Chapter 8

Parallel Communication

Parallel communication, the exchange of data between processing nodes via the CM's internal communications network, is inherently more expensive in terms of execution time than parallel computation. Parallel communication is handled by a PM call to a low-level CMRTS routine, which orchestrates the exchange of information among processing nodes. Since this CMRTS call cannot be put into a PE code block, it does not execute as quickly as the pure computation speeds available on the nodes themselves. In addition, since a communications call breaks up PE code blocks, it incurs the extra overhead of the startup time for the PE code block that follows the communications step.

However, this is not to say that parallel communication should be avoided entirely; merely that you should consider the best use for what little communication your program needs. Any parallel algorithm is going to require at least some communication to get data into the right PE memory banks for computation.

The trick is to make this necessary communication as unobtrusive as possible, and to avoid the use of implicit communication (such as when the compiler must generate temporary arrays and move data into the right layout to perform a parallel computation).

This chapter discusses the way the CM Fortran compiler deals with statements that involve parallel computation, showing the ways in which both you and the compiler can simplify and reduce the amount of communication needed in a program.

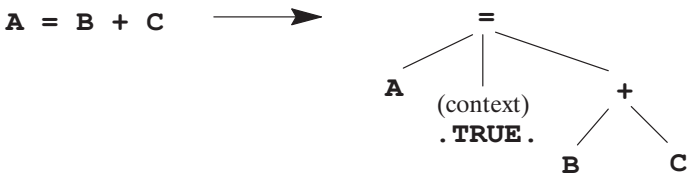
Copyright © 1994 Thinking Machines Corporation

8.1 Compilation of Communication

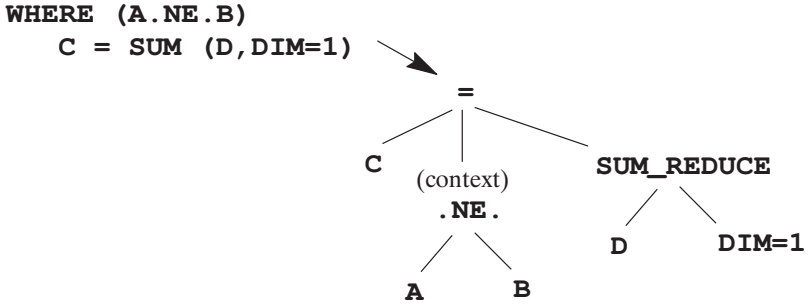
8.1.1 Location Analysis

The CM Fortran compiler identifies all communication, either implicit or explicit, in a program by the following method:

- The compiler converts CM Fortran source lines into *expression trees*. The nodes in these trees represent atomic operators (at the source code level), with their arguments. For example,



  In many cases, a node will also include the *context* of an expression; that is, a description of what part of the arrays will be affected by the operation. It may be an unmasked operation, like the assignment statement above, with a context that is **.TRUE.** everywhere, or it may be a masked expression, like this **WHERE** statement:



- The compiler then assigns a *location* to each node in the tree, essentially a description of the array layout and context in use at that point.

- Where the location differs between a given node and its parent node in the expression tree, the compiler generates appropriate temporary arrays and parallel communication calls to transform the data into an appropriate common layout.

At the completion of this process, the expression tree represents all communication calls required by the program. The number and kind of communication calls may be altered by various optimizations, but the result

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

is the same: a complete map of the parallel communication required by the program.

Note: Any expression involving an aligned array is first translated into an equivalent expression, converting the aligned array into the geometry of the array it is aligned with. For example, the aligned array reference in

```
      REAL A(N), B(M,N+1)
CMF$  ALIGN A(I) WITH B(1,I+1)
      A(1:10) = 5
```

would be translated into something like

```
      REAL A1(M,N+1), B(M,N+1)
      A1(1,2:11) = 5
```

8.1.2 Location Equivalence

The location assigned to each node in an expression tree has two components:

- A *compile-time geometry*: A list of axis descriptors for all arrays in use at that node, consisting of lower and upper axis bounds, axis ordering, axis weights, etc. Basically, this gives the overall shape and layout of the arrays.
- A *compile-time context*: A list of axis subscript expressions, basically describing what part of the arrays are to be operated on. (This is much like a triplet-specified array section.) The compile-time context can be one of three types:
  - *scalar* — an array index specified by a scalar variable
  - *vector* — a vector-valued subscript (index specified as a vector)
  - *triplet* — an index specified as a *start*:*stop*:*stride* triplet

Two locations are considered equivalent if they have the same compile-time geometry and context.

Copyright © 1994 Thinking Machines Corporation

8.1.3 Combining of Locations

At each node of an expression tree, the compiler designates a *preferred location*, that is, a preferred array layout and context into which the compiler will move arrays with other layouts and contexts — this is essentially how the compiler decides what communication operators and temporary arrays to use to move data from one location to another.

In an array assignment, such as **`X=(A+B)`**, for example, the left-hand side of the expression (**`X`** in this case) is the preferred location. If there is a vector context involved, the vector context determines the preferred location. (For example, in **`X(I)=(A(I)+B(I))`**, where **`I`** is a vector array, **`I`** dictates the preferred location.)

The right-hand side of the expression is then examined by a post-order traversal of the expression tree (each node is examined after its children). The location of a binary operation is determined from the preferred location and the locations of the two arguments. (There are some special cases involving scalar contexts, but for the moment we'll ignore them.)

The rules for combining locations are

- If the two arguments are at the same location, that location is the location for the binary operation, unless there is a vector context. If there is vector context, its location becomes the location for the binary operation *and* the new preferred location.
- If the two arguments are not at the same location, the compiler chooses the location by one of the following criteria:
  - the argument that matches the preferred location, if any
  - the argument that matches the location, but is associated with a user-supplied mask
  - the argument that is most easily translated (shifted) to the preferred location
  - a third location that is the best fit between the preferred location and the two argument locations

The last possibility is illustrated by the following example:

```
SUBROUTINE OTHER_LOC(C,A,B,N)
INTEGER N
```


Copyright © 1994 Thinking Machines Corporation

```
      REAL  C(N,N),A(N,N),B(N,N)

      FORALL  (I  =  2:N-1,  J  =  2:N-1)
     $     C(I,J)  =  A(I+1,J+1)  +  B(I+1,J-1)

      RETURN
      END
```

In this example, the addition can be computed by shifting both **A** and **B** along dimension 2, after which the result can be assigned to **C** after a shift in dimension 1. Since there is no easier way to assign a location to the expression, these three shifts are used to relate the component expressions.

8.1.4 Location Comparison Restrictions

There are some restrictions on location comparion:

- The compiler does not rearrange complex expressions into a canonical tree form, so in some cases it may not recognize that two expressions in fact refer to the same location. For example, **(N+M)+1** and **N+(M+1)** are not considered the same. Some special cases, such as binary expressions, are recognized (**N+M** = **M+N**, for example).
- Similarly, expressions used in array contexts are not put in a canonical form, so that the compiler may not recognize that two array locations are in fact the same. For example, the two triplets **(i:n+i-1)** and **(i:n+i+j-1-j)** are not considered the same by the compiler.
- Assumed-shape arrays are always assumed to have distinct locations, unless explicit **ALIGN** directives are used to relate them.
- Since the compiler does not look ahead when combining locations, the preference in case of ties is the location with the smaller internal identifier number, which can lead to suboptimal performance in some cases. The rationale for choosing the one with the smaller identification number is to make the choice consistent in cases where the argument pairs occur in multiple places in the expression tree.

Copyright © 1994 Thinking Machines Corporation

8.1.5 Communication Operator Selection

When the location analysis of a statement determines that communication between processors is necessary, the compiler generates one of the following kinds of data motion, listed in order of compiler preference:

- **news** — NEWS communication (regular grid shifts)
- **cross_geometry_move** — axis rearrangement
- **blockmove** — run-time choice between send or local copy
- **send, get** — general many-to-many communication

(There are other communication operators, such as global reductions used by functions such as **SUM**, and the **SPREAD** function used by **FORALL**, but these four classes are the main kinds of permutative communication.)

NEWS

Two locations are defined to have a **NEWS** relationship if and only if they can be aligned by a constant shift along each dimension. More precisely, this condition is established by determining that geometries agree, strides agree, and the difference in lower bounds is a constant. There is no check on the upper bounds, however, so non-conforming assignments (which are, after all, erroneous) can slip through.

Some examples of statements producing **NEWS** communication are:

```
REAL A(200), B(200), C(200)
A(1:199) = B(2:200)
A(1:199:3) = B(2:200:3)
A(I:I+10) = B(I+5:I+15)
```

The listing file produced by **-list** identifies this communication routine type as **NEWS**.

Cross Geometry Move (VP Move)

The routine **cross_geometry_move** (also sometimes called **vpmove**, after an earlier implementation on the CM-2) is a run-time system operation that maps an array or array section of one rank to another of the same or different rank by making a correspondence between pairs of whole array axes.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

The compiler does not use **cross_geometry_move** in situations that involve indirection by vector-valued subscripts or **FORALL**. Instead, a vectored **send** is used. Also, the compiler does not use **cross_geometry_move** in cases where the source and the destination have the same geometry, such as a transpose or more general axis permutation. In these cases a **send** or some other operation is used.

To determine correspondence between axes of the source and destination arrays (or sections), the compiler first checks that each axis has stride 1 and lower bound 0, zero-based (a lower bound of 1 from user's point of view). It also checks that corresponding parallel axes have the same extents and the same declared ordering. Thus, for example, it will not use **cross_geometry_move** to map an axis declared **:SEND** to an axis declared **:NEWS**, even though these two axis orderings are identical on the CM-5.

The listing file produced by **-list** identifies this communication routine type as **VPMOVE**.

Block Move

The **blockmove** communication type can be used in place of **cross_geometry_move** and **send** in some cases to speed communication where the geometries of arrays are not known at compile time (allocatable arrays, etc.). The **blockmove** communication type turns into either a **send** or a local array copy at run time, according to whether the array geometries involved are the same.

The listing file produced by **-list** identifies this communication routine type as **BLOCKMOVE**.

Sends and Gets

The RTS functions **send** and **get** implement general many-to-many communication of data. This is more costly in terms of execution time than the grid communication expressed by **news** and **cross_geometry_move**, but is necessary where movement of data between processors can't be expressed as a simple shifting of data along array axes or a permutation of axes.

The compiler prefers **send** to **get** because **send** is usually about twice as fast. However, there are cases where **get** is preferable. For example, in the following code, the compiler generates a **get**, which is faster than **send**

Copyright © 1994 Thinking Machines Corporation

because of the difference in geometry sizes (it's quicker to get specific elements out of a large array than to have to decide for each element of the large array whether or not to send it to the smaller destination array).

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

In general, arbitrary many-to-many movements of array data produce **send** operations. Statements with array indexing on the right side of an assignment, or vector subscripts on the left side, can produce **get** communication:

```
      REAL A(200), B(300)
      INTEGER V(200)
C***  SEND COMMUNICATION
      A = B(1:200)
C***  GET COMMUNICATION
      A = B(V)
      A(2:100:2) = B(V(2:100:2))
```

The listing file produced by **-list** identifies this communication routine type as **SEND** or **GET**.

8.1.6 Comparing Communication Functions

As a general rule of thumb, **news** is faster than **send**, and **send** is faster than **get**. The **cross_geometry_move** and **blockmove** operators are currently on the order of **send** in performance, but can be expected to improve in future releases.

This rule holds only for equal amounts of data moving equal distances, of course; there are exceptions to this clean-cut rule. A **send** operation that always exchanges information between memory locations that are on the same PE may be as fast or faster than a **news** operation that moves a lot of data between nodes. And, as shown in the section above on **send** and **get**, a **get** into a 1-D array may have a better performance time than a **send** from a 2-D array.

The performance of communication operators depends on a number of factors:

- set-up time (CMRTS function call, calls to PN communication routines, etc.)

Copyright © 1994 Thinking Machines Corporation

- subgrid sizes (large subgrids may mean more on-processor communication, with better performance, or many off-processor exchanges of data, reducing performance, depending on how the data is moving)
- regularity of communication pattern (if communication can be expressed as a **news** operation, it can be handled faster than a **send**)
- frequency of collisions (do many data values try to go to the same node? This can affect **send**/**get** performance)

8.2 Intrinsics

The compiler processes intrinsic function calls by calling RTS communication functions. However, the compiler recognizes intrinsics and processes them more efficiently than general external procedures (library routines or user-written procedures). In particular, it does not canonicalize array sections passed to an intrinsic function, as it does with external procedures. Here we examine how the compiler processes spreads, reductions, and shifts when expressed with intrinsic functions.

8.2.1 Spreads

Spreads are processed by first aligning the source with the location determined by the destination with the spread axis suppressed. Note, however, that the location of the destination is actually unknown when a spread is processed. Therefore the location of the destination is approximated by the current preferred location.

Here are two examples that show the effects of this algorithm:

```
      SUBROUTINE SPREAD_EXP(A,B,C,D,N1,N2)
      INTEGER N1,N2
      REAL A(N1,N2),B(N1,N2),C(N1),D(N1,N2)
CMF$  LAYOUT A(100:NEWS,:NEWS)
CMF$  LAYOUT B(:NEWS,:NEWS)
CMF$  LAYOUT C(:NEWS)
CMF$  LAYOUT D(:NEWS,:NEWS)
      A = B + (SPREAD(C,DIM=2,NCOPIES=N2) + D)
      RETURN
      END
```


Copyright © 1994 Thinking Machines Corporation

This subroutine generates a **cross_geometry_move**, two **blockmove** operations, and a **spread**. In contrast, the following code generates only one **blockmove** operation:

```
        SUBROUTINE SPREAD_EXP_1(A,B,C,D,N1,N2)
        INTEGER N1,N2
        REAL A(N1,N2),B(N1,N2),C(N1),D(N1,N2)
        REAL TMP(N1,N2)
CMF$    LAYOUT A(100:NEWS,:NEWS)
CMF$    LAYOUT B(:NEWS,:NEWS)
CMF$    LAYOUT C(:NEWS)
CMF$    LAYOUT D(:NEWS,:NEWS)
CMF$    LAYOUT TMP(:NEWS,:NEWS)
        TMP = B + (SPREAD(C,DIM=2,NCOPIES=N2) + D)
        A = TMP
        RETURN
        END
```

Here, the result of the **spread** is already aligned with **D**, **B**, and **TMP**. Finally the entire **TMP** is **blockmove**'d into alignment with **A**.

8.2.2 Reductions

Reductions reduce to the location determined by the source with the reduction axis contextualized with a scalar. Note that this means that

```
REAL A(N1), B(N1,N2)
A = SUM(B,DIM=2)
```

generates a **SEND** of the **SUM** result, whereas

```
REAL A(N1,N2), B(N1,N2)
A(:,1) = SUM(B,DIM=2)
```

generates a contextualized store of the **SUM** result.

8.2.3 Shifts

The shift intrinsics **CSHIFT** and **EOSHIFT** operate on array sections in an interesting fashion. To understand this, one must first understand that the RTS primitives **cshift** and **eoshift** operate on arbitrary initial stride-1

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

segments of axes. Therefore when a shift operation is applied to a non-initial stride-1 segment such as **A(3:K)**, the first step is to shift the array using **news** operations until the segment is in an initial segment. Then the appropriate RTS function is called.

So, for example, if **A** and **B** are aligned arrays of length **2*N**, the following generates one **news** and one RTS **cshift**:

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

This example generates one **news** and one RTS **eoshift**:

```
A(1:N-1) = EOSHIFT(B(2:N),SHIFT=1,DIM=1,BOUNDARY=0.0)
```

These two examples each generate a **news** operation, an RTS **cshift** or **eoshift**, and a second **news** operation to store the result:

```
A(3:N+1) =  CSHIFT(B(2:N),SHIFT=-1,DIM=1)
A(3:N+1) = EOSHIFT(B(2:N),SHIFT=-1,DIM=1,BOUNDARY=0.0)
```

Finally, this generates one **send**, one RTS **cshift**, and then another **send** for the assignment.

```
A(::2) = CSHIFT(B(::2),SHIFT=1,DIM=1)
```

8.3 Intrinsics That Generate Communication

The intrinsics that result in communication are

MAXLOC	**MINLOC**	**MAXVAL**	**MINVAL**	**COUNT**
SUM	**PRODUCT**	**BXOR**	**BOR**	**BAND**
MAXVAL	**MINVAL**	**COUNT**	**ANY**	**ALL**
PROJECT	**DIAGONAL**	**CSHIFT**	**EOSHIFT**	**TRANSPOSE**
MATMUL	**PACK**	**UNPACK**	**RESHAPE**	

Shifts on serial axes generate communication if expressed wtih the intrinsics **CSHIFT** and **EOSHIFT**, but not when expressed with assignments of array sections. The utility procedures **CMF_SCAN_** also generate communication on serial axes. Bitwise logical functions (**BTEST**, **IAND**, **IOR**, etc.) called on serial axes also cause communication to be generated.

Copyright © 1994 Thinking Machines Corporation

8.4 Array Argument Passing

The general rules for passing CM arrays as arguments:

- Whole arrays are always passed in place.
- By default the callee inherits the geometry of the argument. If there is a conflicting **LAYOUT** directive in the callee, the results are unpredictable.
- The compiler assumes that all arguments are distinct. If you pass the same actual array to different dummy arguments or pass both an array and an alias to it to the same subprogram, you are responsible for preventing dependence problems that would give incorrect results.

8.4.1 Noncanonical Arrays as Arguments

When a noncanonical array is passed as an argument, the dummy array in the callee must have the same noncanonical layout as the actual argument. Like mismatched shape or home, mismatched array layout across procedure boundaries causes a run-time error or incorrect results.

The compiler can catch this error if you provide an interface block whenever you pass a noncanonical array as an argument. (Interface blocks are described in the *CM Fortran Language Reference Manual*.) Thus, the directive line that specifies the array layout should appear in three places: in the specification part of the calling routine, in the specification part of the procedure, and in the interface block that duplicates the procedure interface within the calling routine.

8.4.2 Array Sections as Arguments

In most cases, an array section is moved, usually via **send**, to a canonical temporary before being passed as an argument. (An array section that begins at the array origin and is the same rank as the parent array is passed in place.) Array sections are copied out only if modified by the callee.

To avoid this communication, pass in the whole array along with indices that indicate the section of interest. Then have the subprogram select the array

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

section. For example, to have a subroutine operate on a 16-element section of vector **V(1024)** beginning at **V(128)**:

```
CALL  SUB(V,128)
...
SUBROUTINE  SUB(ARRAY,INDEX)
...
ARRAY(INDEX:INDEX+15)  =  ...
```

Note, however, that this approach typically results in performing operations on all 1024 elements and masking out all but 16. And, depending on what the subroutine does, the 16 elements could end up being sent to a temporary after all.

NOTE

Array sections specified by triplets or scalars on serial axes are passed in place if, and only if, the section is contiguous in memory.

8.5 FORALL Optimizations

The **FORALL** statement can generate efficient parallel code for a wide range of expressions. Array assignments made via a **FORALL** statement are treated the same as array assignments made via triplet indexing (and in many cases the two are equivalent). Quite often, a **FORALL** statement can be directly transformed by the compiler into an equally efficient triplet-indexed array assignment. You can use the **-forall_print** compiler switch to instruct the compiler to display all **FORALL** transformations it makes.

However, the **FORALL** statement also makes it effortless for you to use complex and inefficient forms of communication. You should carefully check the **.lis** listing file and the **-forall_print** output generated by the compiler to see what kind of communication operations a **FORALL** is transformed into.

Copyright © 1994 Thinking Machines Corporation

See the *CM Fortran Language Reference Manual* for the definition of the **FORALL** statement. This section presents examples of the kinds of optimizations that the compiler can perform on **FORALL** statements.

8.5.1 Operations Now Optimized

Gathers/Scatters on Serial Dimensions

FORALL can be used to express gather/scatter operations. An example of a scatter is:

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

A gather is:

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

When **FORALL** does a gather or scatter on a serial axis, the compiler can determine that the operation is entirely local and uses the fast indirect addressing hardware.

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(INDEX(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**.

Implementation Note: Here is a detailed definition of the class of indirect references that can be handled without **GET** or **SEND**. Let **A** be an array of rank $n+s$, laid out with n **:NEWS** axes and s **:SERIAL** axes. A **SERIAL-INDIRECT** memory reference is given by a **FORALL** expression with

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

FORALL indices **J1,...,JM, I1,...,IN** with **M <= S**. The index on **:NEWS** axis **K** must be **IK**. The index on **:SERIAL** axis **Jk** can be any elemental expression involving constants, the **FORALL** indices, and other conforming **SERIAL-INDIRECT** memory references. Two **SERIAL-INDIRECT** references are conforming if the arrays have the same "reduced geometry" (i.e., the same geometry with serial axes removed).

For example:

```
CMF$  LAYOUT A(:NEWS,:NEWS,:SERIAL,:SERIAL)
CMF$  LAYOUT B(:SERIAL,:NEWS,:SERIAL,:NEWS)
CMF$  LAYOUT C(:SERIAL,:NEWS,:NEWS)
CMF$  LAYOUT D(:SERIAL,:SERIAL,:NEWS,:NEWS)

      FORALL(J1,J2,I1,I2)
     &  D(J1,J2,I1,I2) =
     &  A(I1,I2,I1+J1,B(J2,I1,I2 + C(J1,I1,I2),I2))
```

Spread Operations on Serial Dimensions

FORALL can be used to express spread operations. When done on serial axes the compiler can determine that the operation is entirely local and generates local memory accesses instead of communication routines.

```
      REAL A(10,10), B(10)
CMF$  LAYOUT A(:SERIAL,:NEWS), B(:NEWS)
      ...
      FORALL(I=1:10) A(I,:) = B(:)
```

Scan Operations

FORALL now generates run-time scan routines, including conditional and segmented scans and scans of scans, rather than generating the less efficient reduction-of-spread expressions. **FORALL** now gives performance comparable to the utility library procedures **CMF_SCAN_***combiner*.

Simple Scans

- Upward inclusive 1D scan, axis 1:

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


Copyright © 1994 Thinking Machines Corporation

- 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))
```

- 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 Library 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 comparable 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)

Copyright © 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:

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

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))
```

- 1D upward, inclusive, segment:

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

Copyright © 1994 Thinking Machines Corporation

```
 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 the 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 **B(MOST_NEG_INT:I)**, which is an error. 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))
```

- 1D upward, exclusive, start:

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


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

- 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)
```

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))
```

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)
```


Copyright © 1994 Thinking Machines Corporation

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 7 or less. For arrays of higher rank, use the utility library procedure **CMF_SEND_***combiner* for best performance.

8.5.2 Expressions Now Parallelized

The **FORALL** statement now accepts the following expressions and generates parallel expressions:

- References to all intrinsic functions and a few specific function calls that are listed below under "restrictions."

  Previous versions of **FORALL** always serialized the transformational functions **CSHIFT**, **EOSHIFT**, **DIAGONAL**, **DOT_PRODUCT**, **FIRSTLOC**, **LASTLOC**, **MATMUL**, **MAXLOC**, **MINLOC**, **PACK**, **PROJECT**, **REPLICATE**, **RESHAPE**, **SPREAD**, **TRANSPOSE**, **UNPACK**, and the inquiry functions **SIZE**, **SHAPE**, **LBOUND**, **UBOUND**, and **RANK**. As of Version 2.0, **FORALL** accepts these functions, except in the cases noted below.

- Use of a **FORALL** index variable in the following ways:
  - in a triplet subscript, such as `A(1:I)` or `B(I:I+5:2)`
  - in an array constructor, such as `[I]` or `[1:I]`
  - as an argument to a statement function, such as `FOO(I)`

8.5.3 Permanent Restrictions

The following cause **FORALL** to execute serially:

- Reference to a character variable, such as `STRING(I)`.
- Reference to an external function.
- "Too many" triplet axes, such as **A(1:I,1:J,1:K,1:L)**. The operation executes serially if the number of vector axes (those with colons) plus the number of distinct **FORALL** indices referenced is above a certain threshold. The threshold in the current release is 7. The

Copyright © 1994 Thinking Machines Corporation

expression above references 4 vector axes and 4 indices, for a total of 8, and thus executes serially.

- **CSHIFT** or **EOSHIFT** of a triplet axis using a **FORALL** index, such as **CSHIFT(B(1:I),SHIFT=1,DIM=1)**.
- A nonconstant **DIM** argument, that is, a dimension not known at compile time, to any reduction function, or to the transformational functions **CSHIFT** or **EOSHIFT**, **FIRSTLOC** or **LASTLOC**, **PROJECT**, **REPLICATE**, **SPREAD**, or to the inquiry functions **UBOUND**, **LBOUND**, or **SIZE**.

Another restriction is that **FORALL** does not support assumed-size arrays (those declared with * as the last axis extent) or the type **CHARACTER**.

8.5.4 Using Alternative Forms of FORALL

Because of **FORALL**'s expressive power, there are often several ways to express a given parallel operation. However, the implementations, and thus the performance characteristics, of the alternative forms are usually different. You should examine closely the processed form of the **FORALL** statements you use. For example, both the following statements extract the diagonal elements from one matrix and store them in the diagonal of another matrix:

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

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

However, the first statement uses scatter/gather communication operators to collect the diagonal elements of **B** and store them in **A**, while the second statement simply does an array assignment under a **WHERE** mask selecting the diagonal elements. The latter implementation, where it does not involve communication to do the assignment (that is, assuming the layouts of the two arrays are the same) will most likely be the more efficient of the two, but the relative performance will depend on the array layouts, the use of serial axes, etc.

Copyright © 1994 Thinking Machines Corporation

In general, you should always examine closely the kinds of statements your **FORALL** expressions are transformed into (by using the compiler switch **-forall_print**), and bear in mind how the impact of array layout and other considerations will affect them.

For example, here is the relevant portion of the **-forall_print** output for these two **FORALL** statements above, showing how they are optimized:

```
Entering Forall #1:
      FORALL(I=1:N) A(I,I) = B(I,I)

Processed Forall:
      A = SCATTER(GATHER(B,-1,[1:N]-LBOUND(B,1),
                             [1:N]-LBOUND(B,2)),
                  A,-1,[1:N]-LBOUND(A,1),
                       [1:N]-LBOUND(A,2))


Entering Forall #2:
      FORALL(I=1:N,J=1:N,I.EQ.J) A(I,J) = B(I,J)

Processed Forall:
      WHERE(SPREAD([1:N],2,MAX(N,0)) .EQ.
            SPREAD([1:N],1,MAX(N,0)))
         A(1:N,1:N) = B(1:N,1:N)
      ENDWHERE
```


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

8.6 Simplifying Communication — Summary

In summary, to get the most efficient performance out of your program:

- Use the **`-list`** and **`-forall_print`** compiler switches to check on the types of communication being used in your program.
- Examine the type(s) of communication occuring in your code, and decide whether or not they are necessary, or whether more efficient, regular types will do the same job.
- To reduce unintended communication and temporary copying of arrays, and make PE code blocks more efficient, try separating statements into elemental computation and assignments involving a single type of commuication.
- Keep in mind the layouts of the arrays you're working with, and remember that small subgrid sizes and incompatible axis declarations can increase communication in your code.

Copyright © 1994 Thinking Machines Corporation