet-spec*$_i$ has the form

index-variable-*name*$_i$ = *subscript*$_i$: *subscript*$_i$ [: *stride*$_i$]

The *forall-triplet-spec(s)* in the FORALL statement effectively specify a (possibly empty) set of k-tuples, where k is the number of elements in the *forall-triplet-spec(s)* list. The cardinality of this set is $N_1 * N_2 * \ldots * N_k$, where each N_i is the cardinality of the set of values determined by the ith *forall-triplet-spec*$_i$. The *subscript*$_i$ and *stride*$_i$ expressions of the ith triplet specification determine the set of values designated by the ith position of each k-tuple, in the same way as the corresponding subexpressions of array sections (see Section 7.1.6). An empty range for any triplet specification causes the set of k-tuples to be empty.

The scalar-mask-*expr* is usually a function of the set of k-tuples, but may be a constant function (an expression), or may be omitted. Depending on the CM execution model, the expression on the right-hand side of the FORALL assignment may or may not be evaluated only with those k-tuples for which the mask expression is true. This evaluation yields a set of pairs <S,V>, where S is a k-tuple of integer values and V is either a scalar value or an array value. Each k-tuple S represents a set of values for each of the index variable names index-variable-*name*$_1$, ..., index-variable-*name*$_k$ for which the mask expression holds true (the set may be empty). Within the *forall-assignment* (on either side of the equals), the *i*th index variable name may be used to reference the ith component of any computed k-tuple. Every index variable name must be used on the left-hand side of the assignment (although several might appear in a single subscript expression), but none of the index variable names need to be used on the right-hand side; typically, some are.

The scalar-mask-*expr* is usually a function of the set of k-tuples, but may be a constant function (an expression), or may be omitted. The expression on the right-hand side of the forall assignment is evaluated only with those k-tuples for which the mask expression is true. This evaluation yields a set of pairs <S,V>, where S is a k-tuple of integer values and V is either a scalar value or an array value. Each k-tuple S represents a set of values for each of the index variable names index-variable-*name*$_1$, ..., index-variable-*name*$_k$ for

Copyright © 1989–1994 Thinking Machines Corporation

which the mask expression holds true (the set may be empty). Within the *forall-assignment* (on either side of the equals), the *i*th index variable name may be used to reference the *i*th component of any computed *k*-tuple. Every index variable name must be used on the left-hand side of the assignment (although several might appear in a single subscript expression), but none of the index variable names need to be used on the right-hand side; typically, some are.

Each value V of the pair <S,V> is assigned to one or more elements of the array element or section named on the left-hand side of the assignment statement. If the left-hand side is a scalar, only one element of the array is assigned the value V. If the left-hand side is a section, the section is assigned the value V (which may be a scalar or a conformable array). No element of the array should be assigned more than once. For each pair <S,V>, the value V is assigned to the array element or section of the target array determined by the values of the subscript and stride expressions in the *array-element-designator* or the *array-section*, when the index variable names are replaced by the values of the *k*-tuple S.

The entire assignment is performed under control of the mask. Depending on the CM execution model, the right-hand side expression may or may not be evaluated for those combinations of subscripts that would yield a false mask expression.

Note that a scalar mask expression used in a FORALL assignment statement has no effect on elemental intrinsic functions referenced in the assignment statement. Thus, the statement

```
 FORALL ( I = 1:10, MOD(I,2) == 0 )
$  A(I) = SUM( A(1:I) )
```

is not equivalent to the statement

```
  FORALL ( I = 1:10, MOD(I,2) == 0 )
$  A(I) = SUM( A(1:I), MASK = MOD([1:I],2) == 0 )
```

Both statements assign only to the even-numbered elements of A, but in the first statement, SUM scans all elements of A in performing its calculation, while the second reference scans only the even-numbered elements of A.

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

10.3.1 Examples of FORALL Statements

The examples in this section illustrate some of the power and expressiveness of the FORALL statement. The examples assume the following parameter and array declarations:

```
      INTEGER P
      PARAMETER ( M=6, N=5, P=30 )
      INTEGER A(M,N), B(N,M), C(M,N,4), D(N,N), V(N), X(P)
CMF$  LAYOUT A(,), B(,), C(,,), D(,), V(), X()

C     Zeros the upper right triangle of D
      FORALL( I=1:N, J=1:N, I < J ) D(I,J) = 0

C     Assigns each array element its index along the 2nd axis
      FORALL( I=1:N ) C(:,I,:) = I

C     Assigns consecutive integers to all elements of array A
      FORALL( I=1:M, J=1:N ) A(I,J) = (I-1)*N + J - 1

C     Both statements replicate vector V along the second
C       dimension of an array D
      FORALL( J=1:N ) D(:,J) = V
      D = SPREAD( V, 2, N )

C     Extracts a matrix diagonal and assigns it to a vector
      FORALL( I=1:N ) V(I) = D(I,I)

C     "Rotates" the square matrix A counter-clockwise
      FORALL( I=1:N, J=1:N ) A(I,J) = A(J,N-I+1)

C     Both statements transpose an array
      FORALL( I=1:N, J=1:N ) A(I,J) = A(J,I)
      A = TRANSPOSE( A )
```

The following code fragment assigns to each element of an array the subscript order value of that element, for a three-dimensional integer array A with arbitrarily-chosen lower and upper bounds in each dimension. This code is a direct implementation of the formulas for calculating subscript order value given in Table 1, p. 76.

```
      INTEGER A(-2:3,-1:4,10000:10003)
CMF$ LAYOUT A(,,)   ! optional

      INTEGER L(3) ! declared lower bounds
      INTEGER U(3) ! declared upper bounds
      INTEGER S(3) ! declared sizes
      INTEGER I1, I2, I3 ! FORALL indexes

      L = LBOUND(A)
      U = UBOUND(A)
      S = MAX( U-L+1, 0 )
```


Copyright © 1989–1994 Thinking Machines Corporation

```
 FORALL( I1=L(1):U(1), I2=L(2):U(2), I3=L(3):U(3) )
$  A(I1,I2,I3) =
$       1 + ( I1 - L(1) )
$         + ( I2 - L(2) )*S(1)
$         + ( I3 - L(3) )*S(2)*S(1)
 ...
```

The FORALL statement in the code above can be replaced by the equivalent

```
S = SHAPE(A)
A = RESHAPE( MOLD=S, SOURCE=[1:SIZE(A)] )
```

illustrating that, although the FORALL statement is very powerful, it may not always be the most appropriate choice.

NOTE: When a program is 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 the function RESHAPE compatible with Fortran 90. (See Chapter 17.)

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