Chapter 15

Communication Primitives

This chapter describes the following CMSSL communication primitives:

- polyshift
- all-to-all rotation
- all-to-all broadcast
- all-to-all reduction
- matrix transpose
- sparse gather and scatter utilities
- sparse vector gather and vector scatter utilities
- block gather and scatter utilities
- mesh partitioning and reordering of pointers
- partitioned gather and scatter utilities
- partitioned vector gather and scatter utilities
- communication compiler
- vector move (extract and deposit)
- computation of block cyclic permutations
- permutation along an axis
- combination of permutations
- send-to-NEWS and NEWS-to-send reordering
- zeroing routine

Copyright © 1994 Thinking Machines Corporation

Sections 15.1 through 15.19 introduce these operations. Each section is followed by a man page that describes the routine(s) in detail. Section 15.21 provides references.

Copyright © 1994 Thinking Machines Corporation

15.1 Polyshift

Many scientific applications make extensive use of array shifts in more than one direction and/or dimension in an array geometry. One well-known example is "stencils" used in solving partial differential equations (PDEs) by explicit finite difference methods. Similar communication patterns are encountered in other applications. For example, in quantum chromodynamics one needs to send (3 x n) complex matrices in each direction of a four-dimensional lattice. Multiple array shifts are also useful in many molecular dynamics codes. In CMSSL, such multiple array shifts are called "polyshifts" (PSHIFTs). They can be recognized in CM Fortran code by a sequence of **CSHIFT** and/or **EOSHIFT** calls in multiple directions of multiple dimensions, with no data dependencies among the arguments and the results of the shifts. There is a potential performance gain in recognizing a polyshift communications pattern, and calling specially developed routines for doing the shifts. In addition, application programs that utilize calls to polyshift routines can benefit from enhanced readability and maintainability. This section describes the implementation of a high-level interface for calling polyshift routines from CM Fortran.

15.1.1 The Polyshift Routines

Given a set of source and destination CM arrays, together with the types, dimensions, and distances of the shifts to be performed, the polyshift routines shift the arrays in multiple directions of multiple dimensions. To perform a polyshift operation (or multiple polyshift operations, sequentially), you must follow these steps:

1. Call either the **pshift_setup** routine or the **pshift_setup_looped** routine.

   These routines are identical on the CM-5; both are provided for compatibility with the CM-200 library. The setup routine creates a setup structure for the specified communication pattern and returns an ID corresponding to that setup structure. You must supply this ID in subsequent calls to **pshift** and **deallocate_pshift_setup**.

2. Call the **pshift** routine.

   This routine performs the polyshift. To perform more than one polyshift operation using the same communication pattern, follow one call to **pshift_setup** or **pshift_setup_looped** with multiple calls to **pshift.**

Copyright © 1994 Thinking Machines Corporation

If the communication pattern changes, you must start with Step 1 again, since multiple communication patterns require multiple setup calls. For example, you might require both a five-point and a nine-point stencil in the solution of a PDE; these would be set up in two different calls to **pshift_setup** or **pshift_setup_looped**, and identified with two IDs. (Refer to the online 9-point stencil example; the pathname is given in the man page.)

3. After all **pshift** calls associated with the same **pshift_setup** or **pshift_setup_looped** call (that is, when a communication pattern is no longer needed), call the **deallocate_pshift_setup** routine to deallocate the setup structure created by the setup routine.

You may have more than one setup active at a time; that is, you may call either or both setup routines more than once without calling the deallocation routine. When you call **pshift** or **deallocate_pshift_setup**, the setup ID you supply identifies the setup you want to use or deallocate.

The code skeleton below shows a CM Fortran program using the polyshift routines.

```
id = pshift_setup (...)
call pshift (..., id, ...)
call deallocate_pshift_setup (id)
```

In this example, the *id* argument is the polyshift setup ID; multiple IDs may be active at the same time.

NOTE

The **pshift** routine allows only one shift operation per direction per axis.

Always include the header file **/usr/include/cm/cmssl-cmf.h** in any program unit that calls either of the functions **pshift_setup** or **pshift_setup_looped** so that the correct return type is declared. This header file is also required for calls to **pshift** in order to use the predefined parameters **CMSSL_CSHIFT**, **CMSSL_EOSHIFT_0**, etc., defined in the man page.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

15.1.2 Optimization Recommendations

Follow these recommendations to optimize your code:

- Use array sizes that result in no padding. PSHIFT performance is best when the array is not padded along any axis *i* on which shifts are being performed. When this requirement is not met, PSHIFT performance is approximately the same as for the equivalent call to **CSHIFT** or **EOSHIFT**.

- Attempt to "balance" your subgrid. Call **CMF_DESCRIBE_ARRAY** (described in the CM Fortran documentation set) to determine your subgrid size. Then use the **weight** parameter in the **LAYOUT** compiler directive to adjust the subgrid size (this must be done empirically). For example, use

  ```
  CMF$ LAYOUT X (2:NEWS, 1:NEWS)
  ```

  to increase the x-axis weight.

- Use array-valued boundaries for end-off shifts only when absolutely necessary.

Copyright © 1994 Thinking Machines Corporation

Polyshift

Given a set of source and destination CM arrays, together with the types, dimensions, and distances of the shifts to be performed, the polyshift routines shift the arrays in multiple directions of multiple dimensions.

SYNTAX

setup_id = **pshift_setup** (*n*, ***cm_array***, *ier*,
type_1, *dim_1*, *dist_1*,
type_2, *dim_2*, *dist_2*,
.
.
.
type_n, *dim_n*, *dist_n*)

setup_id = **pshift_setup_looped** (*n*, ***cm_array***, *ier*,
type_1, *dim_1*, *dist_1*,
type_2, *dim_2*, *dist_2*,
.
.
.
type_n, *dim_n*, *dist_n*)

pshift (*n*, *setup_id*, *ier*,
type_1, ***A_1***, ***B_1***, *dim_1*, *dist_1*[, ***bdry_1***],
type_2, ***A_2***, ***B_2***, *dim_2*, *dist_2*[, ***bdry_2***],
.
.
.
type_n, ***A_n***, ***B_n***, *dim_n*, *dist_n*[, ***bdry_n***])

deallocate_pshift_setup (*setup_id*)

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

ARGUMENTS

n — Input. Scalar integer variable, parameter, or constant. The number of distinct shifts, of the shift types, dimensions, and distances that follow, to do in a single call to **pshift**. *n* must be greater than or equal to 0 and less than or equal to 14. The value you supply to **pshift** must be the same as the value you supplied in the associated **pshift_setup** or **pshift_setup_looped** call. The contents of this argument are not modified.

cm_array — Input. CM array of any logical or numeric data type. This is a "prototypical" array used in deriving the PSHIFT geometry. The arrays ***A_i*** and ***B_i*** that you supply to **pshift** must agree with ***cm_array*** in data type, size, shape, and layout. The contents of this argument are not modified.

setup_id — Input. Scalar integer variable. PSHIFT setup structure identifier that was returned by a previous call to **pshift_setup** or **pshift_setup_looped**. The contents of this argument are not modified.

ier — Output. Return code. Must be a scalar integer variable. Upon return from **pshift_setup** or **pshift_setup_looped**, *ier* contains one of the following codes:

0	Successful return.
2	$n < 0$ or $n > 14$.
3	The setup routine cannot allocate partition manager memory for the setup structure.
4	Some *dim_i* is less than 1 or greater than 7.
5	An unknown shift type was specified; that is, some *type_i* was not one of the five shift types listed under *type_i*, below.
6	You have specified multiple negative shifts on some dimension.
7	You have specified multiple positive shifts on some dimension.

Upon return from **pshift**, *ier* contains one of the following codes:

0	Successful return.

Copyright © 1994 Thinking Machines Corporation

2 *n* does not match the value supplied in the **pshift_setup** or **pshift_setup_looped** call.

3 Some *dim_i* is less than 1 or greater than 7.

4 The array geometry of some ***A_i*** does not match the geometry of the array supplied in the **pshift_setup** or **pshift_setup_looped** call.

5 The array geometry of some ***B_i*** does not match the geometry of the array supplied in the **pshift_setup** or **pshift_setup_looped** call.

6 Some *type_i* does not match the corresponding shift type supplied in the **pshift_setup** or **pshift_setup_looped** call.

7 Some *dist_i* does not match the corresponding distance supplied in the **pshift_setup** or **pshift_setup_looped** call.

8 A boundary array argument is missing from the **pshift** call, or the data type of a boundary array is incorrect.

type_i Input. One of the following predefined CMSSL shift types:

CMSSL_CSHIFT
CMSSL_EOSHIFT_0
CMSSL_EOSHIFT_1
CMSSL_EOSHIFT_SCALAR
CMSSL_EOSHIFT_ARRAY

For the *i*th shift, these give shifts of the type corresponding to the CM Fortran intrinsics (**CSHIFT** or **EOSHIFT**).

CMSSL_EOSHIFT_0 shifts in a 0 of the proper data type at the boundary.

CMSSL_EOSHIFT_1 shifts in a 1 of the proper data type at the boundary.

CMSSL_EOSHIFT_SCALAR shifts in a front-end scalar variable at the boundary. The variable must be specified in the call to **pshift**. The data type must agree with the data type of ***cm_array***.

Copyright © 1994 Thinking Machines Corporation

CMSSL_EOSHIFT_ARRAY shifts in a CM array at the boundary. This array must be specified in the call to **pshift**. The data type must agree with the data type of ***cm_array***. The rank must be one less than the rank of ***cm_array***. Refer to the discussion of **EOSHIFT** in the CM Fortran documentation set for more details. The contents of this argument are not modified.

A_i Output. CM array. The destination of the *i*th shift, as specified by the types, dimensions, and distances that follow. Must agree in data type, size, shape, and layout with the ***cm_array*** passed to **pshift_setup** or **pshift_setup_looped**.

B_i Input. CM array. The source of the *i*th shift, as specified by the types, dimensions, and distances that follow. Must agree in data type, size, shape, and layout with the ***cm_array*** passed to **pshift_setup** or **pshift_setup_looped**. The contents of this argument are not modified.

dim_i Input. Scalar integer variable, parameter, or constant. The dimension along which to perform the *i*th shift. The contents of this argument are not modified.

dist_i Input. Scalar integer variable, parameter, or constant. The distance (number of elements) to shift for the *i*th shift. Positive or negative numbers, as defined in the **CSHIFT** and **EOSHIFT** entries of the *CM Fortran Reference Manual*. The contents of this argument are not modified.

bdry_i Input. Front-end scalar or CM array. The boundary value shifted in if the *i*th shift type is **CMSSL_EOSHIFT_SCALAR** or **CMSSL_EOSHIFT_ARRAY**. The contents of this argument are not modified.

If *type_i* is **CMSSL_EOSHIFT_SCALAR** or **CMSSL_EOSHIFT_ARRAY**, you must supply a ***bdry_i*** argument.

If *type_i* is **CMSSL_CSHIFT**, **CMSSL_EOSHIFT_0**, or **CMSSL_EOSHIFT_1**, do not supply a ***bdry_i*** argument. (CSHIFT does not use a boundary value; **EOSHIFT_0** and **EOSHIFT_1** use boundary values 0 and 1, respectively.)

Copyright © 1994 Thinking Machines Corporation

RETURNED VALUE

setup_id Scalar integer variable. A PSHIFT setup structure identifier returned by **pshift_setup** or **pshift_setup_looped** and required by **pshift** and **deallocate_pshift_setup**.

DESCRIPTION

The polyshift routines perform the following operation:

$$\begin{aligned}
&\boldsymbol{A_1}(,,,i,,,) = \boldsymbol{B_1}(,,,i+dist_1,,,);\\
&\boldsymbol{A_2}(,j,,,,) = \boldsymbol{B_2}(,j+dist_2,,,,);\\
&\quad\vdots \qquad\quad \vdots\\
&\boldsymbol{A_n}(,,,,,k,) = \boldsymbol{B_n}(,,,,,k+dist_n,)
\end{aligned}$$

Usage. To perform a polyshift operation (or multiple polyshift operations, sequentially), you must follow these steps:

1. Call either the **pshift_setup** or the **pshift_setup_looped** routine.

   These routines are identical on the CM-5; both are provided for compatibility with the CM-200 library. The setup routine creates a setup structure for the specified communication pattern and returns an ID corresponding to that setup structure. You must supply this ID in subsequent calls to **pshift** and **deallocate_pshift_setup**.

2. Call the **pshift** routine.

   This routine performs the polyshift. To perform more than one polyshift operation using the same communication pattern, follow one call to **pshift_setup** or **pshift_setup_looped** with multiple calls to **pshift**.

   If the communication pattern changes, you must start with Step 1 again, since multiple communication patterns require multiple setup calls. For example, you might require both a five-point and a nine-point stencil in the solution of a PDE; these would be set up in two different calls to **pshift_setup** or **pshift_setup_looped**, and identified with two IDs. (Refer to the online 9-point stencil example.)

Copyright © 1994 Thinking Machines Corporation

3. After all **pshift** calls associated with the same **pshift_setup** or **pshift_setup_looped** call (that is, when a communication pattern is no longer needed), call the **deallocate_pshift_setup** routine to deallocate the setup structure created by the setup routine.

You may have more than one setup active at a time; that is, you may call either or both setup routines more than once without calling the deallocation routine. When you call **pshift** or **deallocate_pshift_setup**, the setup ID you supply identifies the setup you want to use or deallocate.

Setup Phase. The **pshift_setup** and **pshift_setup_looped** routines allocate a PSHIFT setup structure to be used by **pshift** for performing a specific polyshift, and return an ID for the structure in *setup_id*.

Polyshift Phase. The **pshift** routine executes a specific polyshift communications pattern as determined by the *setup_id* and the source, destination, and boundary arguments.

Deallocation Phase. The **deallocate_pshift_setup** call deallocates the PSHIFT setup structure specified by a given setup ID.

NOTES

Restriction on Number of Shifts. The **pshift** routine performs a maximum of two shifts per array dimension, one in each direction.

Need for Deallocation. The **pshift_setup** and **pshift_setup_looped** calls dynamically allocate partition manager memory. The **deallocate_pshift_setup** call frees this memory.

Include the Header File. Always include the header file **`/usr/include/cm/cmssl-cmf.h`** in any program unit that calls either of the functions **pshift_setup** or **pshift_setup_looped** so that the correct return type is declared. This header file is also required for calls to **pshift** in order to use the predefined parameters **CMSSL_CSHIFT**, **CMSSL_EOSHIFT_0**, etc.

Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the polyshift routines can be found on-line in the subdirectory

```
pshift/cmf
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.2 All-to-All Rotation

NOTE

The routines described in this section were called "all-to-all broadcast" routines in previous versions of this manual.

All-to-all rotation is often used to implement data interactions of the type occuring in many so-called *N*-body computations, in which every particle interacts with every other particle. With an array distributed over a number of memory modules, each of which is associated with a parallel processing element, every module must receive the data from every other module. Another example of an application of all-to-all rotation is matrix-vector multiplication with the matrix distributed with entire rows per processing element, and the vector distributed evenly over the processing elements. Every processing element must gather all the elements of the vector in order to perform the required multiplication.

If every module were to send its data to every other module before any computations start, then for *P* processing elements the memory requirements would grow by a factor of *P*. The all-to-all rotation routine preserves memory by performing the rotation operation as a sequence of permutations. In each communication step, the communication system is used as efficiently as possible.

The all-to-all rotation can be factored into two parts: a local permutation, and all-to-all rotation of local data sets between processing elements. The performance of the CMSSL implementation depends strongly upon the amount of local data motion. This section explains this dependency and provides hints for optimizing computations using the all-to-all rotation.

As with all CMSSL functions, the all-to-all rotation function operates on data arrays as they are allocated by the compiler. Consider a fixed data set *M* structured as a two-dimensional array $A(M/P, P)$ distributed over *N* processing elements, with the first axis of length M/P local to a processing

Copyright © 1994 Thinking Machines Corporation

element. Assume that an all-to-all rotation is performed along the second axis, which has local extent P/N. No local data motion is performed when $P = N$ since the second axis has no local component. When $P > N$, the rotation operation is factored into a local rotation operation of P/N steps for each exchange of $(M/P)(P/N)$ elements between processing elements.

The one-dimensional data structure $A(M)$ requires the most extensive local data motion for a fixed data set M, namely $(M/N)^2$, while no local data motion is required for $P = N$. The inter-processing element data motion is independent of P, but decreases with N.

As an example of how to tune performance by reducing the number of local memory moves, we consider a typical calculation on the CM system (see reference 2 in Section 15.21) as defined by

$$z_i = \sum_{j=1}^{M} F(y_i, x_j) \quad i = 1, ..., M$$

The following pseudocode, using the all-to-all rotation and the Fortran 90 array syntax, is independent of the machine configuration and involves local data motion whenever M is larger than the current number of processing elements being used.

Code 1

```
array:: x(M), y(M), z(M)
do i=1, M
      all-to-all-rotation(i, x(:), axis=1)
      z(:) = z(:) + F(y(:),x(:))
enddo
```

The `:` symbol indicates that the operation is performed on all the array elements at once. The instruction **`all-to-all-rotation(i,x(:),axis=1)`** performs the **`i`**th all-to-all rotation step along the first axis of array **`x`**.

The following alternative encoding of the same computation assumes an N-processing-element configuration. Only N physical rotation operations are performed. The communication cost is minimum.

Copyright © 1994 Thinking Machines Corporation

Code 2

```
array:: x(M/N,N), y(M/N,N), z(M/N,N)
do i=1,N
      all-to-all-rotation(i, x(:,:), axis=2)
      do j=1, M/N
          do k=1,M/N
              z(j,:) = z(j,:) + F(y(j,:), x(k,:))
          enddo
      enddo
enddo
```

CM Fortran implementation of the above pseudocodes (see reference 2) shows that, in practice, a substantial gain can be made when using a two-dimensional data structure, with a serial local axis and a non-local rotation axis, instead of a one-dimensional data structure, given a fixed data set.

Reference 1 shows how to reduce the amount of computation when *F* is symmetric, a particularly important point when solving *N*-body problems on the CM with the all-to-all rotation.

15.2.1 The All-to-All Rotation Routines

Given a real or complex CM array and a selected axis, the all-to-all rotation routines perform a stepwise rotation along the selected axis. Every array element visits every location along the axis. Each step corresponds to a data permutation along the axis, and is typically followed by computations. To perform an all-to-all rotation operation (or multiple rotation operations, sequentially), you must follow these steps:

1. Call the **all_to_all_setup** routine.

   This routine determines how many steps are required to complete an all-to-all rotation of the given array along the selected axis, and determines the permutation pattern for the rotation.

2. Follow the setup routine with a do loop that has one iteration for each required step of the rotation. Each iteration of the loop contains a call to **all_to_all**, which performs one step of the rotation, followed by the computations the application requires.

Copyright © 1994 Thinking Machines Corporation

3. When the rotation is complete, the call **deallocate_all_to_all_setup** to deallocate the space required by the all-to-all rotation.

Only one all-to-all rotation setup can be active at a time; that is, you must deallocate one setup before creating a new one.

The code skeleton below shows a CM Fortran program using the all-to-all rotation. The *step_count* argument is the number of required rotation steps determined by the **all_to_all_setup** routine.

```
call all_to_all_setup (..., step_count, ...)
            .
            .
do i = 1, step_count
   call all_to_all (...)
   where (...)
            .
            .
   elementwise computation
            .
            .
   end where
end do
            .
            .
call deallocate_all_to_all_setup (...)
```


Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

All-to-All Rotation

Given an integer, real, or complex CM array and a selected axis, the all-to-all rotation routines perform a stepwise rotation along the selected axis. Every array element visits every location along the axis. Each step corresponds to a data permutation along the axis, and is typically followed by computations.

SYNTAX

all_to_all_setup (***A, valid_mask****, axis, setup_id, step_count, use_valid, ier*)

all_to_all (***A, valid_mask****, setup_id, step_index, ier*)

deallocate_all_to_all_setup (*setup_id)*

ARGUMENTS

A	The integer, real, or complex CM array to be permuted in the all-to-all loop.
valid_mask	A logical CM array with extents and layout identical to those of ***A***. The **all_to_all_setup** routine sets the values of ***valid_mask***. When you call the **all-to-all** routine, supply the values assigned by the previous, associated **all_to_all_setup** call.
axis	Scalar integer variable. Identifies the axis selected for the all-to-all rotation.
setup_id	Scalar integer variable. Upon return from **all_to_all_setup**, contains a value that you must supply in all subsequent, associated **all_to_all** and **deallocate_all_to_all_setup** calls.
step_count	Scalar integer variable. Upon return from **all_to_all_setup**, contains the number of steps required in the all-to-all rotation loop.
step_index	Scalar integer variable. The current loop index value of the all-to-all loop.

Copyright © 1994 Thinking Machines Corporation

use_valid Scalar logical variable. Upon return, a value of **.true.** indicates that you must use ***valid_mask*** to contextualize the computation. A value of **.false.** indicates that for the particular partition size, array shape, and array layout directives you are using, you need not use ***valid_mask*** to contextualize the computation. For more information, refer to the description and notes below.

ier Scalar integer variable. Upon return from **all_to_all_setup**, contains one of the following codes describing the outcome of the setup request:

0	Success.
−1	***A*** is missing or invalid.
−2	***valid_mask*** is missing or invalid.
−3	*axis* is missing or invalid.
−99	The specified axis is degenerate (has length 1).
other	Execution error.

Upon return from **all_to_all**, contains one of the following codes describing the outcome of the all-to-all step:

0	Success.
−1	***A*** is missing or invalid.
−2	***valid_mask*** is missing or invalid.
−3	The setup object identified by *setup_id* is missing or invalid.
−4	*step_index* is missing or invalid.
other	Execution error.

DESCRIPTION

The all-to-all rotation routines perform the operation

$$A(,,i,) = A(,,P(i),)$$

where P is a permutation based on the *step_index* argument.

Copyright © 1994 Thinking Machines Corporation

Usage. To perform an all-to-all rotation operation (or multiple rotation operations, sequentially), you must follow these steps:

1. Call the **all_to_all_setup** routine.

   This routine determines how many steps are required to complete an all-to-all rotation of the given array along the selected axis, and determines the permutation pattern for the rotation.

2. Follow the setup routine with a do loop that has one iteration for each required step of the rotation. Each iteration of the loop contains a call to **all_to_all**, which performs one step of the rotation, followed by the computations the application requires.

3. When the rotation is complete, the call **deallocate_all_to_all_setup** to deallocate the space required by the all-to-all rotation.

Only one all-to-all rotation setup can be active at a time: you must deallocate one setup before creating a new one.

Setup Phase. Given a CM array and a chosen axis, the **all_to_all_setup** routine determines the permutation pattern for the all-to-all rotation, allocates a setup structure, and places the setup ID in the *setup_id* argument. You must supply *setup_id* as an argument in subsequent **all_to_all** and **deallocate_all_to_all_setup** calls for this rotation.

The **all_to_all_setup** routine also determines the permutation length and returns this number in *step_count*. The permutation length is the number of steps required for the complete all-to-all rotation; that is, the number of permutation steps after which the array elements will be back in their original positions. You must use the value returned in *step_count* as the bound for the all-to-all rotation loop. Because of the manner in which arrays are implemented in CM Fortran, *step_count* may exceed the array axis length. (Refer to the notes below for more information.)

A third returned value, *use_valid*, provides a way to optimize the computations involving the permuted data elements. This argument and the optimization are discussed in the notes below.

Rotation Phase. Given a CM array that has been processed by a previous call to **all_to_all_setup**, and given the value assigned to *setup_id* by **all_to_all_setup**, the **all_to_all** routine permutes all elements of the array along the axis selected in the

Copyright © 1994 Thinking Machines Corporation

all_to_all_setup call. The setup object identified by *setup_id* determines the permutation pattern.

During the all-to-all loop, each element of the array follows a path defined for it at setup time. (See Mathur, K. K. and S. L Johnsson. *All-to-All Communication on the Connection Machine CM-200*. Thinking Machines Corporation Technical Report TR-243, 1992.) The collection of paths followed by the data elements within an all-to-all loop guarantees that each element visits each axis position and returns to its initial location on completion of the loop.

Deallocation Phase. The **deallocate_all_to_all_setup** routine frees the partition manager and parallel processing element storage space that was allocated by the **all_to_all_setup** routine for the all-to-all rotation identified by *setup_id*. Call **deallocate_all_to_all_setup** when the all-to-all rotation loop has finished. Supply the *setup_id* value returned by **all_to_all_setup**.

NOTES

Setup is Private. The *setup_id* argument is private. Application code should never access or modify the contents of an all-to-all setup object.

Step Count is Private. The *step_count* argument is also private. Do not change its value. To ensure that array elements are fully permuted and return to their original positions on completion of the all-to-all rotation loop, be sure to use the value returned in *step_count* as the bound on the do loop.

Permutation Pattern. The order in which array elements visit a given array location depends on the shape of the array, the layout or align directives, and the size of the partition. Computations must not be sensitive to the order of data elements encountered; for example, they should not use the value of *step_index*. (See the Thinking Machines Corporation Technical Report, TR-243, referenced above.)

Two one-dimensional arrays with identical shapes and axis layouts have the same permutation pattern. Likewise, two identical multidimensional arrays have the same permutation pattern. However, within a multidimensional array, the multiple instances of the all-to-all rotation along a selected axis have *different* permutation patterns. For example, if the elements of a two-dimensional array are rotated along rows, then different rows may have different permutation paths. The on-line sample code illustrates this important distinction.

Copyright © 1994 Thinking Machines Corporation

Contextualizing Computations. A CM Fortran array that is allocated on the CM is mapped onto a machine geometry. The machine geometry axes may have greater extents than the corresponding Fortran array axes. The mapping depends on the partition size, array shape, and array layout directives you are using, and therefore varies from program to program and from CM to CM. When they exist, the machine geometry elements that do not contain Fortran array elements are masked out during both computation and communication. The array mapping mechanism is therefore transparent to the user.

However, to optimize performance, the all-to-all rotation routines operate on the whole machine geometry rather than only on the section that contains the Fortran array. The *step_count* value (the number of steps in the all-to-all loop) returned by **all_to_all_setup** is the extent of the machine geometry axis corresponding to the chosen Fortran array axis. The value of *step_count* may therefore exceed the Fortran array axis extent.

Another consequence of the use of the whole machine geometry is that masked machine geometry elements may visit valid Fortran array locations. The logical array ***valid_mask*** required by the **all_to_all_setup** and **all_to_all** routines signals these bad spots at each step of the all-to-all rotation so that you can exclude them from your computations. At each step, you must pass ***valid_mask*** to the **all_to_all** routine. Following the **all_to_all** call, you must contextualize all computations on the permuted elements by enclosing them in a **WHERE** block that references ***valid_mask***. An exception is described under "Optimizing Code," below.

Conversely, valid Fortran array elements may end up in the masked region of the machine geometry, so that not all Fortran array elements are available at each step of the all-to-all rotation. A global reduction operation on the array would give different results at different steps of the rotation. It is guaranteed, however, that all Fortran array elements will have visited all Fortran array axis locations upon completion of the all-to-all rotation.

Optimizing Code. You can save a significant amount of time by omitting the contextualization described above in cases where the rotation axis has the same extent in the Fortran array and the machine geometry. However, it is not safe to remove the contextualization unconditionally because, as mentioned earlier, the mapping of Fortran array to machine geometry depends on variable parameters (partition size, array shape, and array layout directives). We therefore recommend the following safe, though inelegant, solution, which ensures correct results and optimizes the speed of execution. If the *use_valid* argument returned by **all_to_all_setup** is **.false.**, the setup routine has determined that for the particular partition size, array shape, and array layout directives you are using, you do not

Copyright © 1994 Thinking Machines Corporation

need to use a **WHERE** block to contextualize your computations. Therefore, instead of a sequence such as

```
where (valid_mask)
        .
        .
  elementwise computation
        .
        .
end where
```

you may optimize your code by using a sequence such as

```
if (use_valid) then
  where (valid_mask)
        .
        .
    elementwise computation
        .
        .
  end where
 else
        .
        .
  elementwise computation
        .
        .
 end if
```

Specifying the Valid Data Mask Only Once. If you rotate two or more arrays of the same shape and layout along the same axis by making successive calls to **all_to_all** within each iteration of the rotation loop, the arrays may all share the *setup_id, step_count*, and ***valid_mask*** returned by a single **all_to_all_setup** call. In this case, you must include a non-null ***valid_mask*** in exactly one of the **all_to_all** calls; the others must all specify a null valid data mask by passing 0 to the routine in place of ***valid_mask***. The on-line sample program called **example2** provides an example.

Deallocated Setup IDs are Invalid. A *setup_id* that has been deallocated no longer represents a valid setup object. Attempts to use it in a subsequent **all_to_all** call result in errors.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the all-to-all rotation routines can be found on-line in the subdirectory

```
all-to-all/cmf
```

of a CMSSL examples directory whose location is site-specific.

The sample programs illustrate the following important distinction:

- Within a multidimensional array, the multiple instances of the all-to-all rotation along a selected axis have different permutation patterns.
- Two separate arrays with identical shapes and axis layout share the same permutation pattern.

The first example, **`example1`**, performs independent computations on four arrays of size m. Because the computations do not require that the rotations use the same permutation pattern, the program can rotate the multiple instances at once by forming a (4, m) array. This example illustrates the first point above.

In the second example, **`example2`**, the computations require that the rotations of three arrays of size m use the same permutation pattern. Therefore, the program must treat the arrays separately rather than as multiple instances within a single array. The program performs the rotations of the three arrays sequentially.

Copyright © 1994 Thinking Machines Corporation

15.3 All-to-All Broadcast

The **all_to_all_broadcast** routine is similar to the CM Fortran **SPREAD** intrinsic, but offers better performance in many cases. Given source and destination CM arrays of the same type, with rank(destination array) = rank(source array) + 1, **all_to_all_broadcast** copies each instance of a source vector to the destination array and replicates it along a selected axis (the "broadcast axis") of the destination array. You may interleave the active axes with the instance axes differently in the source and destination arrays. The broadcast axis may have any length.

The man page that follows provides syntax information and describes how to obtain the best performance.

Copyright © 1994 Thinking Machines Corporation

All-to-All Broadcast

Given source and destination CM arrays of the same type, with rank(destination array) = rank(source array) + 1, the **all_to_all_broadcast** routine copies each instance of a source vector to the destination array and replicates it along a selected axis (the "broadcast axis") of the destination array.

SYNTAX

all_to_all_broadcast (***y, x,*** *y_vector_axis, y_broadcast_axis, x_vector_axis, ier*)

ARGUMENTS

y	CM array of the same type as ***x*** and rank one greater than that of ***x***. Axis *y_vector_axis* of ***y*** must have the same extent as axis *x_vector_axis* of ***x***. Axis *y_broadcast_axis* of ***y*** defines the extra dimension of ***y***, along which the broadcast is to be performed. All other axes of ***y*** are instance axes, and must match the instance axes of ***x*** in extent and order of declaration. Each source vector lying along axis *x_vector_axis* of ***x*** will be broadcast to multiple destination vectors in ***y***. These destination vectors lie along axis *y_vector_axis* and are counted by axis *y_broadcast_axis*. Thus, upon successful completion, each source vector in ***x*** is copied to ***y*** and replicated, with one copy at each coordinate of axis *y_broadcast_axis* of ***y***.
x	CM array of any rank and type. The source vectors to be copied to ***y*** lie along axis *x_vector_axis*. All other axes are instance axes, and must match the instance axes of ***y*** in extent and order of declaration.
y_vector_axis	Scalar integer variable between 1 and the rank of ***y***, inclusive. Indicates the axis of ***y*** along which the destination vectors lie. Must be different from *y_broadcast_axis*.
y_broadcast_axis	Scalar integer variable between 1 and the rank of ***y***, inclusive. Indicates the axis of ***y*** that counts the replicated destination vectors. May have any length.

Copyright © 1994 Thinking Machines Corporation

x_vector_axis Scalar integer variable between 1 and the rank of ***x***, inclusive. Indicates the axis of ***x*** along which the source vectors lie.

ier Scalar integer variable. Error code. Set to 0 on successful return, or to one of the following values:

1 Invalid array descriptor. The routine was expecting a CM array but was passed either a scalar or a CM array that has not yet been allocated (for example, an array that has neither been used in a CM Fortran array expression nor laid out with a **CMF$LAYOUT** or **CMF$ALIGN** directive).

8 Invalid array ranks. The rank of ***y*** should be one more than the rank of ***x***.

16 Invalid axis numbers. All axis numbers must be between 1 and the rank of corresponding array, inclusive.

32 Corresponding instance axes of ***x*** and ***y*** do not have the same extent.

64 Axes *x_vector_axis* and *y_vector_axis* do not have the same extent.

DESCRIPTION

The **all_to_all_broadcast** routine copies each source vector lying along axis *x_vector_axis* of ***x*** to the corresponding destination vector lying along axis *y_vector_axis* of ***y***, and replicates this destination vector along axis *y_broadcast_axis* of ***y***, placing one copy at each coordinate of axis *y_broadcast_axis*. This operation can be described by the following formula:

$$\boldsymbol{y}(,,i,,:,,) = \boldsymbol{x}(,,,i,,,)$$

NOTES

Performance. In the following discussion, "broadcast axis" refers to axis *y_broadcast_axis* of the destination array ***y***; "copy axis" refers to axis *y_vector_axis* of ***y***; and "vector axis" refers to axis *x_vector_axis* of the source array ***x***.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Performance of the **all_to_all broadcast** routine depends on the order in which the broadcast, copy, vector, and instance axes appear in the array declarations of the calling program, as well as the extents and layouts of these axes. For medium to large arrays, this routine should consistently out-perform the corresponding CM Fortran intrinsics if you follow the guidelines given below.

The performance of the all-to-all broadcast routine will be best if you declare and lay out your arrays using one or more of the following guidelines (given in order of decreasing importance):

1. Maximize the local component of the broadcast axis by giving it a high communication weight or by giving a high value to the **:block** axis descriptor of the **CMF$LAYOUT** directive. This allows performance to rise toward the peak local performance limits for the given data type.

2. Minimize the non-local component of the broadcast axis by giving it a high communication weight or by giving a low value to the **:procs** axis descriptor of **CMF$LAYOUT**. Unless you do this, performance will fall toward the physical communication bandwidth rates for the given data type. (Even if performance does approach these lower rates, however, the routine will generally out-perform equivalent CM Fortran spread intrinsics for the same array shapes and layouts.)

3. (Applies primarily to the vector-unit execution model) Declare the dimensions of the array ***y*** so that the broadcast axis is the rightmost (with the CM Fortran **-axisreorder** switch) or leftmost (with **-noaxisreorder**) non-local axis. CM Fortran will lay out such an array so as to provide a favorable data motion path and smaller memory strides. This will optimize the physical communication bandwidth, which is the lower limit of overall performance.

4. Declare the dimensions of the array ***y*** and lay out its axes using the **:block** and **:procs** descriptors so that all of the following are true:

   - The broadcast axis is to the right (with **-axisreorder**) or left (with **-noaxisreorder**) of the copy axis.

   - The copy and broadcast axes are adjacent to each other in the dimension declaration, or failing that, no non-local axis lies between them.

   - The copy and broadcast axes fall in the same position relative to any non-local instance axes as does the vector axis of ***x***.

Copyright © 1994 Thinking Machines Corporation

- The number of **:procs** in the copy and broadcast axes taken together (the product of their **:procs** numbers multiplied together) is equal to the number of **:procs** in the vector axis of ***x***.
- Corresponding instance axes, if any, in the source and destination arrays have the same number of **:procs**.
- The product of the broadcast axis's **:procs** number and the vector axis's **:block** size is equal to the **:block** size of the copy axis.

Meeting these conditions should avoid an implicit send of the source array ***x***.

Note that for given source and destination shapes and layouts, there is a moderate one-time geometry processing overhead the first time **all_to_all_broadcast** is called in a program. For this reason, typical times are better measured after the first call has been completed.

EXAMPLES

Sample CM Fortran code that uses the all-to-all broadcast routine can be found on-line in the subdirectory

```
all-to-all/cmf
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.4 All-to-All Reduction

The **all_to_all_reduce** routine performs operations similar to those of the CM Fortran intrinsics listed below, but offers better performance in many cases.

SUM
PRODUCT
MINVAL
MAXVAL
ANY
ALL
COUNT

In addition, **all_to_all_reduce** offers an xor operation that has no corresponding CM Fortran intrinsic.

Given source and destination CM arrays, with rank(source array) = rank(destination array) + 1, **all_to_all_reduce** combines sets of vectors within the source array and places each result in a corresponding vector of the destination array. The source vectors lie along axis *x_vector_axis* and are counted by axis *x_reduce_axis* (the "reduction axis") in the source array. The destination vectors lie along axis *y_vector_axis* of the destination array. You may interleave the active axes with the instance axes differently in the source and destination arrays. The reduction axis may have any length.

The man page that follows provides syntax information and describes how to obtain the best performance.

Copyright © 1994 Thinking Machines Corporation

All-to-All Reduction

Given source and destination CM arrays with rank(source array) = rank(destination array) + 1, the **all_to_all_reduce** routine combines sets of vectors within the source array and places each result in a corresponding vector of the destination array.

SYNTAX

all_to_all_reduce (***y, x,*** *y_vector_axis, x_vector_axis, x_reduce_axis, reduce_op, ier*)

ARGUMENTS

y CM array with rank one less than that of ***x***. Axis *y_vector_axis* of ***y*** must have the same extent as axis *x_vector_axis* of ***x***. All other axes of ***y*** are instance axes, and must match the instance axes of ***x*** in extent and order of declaration. Thus, each destination vector lying along axis *y_vector_axis* of ***y*** corresponds to a set of associated source vectors in ***x***; these vectors lie along axis *x_vector_axis* of ***x***, and are counted by axis *x_reduce_axis* of ***x***. Upon successful completion, each associated set of source vectors is combined, and the result is placed in the corresponding destination vector in ***y***.

The arrays ***y*** and ***x*** must have the same data type unless *reduce_op* = **CMSSL_combiner_poll**. The required data types are listed under *reduce_op*, below.

x CM array of rank greater than or equal to 2, whose data type depends on the value of *reduce_op* (see the description of *reduce_op*, below). Contains one or more associated sets of source vectors. Each set of source vectors lies along axis *x_vector_axis* and is counted by axis *x_reduce_axis*. All other axes are instance axes, and must match the instance axes of ***y*** in extent and order of declaration. The **all_to_all_reduce** routine combines each set of associated source vectors in ***x*** as specified by *reduce_op*, and places the result in the corresponding vector of ***y***.

y_vector_axis Scalar integer variable between 1 and the rank of ***y***, inclusive. Indicates the axis of ***y*** along which the destination vectors lie.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

x_vector_axis Scalar integer variable between 1 and the rank of ***x***, inclusive. Indicates the axis of ***x*** along which each set of associated source vectors lies. Must be different from *x_reduce_axis*.

x_reduce_axis Scalar integer variable between 1 and the rank of ***x***, inclusive. Indicates the axis of ***x*** that counts the vectors in each set of source vectors. May have any length.

reduce_op Scalar integer variable that specifies the arithmetic or logical combining operation to be applied to the source vectors. Can be any of the symbolic constants listed below. Each value is listed with the corresponding CM Fortran intrinsic (except for **CMSSL_combiner_xor**, which has no corresponding intrinsic) and data type requirements for ***x*** and ***y***. In this list, we follow the CMSSL documentation convention that "real" refers to either single- or double-precision data.

Symbolic constant	Intrinsic	Data Types
CMSSL_combiner_sum	**SUM**	any numeric
CMSSL_combiner_product	**PRODUCT**	any numeric
CMSSL_combiner_min	**MINVAL**	integer or real
CMSSL_combiner_max	**MAXVAL**	integer or real
CMSSL_combiner_ior	**ANY**	logical*4
CMSSL_combiner_xor	xor	logical*4
CMSSL_combiner_and	**ALL**	logical*4
CMSSL_combiner_poll	**COUNT**	***x***: logical*4; ***y***: integer

For descriptions of these CM Fortran intrinsics, see the CM Fortran documentation set.

ier Scalar integer variable. Error code. Set to 0 on successful return, or to one of the following values:

1 Invalid array descriptor. The routine was expecting a CM array but was passed either a scalar or a CM array that has not yet been allocated (for example, an array that has neither been used in a CM Fortran array expression nor laid out with a **CMF$LAYOUT** or **CMF$ALIGN** directive).

2 Data types of ***x*** and ***y*** are not the same (for combiners other than **CMSSL_combiner_poll**) or are inappropriate for the combiner.

Copyright © 1994 Thinking Machines Corporation

4	Invalid combiner.
8	Invalid array ranks. The rank of ***y*** should be one less than the rank of ***x***.
16	Invalid axis numbers. All axis numbers must be between 1 and the rank of corresponding array, inclusive.
32	Corresponding instance axes of ***x*** and ***y*** do not have the same extent.
64	Axes *x_vector_axis* and *y_vector_axis* do not have the same extent.

DESCRIPTION

The **all_to_all_reduce** routine combines each set of associated source vectors (which lie along axis *x_vector_axis* and are counted by axis *x_reduce_axis* of ***x***) as specified by the value of *reduce_op*, and places the result in the corresponding destination vector (lying along axis *y_vector_axis*) in ***y***. The reduction operations can be described by the following formulas:

$$
\begin{aligned}
\boldsymbol{y}(,,,i,,,,) &= \boldsymbol{x}(,,,i,,:|+,,)\\
\boldsymbol{y}(,,,i,,,,) &= \boldsymbol{x}(,,,i,,:|*,,)\\
\boldsymbol{y}(,,,i,,,,) &= \boldsymbol{x}(,,,i,,:|\mathrm{min},,)\\
\boldsymbol{y}(,,,i,,,,) &= \boldsymbol{x}(,,,i,,:|\mathrm{max},,)\\
\boldsymbol{y}(,,,i,,,,) &= \boldsymbol{x}(,,,i,,:|.\mathrm{ior}.,,)\\
\boldsymbol{y}(,,,i,,,,) &= \boldsymbol{x}(,,,i,,:|.\mathrm{xor}.,,)\\
\boldsymbol{y}(,,,i,,,,) &= \boldsymbol{x}(,,,i,,:|\mathrm{all},,)\\
\boldsymbol{y}(,,,i,,,,) &= \boldsymbol{x}(,,,i,,:|\mathrm{count},,)
\end{aligned}
$$

NOTES

Include the CMSSL Header File. The **all_to_all_reduce** routine uses predefined symbolic constants. Therefore, you must include the statement

```
INCLUDE '/usr/include/ cm/cmssl-cmf.h'
```

at the top of the main file of any program that uses this routine. This file defines symbolic constants and declares the type of the CMSSL functions.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Performance. In the following discussion, "reduction axis" refers to axis *x_reduce_axis* of the source array ***x***; "copy axis" refers to axis *x_vector_axis* of ***x***; and "vector axis" refers to axis *y_vector_axis* of the destination array ***y***.

Performance of the **all_to_all_reduce** routine depends on the order in which the reduction, copy, vector, and instance axes appear in the array declarations of the calling program, as well as the extents and layouts of these axes. For medium to large arrays, this routine should consistently out-perform the corresponding CM Fortran intrinsics if you follow the guidelines given below.

The performance of the all-to-all reduction routine will be best if you declare and lay out your arrays using one or more of the following guidelines (given in order of decreasing importance):

1. Maximize the local component of the reduction axis by giving it a high communication weight or by giving a high value to the **:block** axis descriptor of the **CMF$LAYOUT** directive. This allows performance to rise toward the peak local performance limits for the given data type.

2. Minimize the non-local component of the reduction axis by giving it a high communication weight or by giving a low value to the **:procs** axis descriptor of **CMF$LAYOUT**. Unless you do this, performance will fall toward the physical communication bandwidth rates for the given data type. (Even if performance does approach these lower rates, however, the routine will generally out-perform equivalent CM Fortran axis reduction intrinsics for the same array shapes and layouts.)

3. (Applies primarily to the vector-unit execution model) Declare the dimensions of the source array ***x*** so that the reduction axis is the rightmost (with the CM Fortran **-axisreorder** switch) or leftmost (with **-noaxisreorder**) non-local axis. CM Fortran will lay out such an array so as to provide a favorable data motion path and smaller memory strides. This will optimize the physical communication bandwidth, which is the lower limit of overall performance.

4. Declare the dimensions of the source array ***x*** and lay out its axes using the **:block** and **:procs** descriptors so that the product of the reduction axis's **:procs** number multiplied by its **:block** size is equal to its declared extent. Doing this prevents CM Fortran from padding the reduction axis's local component with unused elements, thus avoiding slower local reduction code that handles this situation.

5. Declare the dimensions of ***x*** and lay out its axes using the **:block** and **:procs** descriptors so that all of the following are true:

Copyright © 1994 Thinking Machines Corporation

- The reduction axis is to the right (with **-axisreorder**) or left (with **-noaxisreorder**) of the copy axis.
- The copy and reduction axes are adjacent to each other in the dimension declaration, or failing that, no non-local axis lies between them.
- The copy and reduction axes fall in the same position relative to any non-local instance axes as does the vector axis of ***y***.
- The number of **:procs** in the copy and reduction axes taken together (the product of their **:procs** numbers multiplied together) is equal to the number of **:procs** in the vector axis of ***y***.
- Corresponding instance axes, if any, in the source and destination arrays have the same number of **:procs**.
- The product of the reduction axis's **:procs** number and the vector axis's **:block** size is equal to the **:block** size of the copy axis.

Meeting these conditions should avoid an implicit send of the destination array ***y***.

Note that for given source and destination shapes and layouts, there is a moderate one-time geometry processing overhead the first time **all_to_all_reduce** is called in a program. For this reason, typical times are better measured after the first call has been completed.

EXAMPLES

Sample CM Fortran code that uses the all-to-all reduction routine can be found on-line in the subdirectory

```
all-to-all/cmf
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.5 Matrix Transpose

The matrix transpose routine, **gen_matrix_transpose**, transposes two axes of a multidimensional array. In the current release, this routine is designed specifically to achieve enhanced performance when one of the axes to be exchanged is local (resides within a single processing element) and the other is non-local (spans multiple processing elements). The axes to be transposed may have any length. This means that if you have a problem that is to be solved along a non-local axis of arbitrary length, you can transpose the axis to be local, and solve the problem locally.

The man page following this section describes the other requirements for achieving enhanced performance. In cases where these requirements are not met, **gen_matrix_transpose** performs the transpose with a simple send operation.

15.5.1 Example

The following example is based on the syntax and requirements presented in the man page. This example occurs on a CM-5 with 128 vector units (32 processing nodes). Axes 2 and 3 of the CM array ***A*** are exchanged, and the resulting array is returned in ***A_T***.

```
      real*8 A(64,256,512), A_T(64,512,256)
cmf$layout A(:serial,:serial,:news)
cmf$layout A_T(:serial,:serial,:news)

      call gen_matrix_transpose(A_T, A, 2, 3)
```

15.5.2 Special Performance Note for Three-Dimensional and Higher-Rank Arrays

The **gen_matrix_transpose** routine yields superior performance when a local axis is exchanged with a non-local axis that is distributed across the vector units on the same processing node. By far the best performance is obtained when the non-local axis spans only two vector units (that is, only the lowest off-chip bit is set). The next best performance results when the non-local axis spans four vector units on the same processing node (the two lowest off-chip bits are set). For the general case of an axis that spans multiple processing nodes, the transpose performance does not depend significantly

Copyright © 1994 Thinking Machines Corporation

of the number of contiguous off-chip bits (except if the axis spans only two nodes, in which case the performance is slightly better).

You can use this fact to improve the performance of transposes involving arrays of rank greater than or equal to three. With nodal CMSSL, you can also exploit this fact with two-dimensional arrays. See the man page for an example.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Matrix Transpose

The matrix transpose routine exchanges two axes of a multidimensional array.

SYNTAX

gen_matrix_transpose (***A_T***, ***A***, *axis_1*, *axis_2*)

ARGUMENTS

A_T	CM array of the same rank and data type as ***A***. Upon successful completion, ***A_T*** contains the array that results when axes *axis_1* and *axis_2* of ***A*** are exchanged. Axis *axis_1* of ***A_T*** must have the same extent as axis *axis_2* of ***A***; axis *axis_2* of ***A_T*** must have the same extent as axis *axis_1* of ***A***. All other axes of ***A_T*** must have the same extents as the corresponding axes of ***A***. In most cases, ***A*** and ***A_T*** must be distinct arrays. However, in certain cases (for example, a square array of rank 2), **gen_matrix_transpose** can work in-place.
A	CM array of any type and rank greater than or equal to 2. The array whose axes are to be exchanged.
axis_1	Scalar integer variable between 1 and the rank of ***A***, inclusive. Identifies one of the two axes of ***A*** to be exchanged.
axis_2	Scalar integer variable between 1 and the rank of ***A***, inclusive. Identifies one of the two axes of ***A*** to be exchanged.

DESCRIPTION

The **gen_matrix_transpose** routine exchanges axes *axis_1* and *axis_2* of the CM array ***A*** and places the results in ***A_T***. This operation can be expressed as follows:

$$A_T(,,i,,j,,,) = A(,,j,,i,,,)$$

Copyright © 1994 Thinking Machines Corporation

NOTES

Performance. In the current release, this routine is designed specifically to achieve enhanced performance in the vector-unit execution model with arrays that satisfy the conditions listed below. (Therefore, in the list below, "processing element" refers to a vector unit in this release.) In all other cases, the transpose is performed with a simple send operation.

- One of the axes to be exchanged (axis *axis_1* or axis *axis_2*) must be local; the other must be non-local. An axis is *local* if all array elements along the axis reside within one processing element. An axis is *non-local* if the array elements along the axis span multiple processing elements.

- The processing element layouts (machine geometries) underlying ***A*** and ***A_T*** must be identical. This requirement is met automatically if you compile your program with the **cmf -nopadding** switch and each array has only one non-local axis. In all other cases, you must ensure that this requirement is met by using the **:block**, **:procs**, and **:pdesc** axis descriptors of the **CMF$LAYOUT** directive, or by using the **CMF_ALLOCATE_DETAILED_ARRAY** utility. **CMF$LAYOUT** and **CMF_ALLOCATE_DETAILED_ARRAY** are described in the CM Fortran documentation set.

Special Performance Note for Three-Dimensional and Higher-Rank Arrays. The **gen_matrix_transpose** routine yields superior performance when a local axis is exchanged with a non-local axis that is distributed across the vector units on the same processing node. By far the best performance is obtained when the non-local axis spans only two vector units (that is, only the lowest off-chip bit is set). The next best performance results when the non-local axis spans four vector units on the same processing node (the two lowest off-chip bits are set). For the general case of an axis that spans multiple processing nodes, the transpose performance does not depend significantly of the number of contiguous off-chip bits (except if the axis spans only two nodes, in which case the performance is slightly better).

You can use this fact to improve the performance of transposes involving arrays of rank greater than or equal to three. With nodal CMSSL, you can also exploit this fact with two-dimensional arrays. For example, consider a problem that is being solved along each axis of a three-dimensional array ***A*** of size 128 x 128 x 128 on a

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

machine with 64 processing nodes (256 vector units). To optimize performance, choose the following array layout:

```
CMF$LAYOUT   A                                          &
CMF$             (:block=128     :pdesc=0               &
CMF$             ,:block=64      :pdesc=1               &
CMF$             ,:block=1       :pdesc=254)
```

With this layout, axis 1 is local to a vector unit. Axis 2 spans two vector units; the subgrid length along axis 2 is 64. Axis 3 spans 128 vector units; the subgrid length along axis 3 is 1. Thus, ***A*** is distributed among all 256 vector units. You can now solve the problem by following these steps:

1. Solve the problem along axis 1, which is local.
2. Exchange axes 1 and 2 by calling **gen_matrix_transpose**. (Recall that the transposed array, ***A_T***, must have the same processing element layout as ***A***.) Since axis 2 is entirely on-node, this transposition is performed very efficiently.
3. Solve the problem along axis 2, which is now local.
4. Exchange axes 2 and 3 so that axis 3 becomes local.
5. Solve the problem along axis 3.

EXAMPLES

Sample CM Fortran code that uses the matrix transpose routine can be found on-line in the subdirectory

matrix-tranpose/cmf/

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.6 Sparse Gather Utility

The sparse gather utility is a communication primitive that is used internally by the CMSSL basic linear algebra routines for arbitrary sparse matrices. The gather utility is intended for applications that do not do explicit sparse linear algebra operations, but want to make use of some of the primitives commonly used in these operations.

15.6.1 The Gather Utility Routines

Given a source vector, a destination array, and an array containing a gathering pattern, the sparse gather utility routines gather elements from the vector into the array. To perform a gather operation (or multiple gather operations, sequentially), you must follow these steps:

1. Call the **sparse_util_gather_setup** routine.
2. Call the **sparse_util_gather** routine.

   To perform more than one gather operation using the same sparsity (gathering pattern), follow one call to **sparse_util_gather_setup** with multiple calls to **sparse_util_gather**. If the sparsity changes, start with Step 1 again.
3. After all **sparse_util_gather** calls associated with the same **sparse_util_gather_setup** call, call the **deallocate_gather_setup** routine to deallocate the processing element storage space required by the setup routine.

You may have more than one setup active at a time; that is, you may call the setup routine more than once without calling the deallocation routine. However, calling the setup routine repeatedly without calling the deallocation routine may cause you to run out of memory. It is therefore strongly recommended that you call **deallocate_gather_setup** as soon as you have finished the associated gather operations.

15.6.2 Definition of the Gather Operation

The gather operation is defined by

where (y_mask) $y = x(p)$

Copyright © 1994 Thinking Machines Corporation

where ***x*** is the vector from which elements are being gathered, ***y*** is the resulting array, ***p*** is an array of pointers supplied by the user application, and ***y_mask*** is a mask for the destination array. These arguments are described in detail in the man page.

The preprocessing performed by the **sparse_util_gather_setup** routine allows a program to amortize the overhead of the setup phase over multiple communication operations, as long as the sparsity of the system remains constant. The on-line sample code performs both preprocessed and unpreprocessed gather operations.

15.6.3 Gather Operation Examples

The examples below are based on the argument definitions supplied in the man page. For clarity, these examples use letters instead of numbers for the elements of ***x*** and ***y***. The examples assume that the application has already performed any required permutations of the source vector or the ***p*** array.

Example 1: Elementwise Gather Operation

Given the source vector

x = [d g k b i j h f a e c]

the destination mask

y_mask = [T T T T T T T T T F F F F]

and the pointers array

p = [1 5 7 1 3 2 2 4 1 – – – –]

(where the symbol **–** indicates masked data), the gather operation results in the destination array

y = [d i h d k g g b d – – – –].

Copyright © 1994 Thinking Machines Corporation

Example 2: Finite-Element Type Application

Given the source vector

$$\boldsymbol{x} = [\mathrm{a}\ \mathrm{b}\ \mathrm{c}\ \mathrm{d}\ \mathrm{e}\ \mathrm{f}\ \mathrm{g}\ \mathrm{h}\ \mathrm{i}\ \mathrm{j}]$$

the destination mask

$$\boldsymbol{y_mask} = \begin{bmatrix} \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{F} \\ \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{F} \\ \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{F} \\ \mathrm{T} & \mathrm{F} & \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{F} \end{bmatrix}$$

and the pointers array

$$\boldsymbol{p} = \begin{bmatrix} 1 & 2 & 4 & 5 & 5 & - \\ 2 & 3 & 5 & 3 & 7 & - \\ 4 & 5 & 6 & 8 & 8 & - \\ 5 & - & 7 & - & - & - \end{bmatrix}$$

the gather operation results in the destination array

$$\boldsymbol{y} = \begin{bmatrix} \mathrm{a} & \mathrm{b} & \mathrm{d} & \mathrm{e} & \mathrm{e} & - \\ \mathrm{b} & \mathrm{c} & \mathrm{e} & \mathrm{c} & \mathrm{g} & - \\ \mathrm{d} & \mathrm{e} & \mathrm{f} & \mathrm{h} & \mathrm{h} & - \\ \mathrm{e} & - & \mathrm{g} & - & - & - \end{bmatrix}.$$

Example 3: Unmasked Destination Array

Given the source vector

$$\boldsymbol{x} = [\mathrm{d}\ \mathrm{g}\ \mathrm{k}\ \mathrm{b}]$$

and the pointers array

$$\boldsymbol{p} = [1\ \ 4\ \ 3\ \ 1\ \ 3\ \ 2],$$

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

and assuming ***y_mask*** = **.true.** (the destination array is not masked), the gather operation results in the destination array

y = [d b k d k g].

Copyright © 1994 Thinking Machines Corporation

Sparse Gather Utility

Given a source vector, a destination array, and an array containing a gathering pattern, the routines described below gather elements from the vector into the array.

SYNTAX

sparse_util_gather_setup (***p, y_mask, x_template***, *setup, ier*)

sparse_util_gather (***y, x***, *setup*)

deallocate_gather_setup (*setup*)

ARGUMENTS

p Integer CM array with the same axis extents and layout directives as ***y***. *Must be one-based*. Indicates the gathering pattern. If an element of the array ***p*** contains the value *n*, then the corresponding element of the destination array receives the *n*th element of the source vector during the gather operation. The contents of ***p*** remain unchanged by **sparse_util_gather_setup**.

y_mask If you need to mask elements of ***y***, declare ***y_mask*** as a logical CM array with the same axis extents and layout directives as ***y***; set to **.true.** the elements that correspond to active elements of ***y***. The contents of ***y_mask*** remain unchanged by **sparse_util_gather_setup**.

If you do not need to mask elements of ***y***, you can conserve processing element memory by supplying the scalar logical value **.true.** for ***y_mask***.

x_template CM array of any type with the same axis extent and layout directives as ***x***. The setup routine uses only the shape and layout of this array, ignoring the contents.

y CM array of any type with arbitrary shape. Destination array to which source vector elements are gathered. The initial values of ***y*** are overwritten.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

x	CM array of rank 1 and of the same type and precision as ***y***. Source vector from which elements are gathered. The contents of ***x*** remain unchanged by **sparse_util_gather**.
setup	Scalar integer variable. Internal variable. The initial value you supply to **sparse_util_gather_setup** is ignored. You must supply **sparse_util_gather** and **deallocate_gather_setup** with the value that **sparse_util_gather_setup** assigns to *setup*.
ier	Scalar integer variable. Upon return from **sparse_util_gather_setup**, contains one of the following codes: 0 Successful return. −1 Invalid arguments (for example, mismatched sizes or shapes).

DESCRIPTION

Definition. The gather operation is defined by

where (***y_mask***) $y = x(p)$

where ***x*** is the vector from which elements are being gathered, ***y*** is the resulting array, and ***p*** is an array of pointers.

Usage. Follow these steps to perform a gather operation (or multiple gather operations, sequentially):

1. Call **sparse_util_gather_setup**.
2. Call **sparse_util_gather**.

   To perform more than one gather operation using the same sparsity (gathering pattern), follow one call to **sparse_util_gather_setup** with multiple calls to **sparse_util_gather**. If the sparsity changes, start with Step 1 again.
3. After all **sparse_util_gather** calls associated with the same **sparse_util_gather_setup** call, call **deallocate_gather_setup** to deallocate the processing element storage space required by the setup routine.

You may have more than one setup active at a time; that is, you may call the setup routine more than once without calling the deallocation routine.

Copyright © 1994 Thinking Machines Corporation

Setup Phase. The **sparse_util_gather_setup** routine analyzes the gathering pattern supplied by the application in the ***p*** argument. Using ***p*** and ***y_mask***, **sparse_util_gather_setup** computes an optimization, or *trace*, for the communication required by the gather operation; allocates the required storage space; and saves the trace for use in subsequent calls to the **sparse_util_gather** routine. The setup routine assigns an appropriate value to the internal variable *setup*; this value must be supplied in subsequent calls to **sparse_util_gather** and **deallocate_gather_setup**.

The saving of the trace saves communication time, particularly when one setup call is amortized by several gather operations.

Gather Phase. The **sparse_util_gather** routine gathers elements from ***x*** into ***y***, using the communication pattern saved by a previous call to **sparse_util_gather_setup**.

As long as the arguments supplied to **sparse_util_gather_setup** remain the same, the application can call **sparse_util_gather** multiple times following one call to **sparse_util_gather_setup**, each time supplying *setup* and a source vector, ***x***, and receiving in return a destination array, ***y***.

Deallocation Phase. The **deallocate_gather_setup** routine deallocates the extra storage space that **sparse_util_gather_setup** allocated for saving the trace. Each **sparse_util_gather_setup** call should be followed (after one or more associated calls to **sparse_util_gather**) by a **deallocate_gather_setup** call.

NOTES

Trace Deallocation. It is strongly recommended that you call **deallocate_gather_setup** as soon as the associated gather operations have finished, as the trace typically occupies a significant amount of processing element storage.

Permutation of the Source Vector. Many applications require permutation of the source vector prior to the gather operation. This permutation is the responsibility of the user application and is not performed by **sparse_util_gather_setup** or **sparse_util_gather**. You can either permute the source vector itself before supplying it to **sparse_util_gather**, or permute the ***p*** array before supplying it to **sparse_util_gather_setup**.

Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the sparse gather and scatter utilities can be found on-line in the subdirectory

```
sparse-utilities/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.7 Sparse Scatter Utility

The sparse scatter utility is a communication primitive that is used internally by the CMSSL basic linear algebra routines for arbitrary sparse matrices. The scatter utility is intended for applications that do not do explicit sparse linear algebra operations, but want to make use of some of the primitives commonly used in these operations.

15.7.1 The Scatter Utility Routines

Given a source array, a destination vector, and an array containing a scattering pattern, the sparse scatter utility routines scatter elements from the array to the vector. To perform a scatter operation (or multiple scatter operations, sequentially), you must follow these steps:

1. Call the **sparse_util_scatter_setup** routine.
2. Call the **sparse_util_scatter** routine.

   To perform more than one scatter operation using the same sparsity (scattering pattern), follow one call to **sparse_util_scatter_setup** with multiple calls to **sparse_util_scatter**. If the sparsity changes, you must start with Step 1 again.
3. After all **sparse_util_scatter** calls associated with the same **sparse_util_scatter_setup** call, call the **deallocate_scatter_setup** routine to deallocate the processing element storage space required by the other two routines.

You may have more than one setup active at a time; that is, you may call the setup routine more than once without calling the deallocation routine. However, calling the setup routine repeatedly without calling the deallocation routine may cause you to run out of processing element memory. It is therefore strongly recommended that you call **deallocate_scatter_setup** as soon as you have finished the associated scatter operations.

15.7.2 Definition of the Scatter Operation

The scatter operation is defined by

where (***x_mask***) $\quad y(p \mid +) = x$

Copyright © 1994 Thinking Machines Corporation

where ***x*** is the array from which elements are being scattered, ***y*** is the resulting vector, ***p*** is an array of pointers supplied by the user application, and ***x_mask*** is a mask for the source array. These arguments are described in detail in the man page.

The preprocessing performed by the **sparse_util_scatter_setup** routine allows a program to amortize the overhead of the setup phase over multiple communication operations, as long as the sparsity of the system remains constant. The on-line sample code performs both preprocessed and unpreprocessed scatter operations.

15.7.3 Scatter Operation Example

The example below is based on the argument definitions in the man page. For clarity, this example uses letters instead of numbers for the elements of ***x*** and ***y***.

Given the source array

$$\boldsymbol{x} = \begin{bmatrix} a & e & i & m & q \\ b & f & j & n & r \\ c & g & k & o & s \\ d & h & l & p & t \end{bmatrix}$$

with mask

$$\boldsymbol{x_mask} = \begin{bmatrix} T & T & T & T & T \\ T & T & T & T & T \\ T & T & T & T & T \\ T & F & T & F & F \end{bmatrix}$$

and ***p*** array

Copyright © 1994 Thinking Machines Corporation

$$\boldsymbol{p} = \begin{bmatrix} 1 & 2 & 4 & 5 & 5 \\ 2 & 3 & 5 & 3 & 7 \\ 4 & 5 & 6 & 8 & 8 \\ 5 & - & 7 & - & - \end{bmatrix}$$

and given the initial destination vector

$\boldsymbol{y} = [v_1\ v_2\ v_3\ v_4\ v_5\ v_6\ v_7\ v_8]$

the scatter operation results in destination vector

$\boldsymbol{y} = [v_1+\mathrm{a}\ v_2+\mathrm{b}+\mathrm{e}\ v_3+\mathrm{f}+\mathrm{n}\ v_4+\mathrm{c}+\mathrm{i}\ v_5+\mathrm{d}+\mathrm{g}+\mathrm{j}+\mathrm{m}+\mathrm{q}\ v_6+\mathrm{k}\ v_7+\mathrm{l}+\mathrm{r}\ v_8+\mathrm{o}+\mathrm{s}]$.

Copyright © 1994 Thinking Machines Corporation

Sparse Scatter Utility

Given a source array, a destination vector, and an array containing a scattering pattern, the routines described below scatter elements from the array to the vector.

SYNTAX

sparse_util_scatter_setup (***p, x_mask, y_template****, setup, ier*)

sparse_util_scatter (***y, x, x_mask****, setup*)

deallocate_scatter_setup (*setup*)

ARGUMENTS

p Integer CM array with the same axis extents and layout directives as ***x***. *Must be one-based*. Indicates the scattering pattern. If an element of the array ***p*** (as supplied to **sparse_util_scatter_setup**) contains the value *n*, then the scatter operation adds the corresponding source array element to destination vector element *n*. *If two or more source array elements are sent to the same destination vector element, the colliding destination values are added*. The contents of ***p*** are modified by a successful call to **sparse_util_scatter_setup**. When you call **sparse_util_scatter**, supply the values assigned to ***p*** by **sparse_util_scatter_setup**.

x_mask If you need to mask elements of ***x***, declare ***x_mask*** as a logical CM array with the same axis extents and layout directives as ***x***; set to **.true.** the elements that correspond to active elements of ***x***. Only those source array elements for which the mask is true are sent to the destination vector. Elements of ***p*** corresponding to masked elements of the source array are ignored. The contents of ***x_mask*** remain unchanged by **sparse_util_scatter_setup**.

You may set any component of ***x_mask*** to **.false.** during the course of the computation without calling the setup routine again. However, a component of ***x_mask*** that has been set to **.false.** before the **sparse_util_scatter_setup** call cannot be set to **.true.** during the computation.

Copyright © 1994 Thinking Machines Corporation

If you do not need to mask elements of ***x***, you can conserve processing element memory by supplying the scalar logical value **.true.** for ***x_mask***.

y_template CM array of any type with the same axis extent and layout directives as ***y***. The setup routine uses only the shape and layout of this array, ignoring the contents.

y CM array of any type and rank 1. The **sparse_util_scatter** routine adds the scattered ***x*** elements to the initial values of ***y***.

x CM array of arbitrary shape and of the same type and precision as ***y***. Source array from which elements are scattered. The contents of ***x*** remain unchanged by **sparse_util_scatter**.

setup Scalar integer variable. Internal variable. The initial value supplied to **sparse_util_scatter_setup** is ignored. You must supply **sparse_util_scatter** and **deallocate_scatter_setup** with the value assigned to *setup* by **sparse_util_scatter_setup**.

ier Scalar integer variable. Upon return from **sparse_util_scatter_setup**, contains one of the following codes:

0	Successful return.
−1	Invalid arguments (for example, mismatched sizes or shapes).

DESCRIPTION

Definition. The scatter operation is defined by

where (***x_mask***) $y(p \mid +) = x$

where ***x*** is the array from which elements are being scattered, ***y*** is the resulting vector, and ***p*** is an array of pointers.

Usage. Follow these steps to perform a scatter operation (or multiple scatter operations, sequentially):

1. Call **sparse_util_scatter_setup**.
2. Call **sparse_util_scatter**.

Copyright © 1994 Thinking Machines Corporation

To perform more than one scatter operation using the same sparsity (scattering pattern), follow one call to **sparse_util_scatter_setup** with multiple calls to **sparse_util_scatter**. If the sparsity changes, start with Step 1 again.

3. After all **sparse_util_scatter** calls associated with the same **sparse_util_scatter_setup** call, call the **deallocate_scatter_setup** routine to deallocate the processing element storage space required by the other two routines.

You may have more than one setup active at a time; that is, you may call the setup routine more than once without calling the deallocation routine. However, calling the setup routine repeatedly without calling the deallocation routine may cause you to run out of memory. It is therefore strongly recommended that you call **deallocate_scatter_setup** as soon as you have finished the associated scatter operations.

Setup Phase. The **sparse_util_scatter_setup** routine analyzes the scattering pattern supplied by the application in the ***p*** argument. Using ***p*** and ***x_mask***, **sparse_util_scatter_setup** assigns an appropriate value to the internal variable *setup*; this value must be supplied in subsequent calls to **sparse_util_scatter**.

Scatter Phase. The **sparse_util_scatter** routine scatters elements from ***x*** and adds them to the initial values of ***y***, using the information returned by a previous call to the **sparse_util_scatter_setup** routine.

As long as the arguments supplied to **sparse_util_scatter_setup** remain the same, the application can call **sparse_util_scatter** multiple times following one **sparse_util_scatter_setup** call, each time supplying a source array, ***x***, and receiving in return a destination vector, ***y***.

Deallocation Phase. The **deallocate_scatter_setup** routine deallocates the extra storage space that the setup routine allocated. Each call to the setup routine should be followed (after one or more calls to **sparse_util_scatter**) by a **deallocate_scatter_setup** call.

Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the sparse gather and scatter utilities can be found on-line in the subdirectory

```
sparse-utilities/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

15.8 Sparse Vector Gather Utility

The sparse vector gather utility, **sparse_util_vec_gather**, performs the same operation as the sparse gather utility, **sparse_util_gather**, except that **sparse_util_vec_gather** operates on vectors rather than individual data elements. The vectors that are gathered may lie along axes of your choice in the source and destination arrays; these axes must be local (that is, either **:serial** or laid out with **:procs**=1).

15.8.1 Definition of the Vector Gather Operation

The vector gather operation is defined by

where (y_mask) $y = x(:, p)$

where x is the array from which vectors are being gathered, y is the resulting destination array, p is an array of pointers supplied by the user application, and y_mask is a mask for the destination array. These arguments are described in detail in the man page.

As with the sparse gather utility, the preprocessing performed by the **sparse_util_vec_gather_setup** routine allows a program to amortize the overhead of the setup phase over multiple communication operations, as long as the sparsity of the system remains constant.

15.8.2 Examples

The examples below are based on the argument definitions supplied in the man page at the end of this section. For clarity, these examples use letters instead of numbers for the elements of x and y. The examples assume that the application has already performed any required permutations of the source array or the pointers array.

Copyright © 1994 Thinking Machines Corporation

Example 1: Vector Gather Operation with Masked Destination Array

Suppose the source array ***x*** has dimensions (3, 8), and the vectors (of length 3) lie along the first axis, which is local:

$$\boldsymbol{x} = \begin{bmatrix} a & b & c & d & e & f & g & h \\ i & j & k & l & m & n & o & p \\ q & r & s & t & u & v & w & x \end{bmatrix}$$

If the destination mask is

$$\boldsymbol{y_mask} = [T\ \ T\ \ T\ \ T\ \ T\ \ T\ \ T\ \ T\ \ T\ \ F\ \ F\ \ F\ \ F]$$

and the pointers array is

$$\boldsymbol{p} = [1\ \ 5\ \ 7\ \ 1\ \ 3\ \ 2\ \ 2\ \ 4\ \ 1\ \ -\ \ -\ \ -\ \ -]$$

(where the symbol **–** indicates masked data), the vector gather operation results in the destination array

$$\boldsymbol{y} = \begin{bmatrix} a & e & g & a & c & b & b & d & a & - & - & - & - \\ i & m & o & i & k & j & j & l & i & - & - & - & - \\ q & u & w & q & s & r & r & t & q & - & - & - & - \end{bmatrix}$$

Example 2: Vector Gather Operation with Unmasked Destination Array

Given the source array

$$\boldsymbol{x} = \begin{bmatrix} a & b & c & d & e & f & g & h \\ i & j & k & l & m & n & o & p \\ q & r & s & t & u & v & w & x \end{bmatrix}$$

(again with vectors of length 3 lying along the first axis), and the pointers array

$$\boldsymbol{p} = [1\ \ 5\ \ 7\ \ 1\ \ 3\ \ 2\ \ 2\ \ 4\ \ 1],$$

and assuming ***y_mask*** = **.true.** (the destination array is not masked), the vector gather operation results in the destination array

Copyright © 1994 Thinking Machines Corporation

$$y = \begin{bmatrix} a & e & g & a & c & b & b & d & a \\ i & m & o & i & k & j & j & l & i \\ q & u & w & q & s & r & r & t & q \end{bmatrix}.$$

Copyright © 1994 Thinking Machines Corporation

Sparse Vector Gather Utility

Given a source array, a destination array, and an array containing a gathering pattern, the routines described below gather vectors from the source array into the destination array.

SYNTAX

sparse_util_vec_gather_setup (***p, y_mask, x_template****, setup, ier*)

sparse_util_vec_gather (***y, x****, y_vector_axis, x_vector_axis, setup*)

deallocate_vec_gather_setup (*setup*)

ARGUMENTS

p Integer CM array with the same axis extents and layout directives as the subarray of ***y*** formed by omitting axis *y_vector_axis* of ***y***. *Must be one-based*. Indicates the gathering pattern. If element (p_1, ..., p_k) of the array ***p*** contains the value n, then the vector ***x***(:, n) or ***x***(n, :) is gathered from the source array to locations ***y***(p_1, ..., : , ..., p_k) in the destination array during the gather operation (where : denotes axis *y_vector_axis* of ***y*** and axis *x_vector_axis* of ***x***). The contents of ***p*** remain unchanged by **sparse_util_vec_gather_setup**.

y_mask If you need to mask elements of ***y***, declare ***y_mask*** as a logical CM array with the same axis extents and layout directives as the subarray of ***y*** formed by omitting axis *y_vector_axis* of ***y***; set to **.true.** the elements that correspond to active vectors within ***y***. The contents of ***y_mask*** remain unchanged by **sparse_util_vec_gather_setup**.

If you do not need to mask elements of ***y***, you can conserve processing element memory by supplying the scalar logical value **.true.** for ***y_mask***.

x_template CM array of any type with the same axis extent and layout directives as the subarray of ***x*** formed by omitting axis *x_vector_axis* of ***x***. The setup routine uses only the shape and layout of this array, ignoring the contents.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

y CM array of any type with arbitrary shape. Destination array to which vectors from the source array are gathered. The gathered vectors will lie along axis *y_vector_axis* of ***y***. The initial values of ***y*** are overwritten.

x CM array of rank 2 and of the same type and precision as ***y***. Source array from which vectors are gathered. The vectors to be gathered must lie along axis *x_vector_axis* of ***x***. The contents of ***x*** remain unchanged by **sparse_util_vec_gather**.

y_vector_axis Scalar integer variable. Specifies the axis of ***y*** along which the gathered vectors will lie. This axis must be local (that is, either **:serial** or laid out with **:procs** = 1), and must have the same extent as axis *x_vector_axis* of ***x***.

x_vector_axis Scalar integer variable. Specifies the axis of ***x*** along which the vectors to be gathered must lie. This axis must be local (that is, either **:serial** or laid out with **:procs** = 1), and must have the same extent as axis *y_vector_axis* of ***y***.

setup Scalar integer variable. Internal variable. The initial value you supply to **sparse_util_vec_gather_setup** is ignored. You must supply **sparse_util_vec_gather** and **deallocate_vec_gather_setup** with the value that **sparse_util_vec_gather_setup** assigns to *setup*.

If you need to call both **sparse_util_gather** and **sparse_util_vec_gather**, you can save memory by supplying the *setup* generated by **sparse_util_vec_gather_setup** to **sparse_util_gather**.

ier Scalar integer variable. Upon return from **sparse_util_vec_gather_setup**, contains one of the following codes:

0	Successful return.
−1	Invalid arguments (for example, mismatched sizes or shapes).

DESCRIPTION

Definition. The vector gather operation is defined by

where (***y_mask***) $\quad y = x(:, p)$

Copyright © 1994 Thinking Machines Corporation

where ***x*** is the array from which vectors are being gathered, ***y*** is the resulting destination array, and ***p*** is an array of pointers.

Usage. Follow these steps to perform a vector gather operation (or multiple vector gather operations, sequentially):

1. Call **sparse_util_vec_gather_setup**.
2. Call **sparse_util_vec_gather**.

   To perform more than one vector gather operation using the same sparsity (gathering pattern), follow one call to **sparse_util_vec_gather_setup** with multiple calls to **sparse_util_vec_gather**. If the sparsity changes, start with Step 1 again.
3. After all **sparse_util_vec_gather** calls associated with the same **sparse_util_vec_gather_setup** call, call **deallocate_vec_gather_setup** to deallocate the processing element storage space required by the setup routine.

You may have more than one setup active at a time; that is, you may call the setup routine more than once without calling the deallocation routine.

Setup Phase. The **sparse_util_vec_gather_setup** routine analyzes the gathering pattern supplied by the application in the ***p*** argument. Using ***p*** and ***y_mask***, **sparse_util_vec_gather_setup** computes an optimization, or *trace*, for the communication required by the gather operation; allocates the required storage space; and saves the trace for use in subsequent calls to the **sparse_util_vec_gather** routine. The setup routine assigns an appropriate value to the internal variable *setup*; this value must be supplied in subsequent calls to **sparse_util_vec_gather** and **deallocate_vec_gather_setup**.

The saving of the trace saves communication time, particularly when one setup call is amortized by several gather operations.

Gather Phase. The **sparse_util_vec_gather** routine gathers vectors from ***x*** into ***y***, using the communication pattern saved by a previous call to **sparse_util_vec_gather_setup**.

As long as the arguments supplied to **sparse_util_vec_gather_setup** remain the same, the application can call **sparse_util_vec_gather** multiple times following one call to **sparse_util_vec_gather_setup**, each time supplying *setup* and a source array, ***x***, and receiving in return a destination array, ***y***.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Deallocation Phase. The **deallocate_vec_gather_setup** routine deallocates the extra storage space that **sparse_util_vec_gather_setup** allocated for saving the trace. Each **sparse_util_vec_gather_setup** call should be followed (after one or more associated calls to **sparse_util_vec_gather**) by a **deallocate_vec_gather_setup** call.

NOTES

Trace Deallocation. It is strongly recommended that you call **deallocate_vec_gather_setup** as soon as the associated gather operations have finished, as the trace typically occupies a significant amount of processing element storage.

Permutation of the Source Array. Many applications require permutation of the source array prior to the vector gather operation. This permutation is the responsibility of the user application and is not performed by **sparse_util_vec_gather_setup** or **sparse_util_vec_gather**. You can either permute the source array itself before supplying it to **sparse_util_vec_gather**, or permute the *p* array before supplying it to **sparse_util_vec_gather_setup**.

Layout and Performance. For best performance, axes *x_vector_axis* and *y_vector_axis* should be either the leftmost or the rightmost axes of their respective arrays.

EXAMPLES

Sample CM Fortran code that uses the sparse vector gather and sparse vector scatter utilities can be found on-line in the subdirectory

```
sparse-utilities/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.9 Sparse Vector Scatter Utility

The sparse vector scatter utility, **sparse_util_vec_scatter**, performs the same operation as the sparse scatter utility, **sparse_util_scatter**, except that **sparse_util_vec_scatter** operates on vectors rather than individual data elements. The vectors that are scattered may lie along axes of your choice in the source and destination arrays; these axes must be local (that is, either **:serial** or laid out with **:procs**=1).

15.9.1 Definition of the Vector Scatter Operation

The scatter operation is defined by

where (***x_mask***) $y(:, p \mid +) = x$

where ***x*** is the array from which vectors are being scattered, ***y*** is the resulting destination array, ***p*** is an array of pointers supplied by the user application, and ***x_mask*** is a mask for the source array. These arguments are described in detail in the man page at the end of this section.

The preprocessing performed by the **sparse_util_vec_scatter_setup** routine allows a program to amortize the overhead of the setup phase over multiple communication operations, as long as the sparsity of the system remains constant.

15.9.2 Example

The example below is based on the argument definitions in the man page. For clarity, this example uses letters instead of numbers for the elements of ***x***.

Suppose the source array ***x*** has dimensions (3, 6), and the vectors (of length 3) lie along the first axis, which is local:

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

$$\boldsymbol{x} = \begin{bmatrix} a & e & i & m & q \\ b & f & j & n & r \\ c & g & k & o & s \end{bmatrix}.$$

If ***x_mask*** is

x_mask = [T T T T F],

the pointers array is

p = [1 4 2 2 −],

and the initial destination array is

$$\boldsymbol{y} = \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} & v_{15} & v_{16} & v_{17} & v_{18} \\ v_{21} & v_{22} & v_{23} & v_{24} & v_{25} & v_{26} & v_{27} & v_{28} \\ v_{31} & v_{32} & v_{33} & v_{34} & v_{35} & v_{36} & v_{37} & v_{38} \end{bmatrix},$$

then the vector scatter operation results in destination array

$$\boldsymbol{y} = \begin{bmatrix} v_{11}+a & v_{12}+i+m & v_{13} & v_{14}+e & v_{15} & v_{16} & v_{17} & v_{18} \\ v_{21}+b & v_{22}+j+n & v_{23} & v_{24}+f & v_{25} & v_{26} & v_{27} & v_{28} \\ v_{31}+c & v_{32}+k+o & v_{33} & v_{34}+g & v_{35} & v_{36} & v_{37} & v_{38} \end{bmatrix}.$$

Copyright © 1994 Thinking Machines Corporation

Sparse Vector Scatter Utility

Given a source array, a destination array, and an array containing a scattering pattern, the routines described below scatter vectors from the source array to the destination array.

SYNTAX

sparse_util_vec_scatter_setup (***p, x_mask, y_template****, setup, ier*)

sparse_util_vec_scatter (***y, x, x_mask****, y_vector_axis, x_vector_axis, setup*)

deallocate_vec_scatter_setup (*setup*)

ARGUMENTS

p Integer CM array with the same axis extents and layout directives as the subarray of ***x*** formed by omitting axis *x_vector_axis* of ***x***. *Must be one-based*. Indicates the scattering pattern. If element (p_1, ..., p_k) of the array ***p*** (as supplied to **sparse_util_vec_scatter_setup**) contains the value n, then the scatter operation adds the vector ***x***(p_1, ..., :, ..., p_k) to the vector ***y***(:, n) or ***y***(n, :) within the destination array (where : denotes axis *y_vector_axis* of ***y*** and axis *x_vector_axis* of ***x***). *If two or more source array elements are sent to the same destination array element, the colliding destination values are added*. The contents of ***p*** remain unchanged by **sparse_util_vec_scatter_setup**.

x_mask If you need to mask elements of ***x***, declare ***x_mask*** as a logical CM array with the same axis extents and layout directives as the subarray of ***x*** formed by omitting axis *x_vector_axis* of ***x***; set to **.true.** the elements that correspond to active vectors within ***x***. Only those vectors for which the mask is true are sent to the destination array. Elements of ***p*** corresponding to masked locations of the source array are ignored. The contents of ***x_mask*** remain unchanged by **sparse_util_vec_scatter_setup**.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

You may set any component of ***x_mask*** to **.false.** during the course of the computation without calling the setup routine again. However, a component of ***x_mask*** that has been set to **.false.** before the **sparse_util_vec_scatter_setup** call cannot be set to **.true.** during the computation.

If you do not need to mask elements of ***x***, you can conserve processing element memory by supplying the scalar logical value **.true.** for ***x_mask***.

y_template CM array of any type with the same axis extent and layout directives as the subarray of ***y*** formed by omitting axis *y_vector_axis* of ***y***. The setup routine uses only the shape and layout of this array, ignoring the contents.

y CM array of any type and rank 2. The **sparse_util_vec_scatter** routine adds the scattered vectors from ***x*** to the initial values of the vectors that lie along axis *y_vector_axis* of ***y***.

x CM array of arbitrary shape and of the same type and precision as ***y***. Source array. The vectors to be scattered must lie along axis *x_vector_axis* of ***x***. The contents of ***x*** remain unchanged by **sparse_util_vec_scatter**.

y_vector_axis Scalar integer variable. Specifies the axis of ***y*** along which the scattered vectors will lie. This axis must be local (that is, either **:serial** or laid out with **:procs** = 1), and must have the same extent as axis *x_vector_axis* of ***x***.

x_vector_axis Scalar integer variable. Specifies the axis of ***x*** along which the vectors to be scattered must lie. This axis must be local (that is, either **:serial** or laid out with **:procs** = 1), and must have the same extent as axis *y_vector_axis* of ***y***.

setup Scalar integer variable. Internal variable. The initial value supplied to **sparse_util_vec_scatter_setup** is ignored. You must supply **sparse_util_vec_scatter** and **deallocate_vec_scatter_setup** with the value assigned to *setup* by **sparse_util_vec_scatter_setup**.

If you need to call both **sparse_util_scatter** and **sparse_util_vec_scatter**, you can save memory by supplying the *setup* generated by **sparse_util_vec_scatter_setup** to **sparse_util_scatter**.

ier Scalar integer variable. Upon return from **sparse_util_vec_scatter_setup**, contains one of the following codes:

Copyright © 1994 Thinking Machines Corporation

0	Successful return.
−1	Invalid arguments (for example, mismatched sizes or shapes).

DESCRIPTION

Definition. The scatter operation is defined by

where (***x_mask***) $y(:, p \mid +) = x$

where ***x*** is the array from which vectors are being scattered, ***y*** is the resulting destination array, and ***p*** is an array of pointers.

Usage. Follow these steps to perform a vector scatter operation (or multiple vector scatter operations, sequentially):

1. Call **sparse_util_vec_scatter_setup**.
2. Call **sparse_util_vec_scatter**.

   To perform more than one vector scatter operation using the same sparsity (scattering pattern), follow one call to **sparse_util_vec_scatter_setup** with multiple calls to **sparse_util_vec_scatter**. If the sparsity changes, start with Step 1 again.
3. After all **sparse_util_vec_scatter** calls associated with the same **sparse_util_vec_scatter_setup** call, call the **deallocate_vec_scatter_setup** routine to deallocate the processing element storage space required by the other two routines.

You may have more than one setup active at a time; that is, you may call the setup routine more than once without calling the deallocation routine. However, calling the setup routine repeatedly without calling the deallocation routine may cause you to run out of memory. It is therefore strongly recommended that you call **deallocate_vec_scatter_setup** as soon as you have finished the associated scatter operations.

Setup Phase. The **sparse_util_vec_scatter_setup** routine analyzes the scattering pattern supplied by the application in the ***p*** argument. Using ***p*** and ***x_mask***, **sparse_util_vec_scatter_setup** assigns appropriate values to the internal variable *setup*, which must be supplied in subsequent calls to **sparse_util_vec_scatter**.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Scatter Phase. The **sparse_util_vec_scatter** routine scatters vectors from ***x*** and adds them to the initial values of ***y***, using the information returned by a previous call to the **sparse_util_vec_scatter_setup** routine.

As long as the arguments supplied to **sparse_util_vec_scatter_setup** remain the same, the application can call **sparse_util_vec_scatter** multiple times following one **sparse_util_vec_scatter_setup** call, each time supplying a source array, ***x***, and receiving in return a destination array, ***y***.

Deallocation Phase. The **deallocate_vec_scatter_setup** routine deallocates the extra storage space that the setup routine allocated. Each call to the setup routine should be followed (after one or more calls to **sparse_util_vec_scatter**) by a **deallocate_vec_scatter_setup** call.

NOTES

Layout and Performance. For best performance, axes *x_vector_axis* and *y_vector_axis* should be either the leftmost or the rightmost axes of their respective arrays.

EXAMPLES

Sample CM Fortran code that uses the sparse vector gather and scatter utilities can be found on-line in the subdirectory

```
sparse-utilities/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.10 Block Gather and Scatter Utilities

The **block_gather** and **block_scatter** routines move a block of data from a source CM array into a destination CM array. The arrays must have the same rank (≥ 2), type (integer, real, or complex), precision, and layout, with at least one local axis and at least one non-local axis. The gather or scatter operation occurs along a single specified local axis. In the simplest case, a block of data elements is moved from a two-dimensional source array (with one local dimension and one non-local dimension) to a similar destination array. You can add instances by extending the non-local axis or by adding more axes (which may be local or non-local).

In **block_gather**, the source starting index for the gather operation can be different for each instance; the destination starting index is the same for all instances. In **block_scatter**, the source starting index is the same for all instances; the destination starting index can be different for each instance. In both **block_gather** and **block_scatter**, the block of data that is moved in each instance can be spread out along the local axis, with gaps between elements, in both the source and destination arrays.

You can use **block_gather** and **block_scatter** to avoid implicit indirect addressing or communication operations.

The man page at the end of this section provides detailed information about these routines and their arguments. Refer to the argument list when examining the following three figures. Figure 44 shows a block gather operation; Figure 45 shows a block scatter operation. In these figures, the source and destination arrays are two-dimensional; axis *vector_axis* is vertical. Only a single instance is pictured.

Copyright © 1994 Thinking Machines Corporation

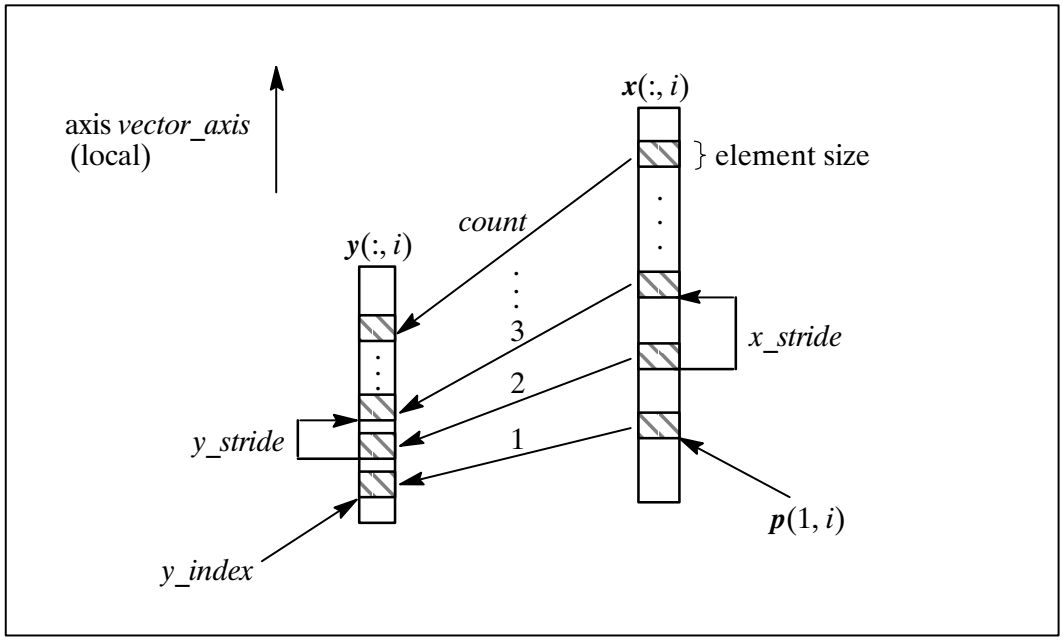

Figure 44. A block gather operation. The source and destination arrays are 2-dimensional. The vertical axis is the selected local axis. Only a single instance is shown.

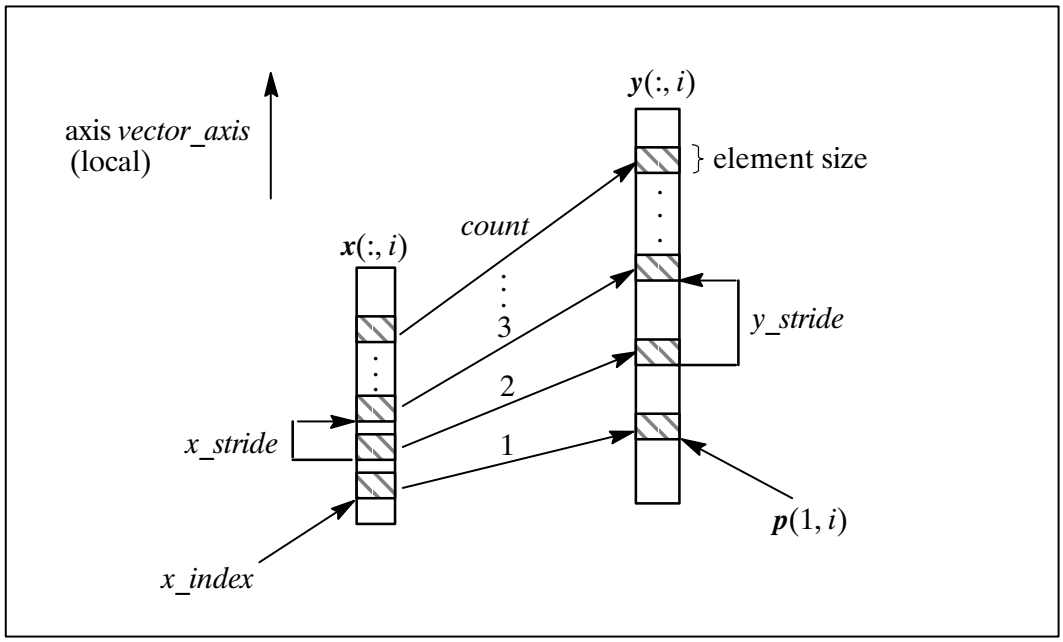

Figure 45. A block scatter operation. The source and destination arrays are 2-dimensional. The vertical axis is the selected local axis. Only a single instance is shown.

Copyright © 1994 Thinking Machines Corporation

Figure 46 shows two instances of the same block scatter operation pictured in Figure 45. Note that the destination starting indices are different for the two instances, whereas the source starting indices, strides, and count are the same for all instances.

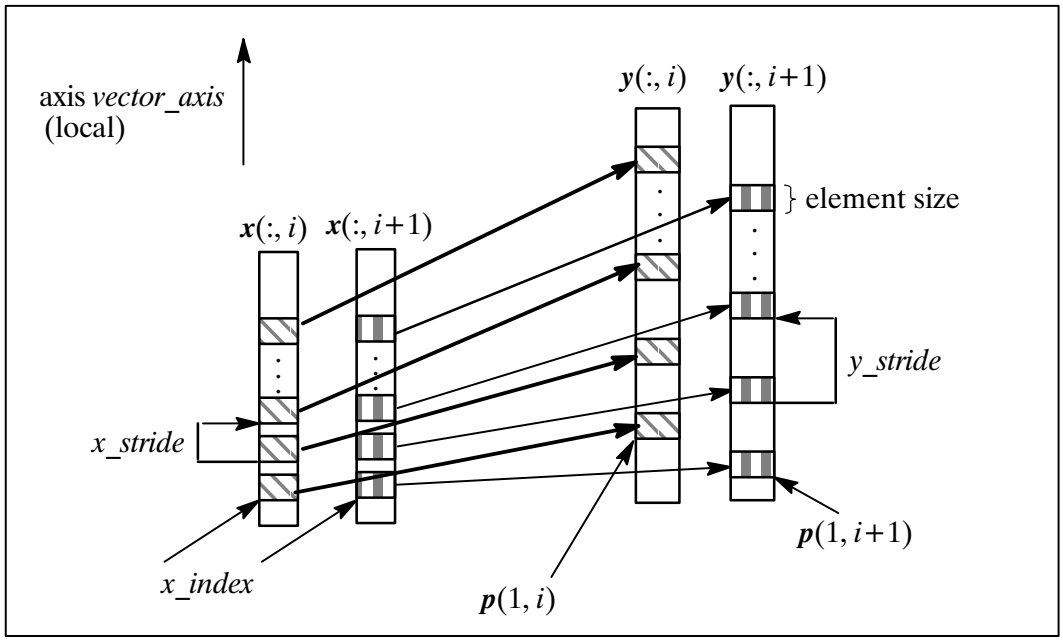

Figure 46. Two instances of the block scatter operation from Figure 45.

Copyright © 1994 Thinking Machines Corporation

Block Gather and Scatter Utilities

The **block_gather** and **block_scatter** routines move a block of data from a source CM array into a destination CM array. The arrays must have the same rank (≥ 2), type (integer, real, or complex), precision, and layout, with at least one local axis and at least one non-local axis. In the gather operation, the source starting index can be different for each instance; the destination starting index is the same for all instances. In the scatter operation, the source starting index is the same for all instances; the destination starting index can be different for each instance.

SYNTAX

block_gather (***y***, *y_stride, y_index,* ***x***, *x_stride,* ***p***, *count, vector_axis*)

block_scatter (***y***, *y_stride,* ***p***, ***x***, *x_stride, x_index, count, vector_axis*)

ARGUMENTS

y	Integer, real, or complex CM array of rank ≥ 2. Must have at least one local axis and at least one non-local axis. Serves as the destination array.
y_stride	Scalar integer. Specifies the distance (along axis *vector_axis*) between the ***y*** array locations into which data is to be moved. A gap of (*y_stride* − 1) elements will be left between incoming destination data elements. If *y_stride* = 1, destination data elements are placed in locations that are contiguous along axis *vector_axis*.
y_index	Scalar integer used only in the **block_gather** call. Specifies the ***y*** starting index (along axis *vector_axis*) for all instances of the gather operation.
x	CM array of the same rank, type, precision, and layout as ***y***. Serves as the source array. The axis identified by *vector_axis* may have different extents in ***x*** and ***y***; however, all other axes must have the same extents in ***x*** and ***y***.

Copyright © 1994 Thinking Machines Corporation

x_stride Scalar integer. Specifies the distance (along axis *vector_axis*) between the ***x*** array locations from which data is to be moved. A gap of (*x_stride* − 1) elements will be left between outgoing source data elements. If *x_stride* = 1, source data elements are moved from locations that are contiguous along axis *vector_axis*.

x_index Scalar integer used only in the **block_scatter** call. Specifies the ***x*** starting index (along axis *vector_axis*) for all instances of the scatter operation.

p Integer CM array that has the same rank and layout as ***x*** and ***y***. Axis *vector_axis* must have extent 1; all other axes must have the same extents as in ***x*** and ***y***. The meaning of the element

$$\boldsymbol{p}(u_1, ..., u_{vector_axis-1}, 1, u_{vector_axis+1}, ..., u_n)$$

depends on whether you are calling **block_gather** or **block_scatter**:

- When you call **block_gather**, it is the ***x*** starting index (along axis *vector_axis*) for the gather operation occurring at locations $(u_1, ..., u_{vector_axis-1}, :, u_{vector_axis+1}, ..., u_n)$ of ***x*** and ***y***.
- When you call **block_scatter**, it is the ***y*** starting index (along axis *vector_axis*) for the scatter operation occurring at locations $(u_1, ..., u_{vector_axis-1}, :, u_{vector_axis+1}, ..., u_n)$ of ***x*** and ***y***.

count Scalar integer. The number of data elements to be moved in each instance.

vector_axis Scalar integer. Identifies the local axis along which data is to be moved from ***x*** to ***y***.

DESCRIPTION

The **block_gather** and **block_scatter** routines move a block of data from a source CM array into a destination CM array. The arrays must have the same rank (≥ 2), type (real or complex), precision, and layout, with at least one local axis and at least one non-local axis. The gather or scatter operation occurs along a single specified local axis. In the simplest case, a block of data elements is moved from a two-dimensional source array (with one local dimension and one non-local dimension) to a

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

similar destination array. You can add instances by extending the non-local axis or by adding more axes (which may be local or non-local).

In **block_gather**, the source starting index for the gather operation can be different for each instance; the destination starting index is the same for all instances. In **block_scatter**, the source starting index is the same for all instances; the destination starting index can be different for each instance. In both **block_gather** and **block_scatter**, the block of data that is moved in each instance can be spread out along the local axis, with gaps between elements, in both the source and destination arrays.

You can use **block_gather** and **block_scatter** to avoid implicit indirect addressing or communication operations.

Given ***x*** and ***y*** arrays of rank 2, with one axis local and the other non-local with length equal to the number of processing elements, the following CM Fortran pseudo-code describes the gather operation:

```
      REAL*4 Y(32,64),X(128,64),p(1,64)
CMF$LAYOUT Y(:serial,),X(:serial,),p(:serial,)

      do i=1,count
           Y(y_index+y_stride*(i-1),:)=
     +           X(p(1,:)+x_stride*(i-1),:)
      enddo
```

The effect of this code is to copy a block of *count* elements of ***x*** (strided by *x_stride* elements and starting at a different starting index in each processing element) into ***y***, striding by *y_stride* elements and starting at a fixed starting index of *y_index*.

The scatter operation performs the inverse operation, copying a block of *count* elements of ***x*** (strided by *x_stride* elements and starting at a fixed starting index of *x_index*) into ***y***, striding by *y_stride* elements and starting at a different starting index in each processing element.

The block gather and scatter operations can also be described as follows:

Block gather: $\boldsymbol{y}(,,,i,,,) = \boldsymbol{x}(,,, p(,,,1,,,),,,)$
Block scatter: $\boldsymbol{y}(,,,p(,,,1,,,),,,) = \boldsymbol{x}(,,,i,,,)$

where

$y_index \le i \le y_index + (count-1) * y_stride$
$x_index \le i \le x_index + (count-1) * x_stride$

Copyright © 1994 Thinking Machines Corporation

If collisions occur at the destination during the scatter operation, one of the colliding values is stored; the others are lost.

NOTES

CAUTION. These routines are a high-level interface to functions that are used internally in CMSSL. These functions may be misused and can access memory beyond the normal bounds of the array. While some safety checking will occur when safety is enabled, it will not be able to catch all misuses of the function, so users of these functions should be sure they stay within the bounds of the arrays.

EXAMPLES

Sample CM Fortran code that uses the block gather and scatter utilities can be found on-line in the subdirectory

```
block-scatter-gather/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.11 Partitioning of an Unstructured Mesh and Reordering of Pointers

The routines described in this section allow you to reorder an array of pointers derived from a mesh so that the off-processing-element communication required by subsequent gather and scatter operations is reduced. The following routines are provided:

generate_dual Given an *element nodes* array, ***ien***, that describes an unstructured mesh, this routine produces the corresponding *dual connectivity* array, ***idual***.

partition_mesh Given a dual connectivity array describing an unstructured mesh, this routine returns a permutation, ***q***, that reorders the mesh elements to form discrete *partitions*. The routine also returns the number of resulting partitions and the number of elements per partition.

reorder_pointers Given a pointers array, ***p***, and a permutation ***q*** (for example, the permutation returned by **partition_mesh)**, this routine reorders the pointers array along any specified axis using ***q***.

renumber_pointers Given a pointers array, ***p***, that has been reordered by **reorder_pointers**, this routine changes the pointer values to improve data locality and returns the renumbering mapping. (This routine also works on any pointers array ***p*** that represents a mesh.)

If you derive the pointers array ***p*** from a mesh, reorder ***p*** using the permutation ***q*** returned by **partition_mesh**, and then supply these reordered pointers to the setup routine for the partitioned gather or scatter operation (described in Sections 15.12 and 15.13, respectively), the setup routine takes advantage of data locality, and the off-processing-element communication required by the gather or scatter is reduced.

If you use the **renumber_pointers** routine, which is optional but may improve performance, you must reorder your data before and after calling the partitioned gather and scatter routines. Details are provided later in this section.

Copyright © 1994 Thinking Machines Corporation

Note that in the C* interface to CMSSL, the reordering and renumbering routines are called **CMSSL_reorder_mappping** and **CMSSL_renumber_mapping**, respectively.

15.11.1 Definitions

An *unstructured mesh* is a finite collection of elements and nodes. Each element is a bounded region of n-space (where $n = 1, 2$, or 3) defined by nodes and faces. Each element and each node is assigned a number. Figure 47 shows a two-dimensional unstructured mesh with 11 elements. Node numbers are not shown in this figure.

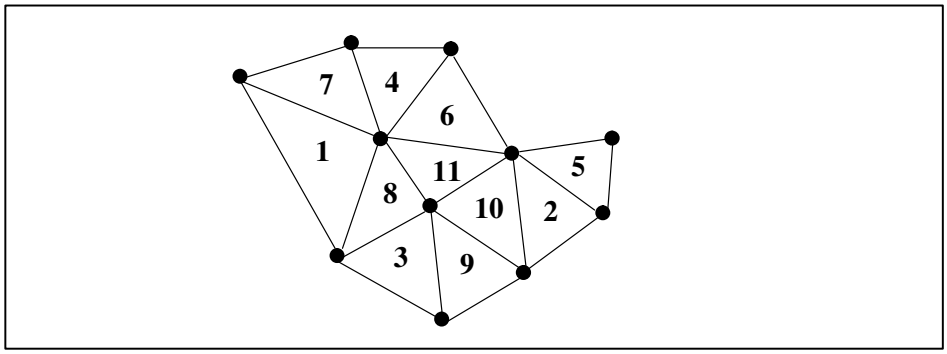

Figure 47. A two-dimensional unstructured mesh with 11 elements.

Two ways of representing an unstructured mesh are as follows:

- Each node in the mesh is associated with one or more elements of the *element nodes* array, ***ien***. This array has rank 2. One axis (called axis *node_axis* in this book) has length *nnode*, and the other has length *nel*, where *nnode* is the maximum number of nodes per element and *nel* is the number of elements in the mesh. If *node_axis* = 1, then ***ien*** is defined by

  ien(m, n) = the node number of the mth node of the nth mesh element ($1 \leq m \leq nnode$, $1 \leq n \leq nel$).

  If *node_axis* = 2, ***ien*** is defined by

  ien(n, m) = the node number of the mth node of the nth mesh element ($1 \leq m \leq nnode$, $1 \leq n \leq nel$).

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

In Figure 48, *node_axis* = 1 and ***ien***(1, 8) = ***ien***(2, 1) = 1.

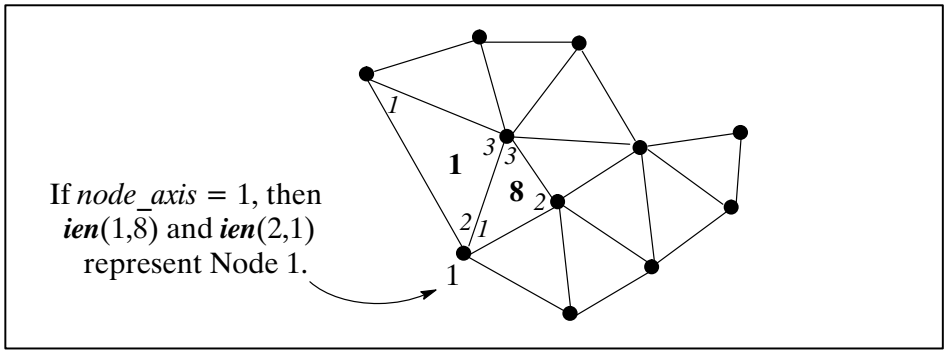

Figure 48. The *ien* array identifies nodes.

- The connectivity of the mesh elements across faces is represented by the *dual connectivity* array, ***idual***. This array also has rank 2. One axis (called axis *face_axis* in this book) has length *nface*, and the other has length *nel*, where *nface* is the maximum number of faces per element and *nel* is the number of elements in the mesh. If *face_axis* = 1, then ***idual*** is defined by

  idual(*m*, *n*) = the element that shares face *m* with element *n*;
  idual(*m*, *n*) = 0 if face *m* of element *n* is a boundary of the mesh.
  ($1 \leq m \leq nface$, $1 \leq n \leq nel$)

  If *face_axis* = 2, ***idual*** is defined by

  idual(*n, m*) = the element that shares face *m* with element *n*;
  idual(*n, m*) = 0 if face *m* of element *n* is a boundary of the mesh.
  ($1 \leq m \leq nface$, $1 \leq n \leq nel$)

  In Figure 49, *face_axis* = 1 and ***idual***(1, 1) = 7, ***idual***(2, 1) = 8, and ***idual***(3, 1) = 0.

Copyright © 1994 Thinking Machines Corporation

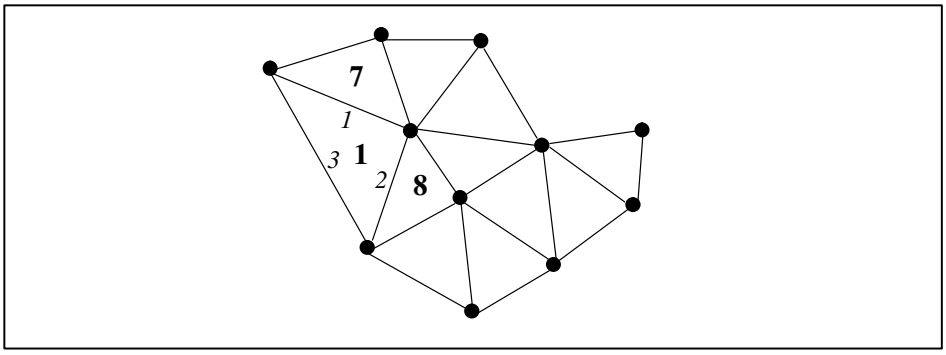

Figure 49. The *idual* array identifies neighboring elements.

NOTE

In the mesh example above, all elements are of the same type (that is, they all have the same number of nodes). In the current implementation, the **generate_dual** routine imposes this restriction. However, you can work with a mesh with different element types when you call **partition_mesh**, by setting the extra faces of the input array ***idual*** to zero. For example, to supply **partition_mesh** with a mesh with both triangles (3 faces) and quadrilaterals (4 faces), where *face_axis* = 1, set *nface* = 4 (the larger of the two), and set ***idual***(4, *n*) = 0 if element *n* is a triangle.

15.11.2 Finite Element Numbering Scheme

The **generate_dual** routine generates the ***idual*** array corresponding to the ***ien*** array and element type you supply. Your ***ien*** array must use the standard finite element numbering scheme. Figure 50 shows the standard numbering schemes for the element types supported by **generate_dual**.

Copyright © 1994 Thinking Machines Corporation

The **generate_dual** routine can handle higher-order element meshes as long as the nodes at the vertices of the elements are the first ones listed in ***ien***.

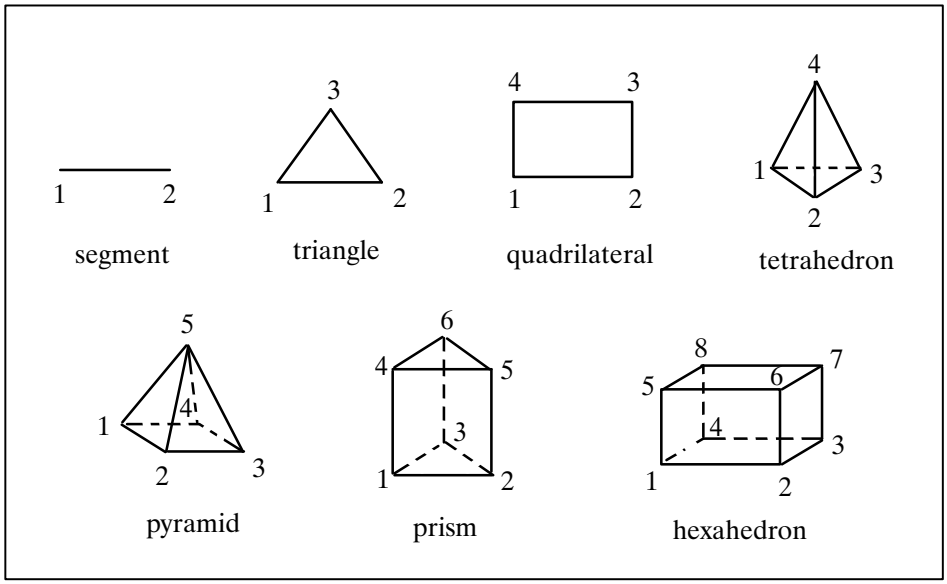

Figure 50. The standard finite element numbering schemes expected by generate_dual.

15.11.3 Partitioning Rules

You must supply the **partition_mesh** routine with an ***idual*** array that describes a mesh. The routine determines the number of partitions based on the number of mesh elements and the value of the input argument *numproc*, as described below. Note that in this discussion and throughout this book, the term *processing element* refers to the value returned by **CMF_NUMBER_OF_PROCESSORS( )**. This CM Fortran utility is described in the CM Fortran documentation set.

The elements of the mesh (represented by the ***idual*** array) are laid out on the processing elements by the compiler; they span as many processing elements as possible, subject to the multiple-of-eight rule if that rule is in effect. Partitioning is accomplished as follows:

Copyright © 1994 Thinking Machines Corporation

1. The **partition_mesh** routine divides the processing elements into equal subsets (that is, each subset contains the same number of processing elements). The number of subsets is determined by the input value of *numproc*, which must be either ≤ 0 or a positive power of 2 that is less than or equal to **CMF_NUMBER_OF_PROCESSORS( )**. These values have the following effects:

   - If *numproc* ≤ 0, there are as many subsets as processing elements; each subset consists of just one processing element.
   - If *numproc* $= n > 0$, there are n subsets. If n is strictly less than **CMF_NUMBER_OF_PROCESSORS( )**, then each subset contains multiple processing elements.

2. The routine then reorders the mesh elements and divides them among the subsets. When this process is complete, all subsets except the last hold the same number, k, of mesh elements; the last subset may hold $l \leq k$ elements.

3. Partitions are mapped to subsets. Each partition is mapped to all of the processing elements in the associated subset, except the last partition, which may be mapped to only some of the processing elements in the subset. A processing element never contains mesh elements from more than one partition.

The reordering and subdividing of the mesh elements in step 2 occurs in steps. If *numproc* $= n > 0$, there are $\log_2(n)$ steps in this process.

Upon completion, **partition_mesh** returns the number of subsets in *numproc*, the number of partitions in *numpar*, and the number of elements per partition (k, in step 2 above) in *nelpar*. (If padding is in effect, *nelpar* is actually the *largest* number of elements each partition can contain.)

One reason for using multiple-processing-element subsets (*numproc* > 0) would be to divide a very large mesh (which cannot be handled as one piece by your application) into several smaller pieces that can be processed sequentially. For example, you might split a large mesh in half using **partition_mesh** with *numproc* $= 2$; then partition each resulting piece and map it to all the processing elements using **partition_mesh** with *numproc* $= 0$; and then perform the computations on the two pieces sequentially.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Example

Suppose you are using 128 processing elements and your mesh contains 201 elements. The compiler lays out the mesh elements (***idual***) on the processing elements as follows:

- If the multiple-of-eight rule is *not* in effect (either because you included the **–nopadding** switch on the **cmf** command line, or because you are not using the vector units execution model), there are two mesh elements in each of 100 processing elements; the one remaining mesh element resides in the 101st processing element (see Figure 51).

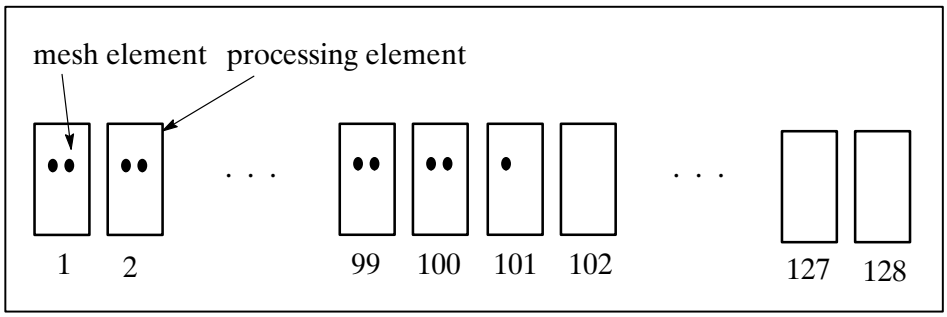

Figure 51. Mesh with 201 elements in machine configuration with 128 processing elements, with no multiple-of-eight rule.

- If the multiple-of-eight rule *is* in effect (that is, you are using the vector units execution model and padding is in effect), there are eight mesh elements in each of 25 processing elements; the one remaining mesh element resides in the 26th processing element (see Figure 52).

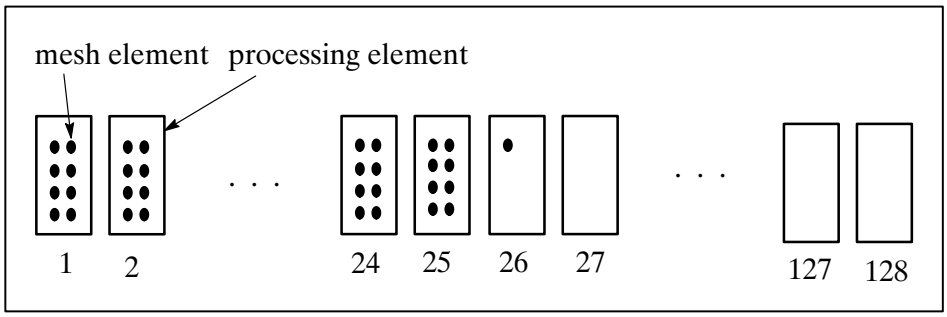

Figure 52. Mesh with 201 elements in machine configuration with 128 processing elements and multiple-of-eight rule.

Copyright © 1994 Thinking Machines Corporation

If you set *numproc* $\leq$ 0 on input, then there are 128 subsets, each consisting of a single processing element. In the non-multiple-of-eight case (Figure 51), steps 2 and 3 result in 101 partitions; **partition_mesh** returns *numproc* = 128, *numpar* = 101, and *nelpar* = 2. In the multiple-of-eight case (Figure 52), steps 2 and 3 result in 26 partitions; **partition_mesh** returns *numproc* = 128, *numpar* = 26, and *nelpar* = 8.

On the other hand, if you set *numproc* = 4, **partition_mesh** divides the processing elements into 4 subsets, each with 32 processing elements. Figure 53 shows the subsets and partitions resulting from steps 2 and 3 in the non-multiple-of-eight case. Upon completion, **partition_mesh** returns *numproc* = 4, *numpar* = 4, and *nelpar* = 64.

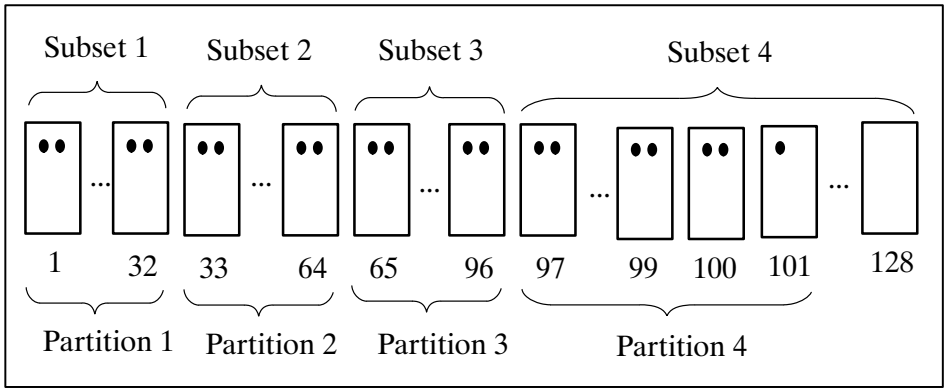

Figure 53. Subsets and partitions with no multiple-of-eight rule.

Figure 54 shows the subsets and partitions resulting from steps 2 and 3 in the multiple-of-eight case. Upon completion, **partition_mesh** returns *numproc* = 4, *numpar* = 1, and *nelpar* = 256. (In this case, since padding is in effect, *nelpar* is the largest number of elements each partition can contain, even though the mesh does not have enough elements to fill one partition.)

Copyright © 1994 Thinking Machines Corporation

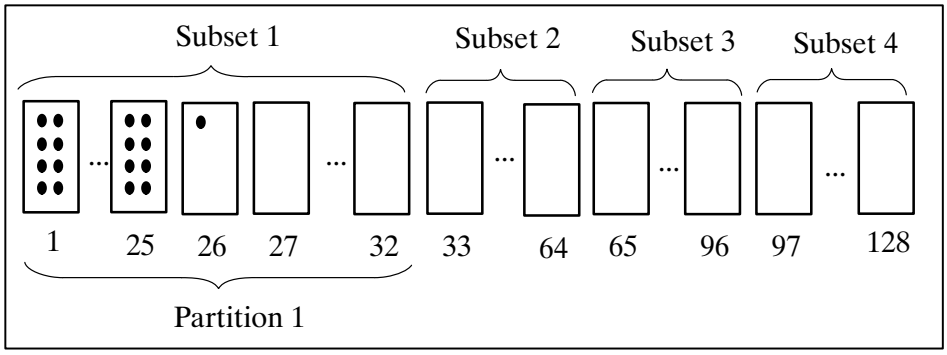

Figure 54. Subsets and (single) partition with multiple-of-eight rule.

Implementation and Statistics

For readers interested in details of the implementation: **partition_mesh** is based on the recursive spectral bisection algorithm. This algorithm solves an eigenvalue problem at each step of the partitioning process using a special implementation of the Lanczos algorithm. The solution of this eigenvalue problem is used to divide the partitions in half. The **partition_mesh** *verbose* option prints statistics at each step of the partitioning process; the man page provides details.

15.11.4 The Partitioning Permutation

The **partition_mesh** routine generates a permutation of the mesh element numbers, and returns the permutation in the integer array ***q***. When element *m* is replaced with element ***q***(*m*), effectively the nodes of the mesh remain unchanged, but the elements are reordered. Reordered elements 1 through *nelpar* comprise the first partition, elements (*nelpar*+1) through 2**nelpar* comprise the second partition, and so on.

Example

Suppose you supply **CMSSL_partition_mesh** with *numproc* = 0 and an ***idual*** array that represents the 11-element mesh shown in Figure 55; and suppose the current machine configuration has four processing elements. Based on the partitioning principles discussed above, **partition_mesh** determines that

Copyright © 1994 Thinking Machines Corporation

four partitions are required; they will hold elements 1−3, 4−6, 7−9, and 10−11, respectively. The **partition_mesh** routine might return the following ***q*** array:

q(1) = 1	***q***(7) = 2
q(2) = 10	***q***(8) = 4
q(3) = 7	***q***(9) = 8
q(4) = 3	***q***(10) = 9
q(5) = 11	***q***(11) = 5
q(6) = 6	

This permutation results in the reordered mesh shown in Figure 55. Heavy lines denote partition boundaries.

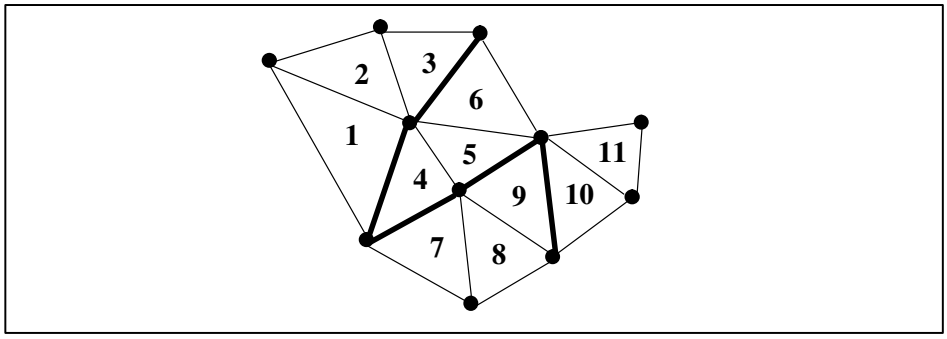

Figure 55. Partitions of a reordered mesh.

15.11.5 Reordering a Pointers Array

The **reorder_pointers** routine reorders a given pointers array, ***p***, along the axis you specify with the *axis* argument, using a permutation ***q***. The array ***q*** must have rank 1 and length equal to the extent of axis *axis* of ***p***, and must be a permutation along axis *axis* of ***p***.

If ***p*** has rank *s*, **reorder_pointers** replaces element (p_1, p_2, ..., p_i, ..., p_s) with element (p_1, p_2, ..., ***q***(p_i), ..., p_s), where i = *axis*. The reordered pointers array can then be supplied to the setup routines for the partitioned gather or scatter (see Sections 15.12 and 15.13).

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

15.11.6 Renumbering a Pointers Array

The **renumber_pointers** routine changes the values of a pointers array, ***p***, to improve data locality. That is, **renumber_pointers** performs the operation

where (***mask***) ***p*** = ***r***(***p***)

where ***p*** is a pointers array, ***r*** is an output integer CM array containing the renumbering mapping, and ***mask*** is a logical CM array used to mask out the elements of ***p*** whose values are less than 1. (These elements would also be masked when ***p*** is supplied to the partitioned gather and scatter utilities, which require ***p*** to be one-based.)

It is recommended that you call **renumber_pointers** before calling the partitioned gather and scatter utilities. If you find that using **renumber_pointers** does not enhance performance, you may omit it.

NOTE

If you want to renumber pointers that represent a mesh you have already partitioned with **partition_mesh**, you must reorder the pointers with **reorder_pointers** before calling **renumber_pointers**.

When you call **renumber_pointers**, you must specify, in the *axis* argument, the axis that counts the mesh elements represented by the pointers. (For example, if your pointers array is an element nodes array, *axis* must be the axis with length *nel*, not axis *node_axis*.) If you are renumbering pointers that have been reordered with **reorder_pointers**, then the value of *axis* in the **renumber_pointers** call must be the same as it was in the **reorder_pointers** call; that is, *axis* must be the axis along which the pointers were reordered.

If you use **renumber_pointers**, you must reorder your data before and after calling the partitioned gather and scatter routines. For example, in a typical application you might follow these steps:

1. Call **renumber_pointers**, which renumbers ***p*** and returns ***r***.

Copyright © 1994 Thinking Machines Corporation

2. Reorder x_0, your initial source array for the gather operation, by mapping x_0 to $x_0(r)$.
3. Perform partitioned gather and scatter operations.
4. Reorder your final array, x_N, by mapping $x_N(r)$ to x_N.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Partitioning of an Unstructured Mesh and Reordering of Pointers

The routines described below allow you to reorder an array of pointers derived from an unstructured mesh so that the off-processing-element communication required by subsequent gather and scatter operations is reduced.

SYNTAX

generate_dual (***idual***, ***ien***, *face_axis, node_axis, elem_type, ier*)

partition_mesh (***idual***, ***q***, *face_axis, numproc, numpar, nelpar, storage_option, verbose, ier*)

reorder_pointers (***p***, ***q***, *axis*)

renumber_pointers (***p***, ***r***, ***mask***, *axis*)

ARGUMENTS

idual Integer one-based CM array that describes an unstructured mesh. Must have rank 2. Axis *face_axis* must have length *nface* and the other axis must have length *nel*, where *nface* is the maximum number of faces per element and *nel* is the number of elements in the mesh. If *face_axis* = 1, this *dual connectivity* array is defined by

idual(m, n) = the element that shares face m with element n; ***idual***$(m, n) = 0$ if face m of element n is a boundary of the mesh. $(1 \leq m \leq nface,\ 1 \leq n \leq nel)$

If *face_axis* = 2, ***idual*** is defined by

idual(n, m) = the element that shares face m with element n; ***idual***$(n, m) = 0$ if face m of element n is a boundary of the mesh. $(1 \leq m \leq nface,\ 1 \leq n \leq nel)$

When you call **generate_dual**, ***idual*** is an output argument; it is the dual connectivity array corresponding to the ***ien*** array you supply.

Copyright © 1994 Thinking Machines Corporation

When you call **partition_mesh**, ***idual*** is an input argument; it is the dual connectivity array that represents the mesh you want to partition.

ien Input integer CM array that describes an unstructured mesh. Must be at least one-based, with rank 2. Axis *node_axis* must have length *nnode* and the other axis must have length *nel*, where *nnode* is the maximum number of nodes per element and *nel* is the number of elements in the mesh. If *node_axis* = 1, this *element nodes* array is defined by

ien(m, n) = the node number of the mth node of the nth mesh element ($1 \leq m \leq nnode$, $1 \leq n \leq nel$).

If *node_axis* = 2, ***ien*** is defined by

ien(n, m) = the node number of the mth node of the nth mesh element ($1 \leq m \leq nnode$, $1 \leq n \leq nel$).

The ***ien*** array you provide to **generate_dual** must use the standard finite element numbering scheme for numbering nodes.

face_axis Scalar integer variable with value 1 or 2. Specifies the axis of ***idual*** that has length *nface*.

node_axis Scalar integer variable with value 1 or 2. Specifies the axis of ***ien*** that has length *nnode*.

elem_type Character*3 scalar string variable. Specifies the type of elements in the mesh whose ***ien*** array you are providing. Must have one of the following values:

Value	Element type
'SEG'	segment
'TRI'	triangle
'QUA'	quadrilateral
'TET'	tetrahedron
'PYR'	pyramid
'PRI'	prism
'HEX'	hexahedron (brick)

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

q When you call **partition_mesh**, ***q*** is an output integer CM array of rank 1 and length equal to the number of elements in the mesh you want to partition. Upon successful completion, ***q*** contains a permutation that reorders the mesh into partitions. When element *n* is replaced with element ***q***(*n*), effectively the nodes of the mesh remain unchanged, but the elements are reordered. Reordered elements 1 through *nelpar* comprise the first partition, elements (*nelpar*+1) through 2**nelpar* comprise the second partition, and so on.

When you call **reorder_pointers**, ***q*** is an input integer CM array of rank 1 and length equal to the extent of axis *axis* of ***p***; ***q*** must be a permutation along axis *axis* of ***p***.

p Integer CM array of any rank and shape.

On input to **reorder_pointers**, ***p*** contains pointers. On successful completion of **reorder_pointers**, ***p*** contains the reordered pointers. If ***p*** has rank *s*, **reorder_pointers** replaces element (p_1, p_2, ..., p_i, ..., p_s) with element (p_1, p_2, ..., ***q***(p_i), ..., p_s), where i = *axis*.

On input to **renumber_pointers**, ***p*** contains pointers. On successful completion of **renumber_pointers**, for each element (p_1, p_2, ..., p_s) for which ***mask*** is **.true.**, ***p***(p_1, p_2, ..., p_s) is replaced by ***r***(***p***(p_1, p_2, ..., p_s)).

axis Scalar integer variable. Must be greater than or equal to 1 and less than or equal to the rank of ***p***.

When you call **reorder_pointers**, *axis* specifies the axis of ***p*** that is to be reordered with ***q***.

When you call **renumber_pointers**, *axis* specifies the axis of ***p*** that counts the mesh elements represented by the pointers. If ***p*** has been reordered with **reorder_pointers**, then the value of *axis* in the **renumber_pointers** call must be the same as it was in the **reorder_pointers** call; that is, *axis* must be the axis along which ***p*** was reordered.

r Integer CM array of rank 1 and extent equal to the maximum value of the array ***p***. Upon successful completion of **renumber_pointers**, ***r*** contains the mapping that **renumber_pointers** has used to renumber ***p*** for improved data locality.

Copyright © 1994 Thinking Machines Corporation

mask Logical CM array of the same rank and shape as ***p***. Since the partitioned gather and scatter utilities require ***p*** to be one-based, ***mask*** is used to mask out the elements of ***p*** whose values are less than 1. If you do not need to mask any elements of ***p***, you may supply the scalar logical value **.true.** for ***mask***.

numproc Scalar integer variable. (Cannot be a constant.) On input, must be either ≤ 0 or a positive power of 2 that is less than or equal to **CMF_NUMBER_OF_PROCESSORS( )**. When **partition_mesh** divides the processing elements into subsets, as discussed in the Description section, the input value of *numproc* determines the number of subsets, as follows:

- If *numproc* ≤ 0, there are as many subsets as processing elements; each subset consists of just one processing element.
- If *numproc* $= n > 0$, there are n subsets.

On return, *numproc* contains the number of subsets.

numpar Output scalar integer variable. On return, contains the number of partitions formed by the reordered mesh.

nelpar Output scalar integer variable. On return, when padding is not in effect, *nelpar* contains the number of elements in each partition (with the possible exception of the last partition, which may contain fewer elements). When padding is in effect, *nelpar* contains the *largest* number of elements each partition can contain.

storage_option Scalar integer variable. Must be either **CMSSL_partition_default** or **CMSSL_partition_low_storage**. The low-storage option is slower than the default option, but uses less storage for working arrays. In particular, **CMSSL_partition_default** uses approximately (309 + 3*nface*) doublewords per mesh element, and **CMSSL_partition_low_storage** uses approximately (11 + 3*nface*) doublewords per mesh element. These storage evaluations are not exact, but very closely approximate the actual memory usage.

verbose Scalar logical variable. If **.true.**, *verbose* causes **partition_mesh** to print statistics to standard output at each step of the partitioning algorithm. For details, see the Description section.

Copyright © 1994 Thinking Machines Corporation

ier Scalar integer variable. Error code. On return from **generate_dual**, *ier* has one of the following values:

0	Successful return.
−1	Invalid *elem_type*.
−2	Invalid dimensions for ***idual*** or ***ien***.
−3	***ien*** is not one-based.
−4	The mesh topology defined by the ***ien*** array is incorrect (for example, more than two elements share a face).
−5	Invalid *face_axis*.
−6	Invalid *node_axis*.

On return from **partition_mesh**, *ier* has one of the following values:

0	Successful return.
−1	The second dimension of ***idual*** is not equal to the length of ***q***.
−2	*numproc* is invalid on input (must be either ≤ 0 or a positive power of 2 that is $\leq$ **CMF_NUMBER_OF_PROCESSORS( )**).
−3	Invalid *face_axis*.

DESCRIPTION

The routines described here perform the following operations:

generate_dual Given an *element nodes* array, ***ien***, that describes an unstructured mesh, this routine produces the corresponding *dual connectivity* array, ***idual***.

Copyright © 1994 Thinking Machines Corporation

partition_mesh Given a dual connectivity array describing an unstructured mesh, this routine returns a permutation, ***q***, that reorders the mesh elements to form discrete *partitions*. The routine also returns the number of resulting partitions and the number of elements per partition.

reorder_pointers Given a pointers array, ***p***, and a permutation ***q*** (for example, the permutation returned by **partition_mesh)**, this routine reorders the pointers array along axis *axis* using ***q***.

renumber_pointers Given a pointers array, ***p***, that has been reordered by **reorder_pointers**, this routine changes the pointer values to improve data locality and returns the renumbering mapping in ***r***. That is, **renumber_pointers** performs the operation

where (***mask***) $p = r(p)$.

(This routine also works on any pointers array ***p*** that represents a mesh.)

Mixed Element Types. In the current implementation, the **generate_dual** routine requires that all elements of the mesh have the same type (the same number of nodes). However, you can work with a mesh with different element types when you call **partition_mesh**, by setting the extra faces of the input array ***idual*** to zero. For example, to supply **partition_mesh** with a mesh with both triangles (3 faces) and quadrilaterals (4 faces), where *face_axis* = 1, set *nface* = 4 (the larger of the two), and set ***idual***$(4, n) = 0$ if element n is a triangle.

Partitioning Rules. The **partition_mesh** routine determines the number of partitions based on the number of mesh elements and the input value of *numproc*, as described below. Note that the term *processing element* refers to the value returned by **CMF_NUMBER_OF_PROCESSORS( )**.

The elements of the mesh (represented by the ***idual*** array) are laid out on the processing elements by the compiler; they span as many processing elements as possible, subject to the multiple-of-eight rule if that rule is in effect. Partitioning is accomplished as follows:

1. The **partition_mesh** routine divides the processing elements into equal subsets (that is, each subset contains the same number of processing elements). The number of subsets is determined by the input value of *numproc*, which have the following effects:

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- If *numproc* ≤ 0, there are as many subsets as processing elements; each subset consists of just one processing element.
- If *numproc* $= n > 0$, there are n subsets. If n is strictly less than **CMF_NUMBER_OF_PROCESSORS( )**, then each subset contains multiple processing elements.

2. The routine then reorders the mesh elements and divides them among the subsets. When this process is complete, all subsets except the last hold the same number, k, of mesh elements; the last subset may hold $l \leq k$ elements.
3. Partitions are mapped to subsets. Each partition is mapped to all of the processing elements in the associated subset, except the last partition, which may be mapped to only some of the processing elements in the subset. A processing element never contains mesh elements from more than one partition.

The reordering and subdividing of the mesh elements in step 2 occurs in steps. If *numproc* $= n > 0$, there are $\log_2(n)$ steps in this process.

Printed Statistics. If you set *verbose* to **.true.**, **partition_mesh** prints statistics to standard output at each step of the partitioning algorithm. The following output is produced when *verbose* = **.true.** in the **partition_mesh** sample code (see Examples, below) and the code is run on a 32-node CM-5:

```
Statistics:  92  0.99899E-03  0.99899E-03
Eigenvalues:  0.93834E-03  0.93834E-03

Statistics: 157  0.25159E-03  0.98060E-03
Eigenvalues:  0.38574E-03  0.80637E-03

Statistics:  89  0.12348E-03  0.99106E-03
Eigenvalues:  0.13715E-02  0.18081E-02

Statistics: 108  0.38294E-05  0.92622E-03
Eigenvalues:  0.15983E-02  0.82448E-02

Statistics:  70  0.12419E-06  0.99841E-03
Eigenvalues:  0.42342E-02  0.25285E-01

Statistics:  69  0.29024E-07  0.88740E-03
Eigenvalues:  0.10248E-01  0.25284E-01

Statistics:  48  0.24599E-13  0.60988E-03
```


Copyright © 1994 Thinking Machines Corporation

```
Eigenvalues:  0.11724E-01  0.10398E+00

Number of graph edges:       10169
Number of graph edge cuts:   1394
Percentage of cuts:        13.71 %
```

The partitioning routine is based on the recursive spectral bisection algorithm. This algorithm solves an eigenvalue problem at each step of the partitioning process using a special implementation of the Lanczos algorithm. The solution of this eigenvalue problem is used to split the partitions in half. If *numproc* is the number of processing elements, there will be $\log_2(numproc)$ steps in the partitioning process (in the first step, the whole mesh is subdivided into 2 partitions, each of which gets subdivided into 2 partitions during the second step, and so on). At each step of the partitioning process, the following information is printed:

- `Statistics:`

  Number of Lanczos iterations
  Minimum, over the current partitions, of the error bound for Lanczos
  Maximum, over the current partitions, of the error bound for Lanczos

- `Eigenvalues`:

  Minimum, over the current partitions, of the smallest non-zero eigenvalue
  Maximum, over the current partitions, of the smallest non-zero eigenvalue

The last 3 lines printed represent a measure of the quality of the partitioning:

- `Number of graph edges:`

  This is the total number of edges in the graph used as the representation of the problem to be partitioned. In the case of a mesh, it corresponds to the number of interior faces in the mesh (that is, excluding the boundary faces).

- `Number of graph edge cuts:`

  This is the number of edges in the graph that are cut by the partitioning routine. In the case of a mesh, it is the number of faces shared by 2 partitions.

- `Percentage of cuts:`

  This number is

  100*(`Number of graph edge cuts`/`Number of graph edges`).

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

NOTES

Include the CMSSL Header File. The **partition_mesh** routine uses predefined symbolic constants. Therefore, you must include the statement

```
INCLUDE '/usr/include/ cm/cmssl-cmf.h'
```

at the top of the main file of any program that uses this routine. This file defines symbolic constants and declares the type of the CMSSL functions.

Arrays with Inactive Elements. For arrays that are larger than the number of active elements, set the inactive components of ***idual*** to 0.

Use of the Renumbering Routine. It is recommended that you call **renumber_pointers** before calling the partitioned gather and scatter utilities. If you find that using **renumber_pointers** does not enhance performance, you may omit it.

If you want to renumber pointers that represent a mesh you have already partitioned with **partition_mesh**, you must reorder the pointers with **reorder_pointers** before calling **renumber_pointers**.

If you use **renumber_pointers**, you must reorder your data before and after calling the partitioned gather and scatter routines. For example, in a typical application you might follow these steps:

1. Call **renumber_pointers**, which renumbers ***p*** and returns ***r***.
2. Reorder x_0, your initial source array for the gather operation, by mapping x_0 to $x_0(r)$.
3. Perform partitioned gather and scatter operations.
4. Reorder your final array, x_N, by mapping $x_N(r)$ to x_N.

Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the **generate_dual**, **partition_mesh**, **reorder_pointers**, and **renumber_pointers** routines can be found on-line in the subdirectory

```
partitioning/cmf
```

of a CMSSL examples directory whose location is site-specific.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

15.12 Partitioned Gather Utility

The partitioned gather utility performs the same operations as the sparse gather and sparse vector gather routines. If you supply a pointers array that is reordered to achieve data locality, the partitioned gather takes advantage of this locality, reducing communication time.

If your pointers are derived from a mesh, you can produce a partitioned pointers array using the **partition_mesh** and **reorder_pointers** routines described in Section 15.11.

The partitioned gather utility is described in the man page that follows.

Copyright © 1994 Thinking Machines Corporation

Partitioned Gather Utility

Given a source array, a destination array, and a pointers array containing a gathering pattern, the routines described below gather elements or vectors from the source array into the destination array. If the pointers array is reordered to achieve data locality, communication time is reduced.

SYNTAX

part_gather_setup (***p, y_mask, x_template****, setup, ier*)

part_gather (***y, x, y_mask****, setup*)

part_vector_gather (***y, x, y_mask****, y_vector_axis, x_vector_axis, setup*)

deallocate_part_gather_setup (*setup*)

ARGUMENTS

p One-based CM array whose dimensions must satisfy the following conditions:

- ***p*** must have the same axis extents and layout directives as the array ***y*** you supply in any subsequent associated call to **part_gather**.
- ***p*** must have the same axis extents and layout directives as the subarray (formed by omitting axis *y_vector_axis*) of the array ***y*** you supply in any subsequent associated call to **part_vector_gather**.

The array ***p*** contains pointers that indicate the gathering pattern, as follows. If ***y_mask***(p_1, ..., p_k) = **.true.** and element (p_1, ..., p_k) of ***p*** contains the value n, then

- In a subsequent call to **part_gather**, element ***x***(n) is gathered from the source array to location ***y***(p_1, ..., p_k) in the destination array during the gather operation.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- In a subsequent call to **part_vector_gather**, the vector $\boldsymbol{x}(:,n)$ or $\boldsymbol{x}(n, :)$ is gathered from the source array to locations $\boldsymbol{y}(p_1, ..., :, ..., p_k)$ in the destination array during the gather operation (where : denotes axis *x_vector_axis* of ***x*** and axis *y_vector_axis* of ***y***).

The contents of ***p*** remain unchanged by **part_gather_setup**.

The **part_gather_setup** routine takes advantage of possible data locality to reduce communication time.

y_mask If you need to mask elements of ***y***, declare ***y_mask*** as a logical CM array with the same axis extents and layout directives as

- The array ***y*** you supply in any subsequent associated call to **part_gather**, or
- The subarray (formed by omitting axis *y_vector_axis*) of the array ***y*** you supply in any subsequent associated call to **part_vector_gather**.

Set to **.true.** the elements that correspond to active elements of ***y***. The contents of ***y_mask*** remain unchanged by **part_gather_setup**, **part_gather**, and **part_vector_gather**.

If you do not need to mask elements of ***y***, you can conserve processing element memory and get better performance by supplying the scalar logical value **.true.** for ***y_mask***.

You may set any component of ***y_mask*** to **.false.** during the course of the computation without calling the setup routine again. However, a component of ***y_mask*** that has been set to **.false.** before the **part_gather_setup** call cannot be set to **.true.** during the computation.

x_template CM array of any type with the same axis extent and layout directives as

- The array ***x*** you supply in any subsequent associated call to **part_gather**, or
- The subarray, formed by omitting axis *x_vector_axis*, of the array ***x*** you supply in any subsequent associated call to **part_vector_gather**.

The setup routine uses only the shape and layout of this array, ignoring the contents.

Copyright © 1994 Thinking Machines Corporation

y CM array of any type with arbitrary shape. Destination array to which elements or vectors from the source array are gathered. The values of active elements of ***y*** are overwritten.

When you call **part_gather**, ***y*** must have the same layout as the array ***p*** you supplied in the **part_gather_setup** call.

When you call **part_vector_gather**, you must declare ***y*** with an extra local axis (axis *y_vector_axis*) with the same extent as axis *x_vector_axis* of ***x***. Upon return from **part_vector_gather**, the gathered vectors will lie along this axis of ***y***. The remaining axes of ***y*** must have the same layout as the corresponding axes of the array ***p*** you supplied in the **part_gather_setup** call.

If you use these routines in conjunction with the **partition_mesh** routine, you should declare all axes of ***y*** to be local (that is, either **:serial** or laid out with **:procs** = 1) except the reordered axis, which should be non-local. This strategy achieves maximum locality and yields the best performance.

x CM array of rank 1 (in a call to **part_gather**) or 2 (in a call to **part_vector_gather**), with the same type as ***y***. Source array from which elements or vectors are gathered. In a call to **part_vector_gather**, the vectors to be gathered must lie along axis *x_vector_axis* of ***x***. The contents of ***x*** remain unchanged by **part_gather** and **part_vector_gather**.

y_vector_axis Scalar integer variable. Specifies the axis of ***y*** along which the gathered vectors will lie. This axis must be local (that is, either **:serial** or laid out with **:procs** = 1), and must have the same extent as axis *x_vector_axis* of ***x***.

x_vector_axis Scalar integer variable. Specifies the axis of ***x*** along which the vectors to be gathered must lie. This axis must be local (that is, either **:serial** or laid out with **:procs** = 1), and must have the same extent as axis *y_vector_axis* of ***y***.

setup Scalar integer variable. Internal variable. The initial value you supply to **part_gather_setup** is ignored. You must supply **part_gather**, **part_vector_gather**, and **deallocate_part_gather_setup** with the value that **part_gather_setup** assigns to *setup*.

ier Scalar integer variable. Upon return from **part_gather_setup**, contains one of the following codes:

Copyright © 1994 Thinking Machines Corporation

0	Successful return.
−1	Invalid arguments (for example, mismatched sizes or shapes).

DESCRIPTION

Definition. The gather operation is defined by

where (***y_mask***) $y = x(p)$ (**part_gather**)
where (***y_mask***) $y = x(:, p)$ (**part_vector_gather**)

where ***x*** is the array from which elements or vectors are being gathered, ***y*** is the resulting destination array, and ***p*** is an array of pointers.

Usage. Follow these steps to perform a gather operation (or multiple gather operations, sequentially):

1. Call **part_gather_setup**.
2. Call **part_gather** or **part_vector_gather**.

   To perform more than one gather operation using the same sparsity (gathering pattern), follow one call to **part_gather_setup** with multiple calls to **part_gather** and/or **part_vector_gather**. (You may intersperse calls to these two routines.) If the sparsity changes, start with Step 1 again.
3. After all **part_gather** and **part_vector_gather** calls associated with the same **part_gather_setup** call, call **deallocate_part_gather_setup** to deallocate the processing element storage space required by the setup routine.

You may have more than one setup active at a time; that is, you may call the setup routine more than once without calling the deallocation routine.

Setup Phase. The **part_gather_setup** routine analyzes the gathering pattern supplied by the application in the ***p*** argument. Using ***p*** and ***y_mask***, **part_gather_setup** computes an optimization, or *trace*, for the communication required by the gather operation; allocates the required storage space; and saves the trace for use in subsequent calls to the **part_gather** or **part_vector_gather** routine. The setup routine assigns an appropriate value to the internal variable *setup*, which must be supplied in subsequent calls to **part_gather**, **part_vector_gather**, and **deallocate_part_gather_setup**.

Copyright © 1994 Thinking Machines Corporation

The saving of the trace saves communication time, particularly when one setup call is amortized by several gather operations.

Gather Phase. The **part_gather** and **part_vector_gather** routines gather elements or vectors, respectively, from ***x*** into ***y***, using the communication pattern saved by a previous call to **part_gather_setup**.

As long as the arguments supplied to **part_gather_setup** remain the same (except for ***y_mask***; see above), the application can call **part_gather** and/or **part_vector_gather** multiple times following one call to **part_gather_setup**, each time supplying *setup*, a mask, and a source array, ***x***, and receiving in return a destination array, ***y***.

Deallocation Phase. The **deallocate_part_gather_setup** routine deallocates the extra storage space that **part_gather_setup** allocated for saving the trace. Each **part_gather_setup** call should be followed (after one or more associated calls to **part_gather** and/or **part_vector_gather**) by a **deallocate_part_gather_setup** call.

NOTES

Trace Deallocation. It is strongly recommended that you call **deallocate_part_gather_setup** as soon as the associated gather operations have finished, as the trace typically occupies a significant amount of processing element storage.

Permutation of the Source Array. Some applications require permutation of the source array prior to the gather operation. This permutation is the responsibility of the user application and is not performed by **part_gather_setup**, **part_gather**, or **part_vector_gather**.

Layout and Performance. For best performance, axes *x_vector_axis* and *y_vector_axis* should be either the leftmost or the rightmost axes of their respective arrays. If you use these routines in conjunction with the **partition_mesh** routine, you should declare all axes of ***y*** to be local (that is, either **:serial** or laid out with **:procs** = 1) except the reordered axis, which should be non-local. This strategy achieves maximum locality and yields the best performance.

Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the partitioned gather and scatter utilities can be found on-line in the subdirectory

```
partitioning/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.13 Partitioned Scatter Utility

The partitioned scatter utility performs the same operations as the sparse scatter and sparse vector scatter routines. If you supply a pointers array that is reordered to achieve data locality, the partitioned scatter takes advantage of this locality, reducing communication time.

If your pointers are derived from a mesh, you can produce a partitioned pointers array using the **partition_mesh** and **reorder_pointers** routines described in Section 15.11.

The partitioned scatter utility is described in the man page that follows.

Copyright © 1994 Thinking Machines Corporation

Partitioned Scatter Utility

Given a source array, a destination array, and a pointers array containing a scattering pattern, the routines described below scatter elements or vectors from the source array to the destination array. If the pointers array is reordered to achieve data locality, communication time is reduced.

SYNTAX

part_scatter_setup (***p, x_mask, y_template****, setup, ier*)

part_scatter (***y, x, x_mask****, setup*)

part_vector_scatter (***y, x, x_mask****, y_vector_axis, x_vector_axis, setup*)

deallocate_part_scatter_setup (*setup*)

ARGUMENTS

p One-based integer CM array whose dimensions must satisfy the following conditions:

- ***p*** must have the same axis extents and layout directives as the array ***x*** you supply in any subsequent associated call to **part_scatter**.
- ***p*** must have the same axis extents and layout directives as the subarray, formed by omitting axis *x_vector_axis*, of the array ***x*** you supply in any subsequent associated call to **part_vector_scatter**.

The array ***p*** contains pointers that indicate the scattering pattern, as follows. If ***x_mask***$(p_1, ..., p_k)$ = **.true.** and element $(p_1, ..., p_k)$ of ***p*** contains the value n, then

- A subsequent call to **part_scatter** adds element ***x***$(p_1, ..., p_k)$ of the source array to destination element ***y***(n).

Copyright © 1994 Thinking Machines Corporation

- A subsequent call to **part_vector_scatter** adds the vector $\boldsymbol{x}(p_1, ..., :, ..., p_k)$ to the vector $\boldsymbol{y}(:, n)$ or $\boldsymbol{y}(n, :)$ within the destination array (where : denotes axis *x_vector_axis* of ***x*** and axis *y_vector_axis* of ***y***).

If two or more source array elements are sent to the same destination array element, the colliding destination values are added.

The contents of ***p*** remain unchanged by **part_scatter_setup**.

The **part_scatter_setup** routine takes advantage of possible data locality to reduce communication time.

x_mask If you need to mask elements of ***x***, declare ***x_mask*** as a logical CM array with the same axis extents and layout directives as

- The array ***x*** you supply in any subsequent associated call to **part_scatter**, or
- The subarray, formed by omitting axis *x_vector_axis*, of the array ***x*** you supply in any subsequent associated call to **part_vector_scatter**.

Set to **.true.** the elements that correspond to active elements of ***x***. Only those source array elements (or vectors) for which the mask is true are sent to the destination array. Elements of ***p*** corresponding to masked locations of the source array are ignored. The contents of ***x_mask*** remain unchanged by **part_scatter_setup**.

If you do not need to mask elements of ***x***, you can conserve processing element memory and get better performance by supplying the scalar logical value **.true.** for ***x_mask***.

You may set any component of ***x_mask*** to **.false.** during the course of the computation without calling the setup routine again. However, a component of ***x_mask*** that has been set to **.false.** before the **part_gather_setup** call cannot be set to **.true.** during the computation.

y_template CM array of any type with the same axis extent and layout directives as

- The array ***y*** you supply in any subsequent associated call to **part_scatter**, or

Copyright © 1994 Thinking Machines Corporation

- The subarray, formed by omitting axis *y_vector_axis*, of the array ***y*** you supply in any subsequent associated call to **part_vector_scatter**.

The setup routine uses only the shape and layout of this array, ignoring the contents.

y CM array of any type and rank 1 (in a call to **part_scatter**) or 2 (in a call to **part_vector_scatter**). The **part_scatter** routine adds the scattered elements from ***x*** to the initial values of ***y***. The **part_vector_scatter** routine adds the scattered vectors from ***x*** to the initial values of the vectors that lie along axis *y_vector_axis* of ***y***. This axis must be local (that is, either **:serial** or laid out with **:procs** = 1), and must have the same extent as axis *x_vector_axis* of ***x***.

x CM array of arbitrary shape and of the same type and precision as ***y***. Source array from which elements or vectors are scattered. The contents of ***x*** remain unchanged by **part_scatter** and **part_vector_scatter**.

When you call **part_scatter**, ***x*** must have the same layout as the array ***p*** you supplied in the **part_scatter_setup** call.

When you call **part_vector_scatter**, you must declare ***x*** with an extra local axis (axis *x_vector_axis*) with the same extent as axis *y_vector_axis* of ***y***. The vectors to be scattered must lie along this axis of ***x***. The remaining axes of ***x*** must have the same layout as the corresponding axes of the array ***p*** you supplied in the **part_scatter_setup** call.

If you use these routines in conjunction with the **partition_mesh** routine, you should declare all axes of ***x*** to be local (that is, either **:serial** or laid out with **:procs** = 1) except the reordered axis, which should be non-local. This strategy achieves maximum locality and yields the best performance.

y_vector_axis Scalar integer variable. Specifies the axis of ***y*** along which the scattered vectors will lie. This axis must be local (that is, either **:serial** or laid out with **:procs** = 1), and must have the same extent as axis *x_vector_axis* of ***x***.

x_vector_axis Scalar integer variable. Specifies the axis of ***x*** along which the vectors to be scattered must lie. This axis must be local (that is, either **:serial** or laid out with **:procs** = 1), and must have the same extent as axis *y_vector_axis* of ***y***.

Copyright © 1994 Thinking Machines Corporation

setup Scalar integer variable. Internal variable. The initial value supplied to **part_scatter_setup** is ignored. You must supply **part_scatter**, **part_vector_scatter**, and **deallocate_part_scatter_setup** with the value assigned to *setup* by **part_scatter_setup**.

ier Scalar integer variable. Upon return from **part_scatter_setup**, contains one of the following codes:

0	Successful return.
−1	Invalid arguments (for example, mismatched sizes or shapes).

DESCRIPTION

Definition. The scatter operation is defined by

where (***x_mask***) $y(p\,|+) = x$ (**part_scatter**)
where (***x_mask***) $y(:, p\,|+) = x$ (**part_vector_scatter**)

where x is the array from which elements or vectors are being scattered, y is the resulting destination array, and p is an array of pointers.

Usage. Follow these steps to perform a scatter operation (or multiple scatter operations, sequentially):

1. Call **part_scatter_setup**.
2. Call **part_scatter** or **part_vector_scatter**.

   To perform more than one scatter operation using the same sparsity (scattering pattern), follow one call to **part_scatter_setup** with multiple calls to **part_scatter** and/or **part_vector_scatter**. (You may intersperse calls to these two routines.) If the sparsity changes, start with Step 1 again.
3. After all **part_scatter** and **part_vector_scatter** calls associated with the same **part_scatter_setup** call, call the **deallocate_part_scatter_setup** routine to deallocate the processing element storage space required by the setup routine.

You may have more than one setup active at a time; that is, you may call the setup routine more than once without calling the deallocation routine. However, calling the setup routine repeatedly without calling the deallocation routine may cause you to run out of memory. It is therefore strongly recommended that you call

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

deallocate_part_scatter_setup as soon as you have finished the associated scatter operations.

Setup Phase. The **part_scatter_setup** routine analyzes the scattering pattern supplied by the application in the ***p*** argument. Using ***p*** and ***x_mask***, **part_scatter_setup** assigns an appropriate value to the internal variable *setup*, which must be supplied in subsequent calls to **part_scatter** and **part_vector_scatter**.

Scatter Phase. The **part_scatter** and **part_vector_scatter** routines scatter elements or vectors, respectively, from ***x*** and add them to the initial values of ***y***, using the information returned by a previous call to the **part_scatter_setup** routine.

As long as the arguments supplied to **part_scatter_setup** remain the same (except for ***x_mask***; see above), the application can call **part_scatter** and/or **part_vector_scatter** multiple times following one **part_scatter_setup** call, each time supplying a source array, ***x***, and a mask, ***x_mask***, and receiving in return a destination array, ***y***.

Deallocation Phase. The **deallocate_part_scatter_setup** routine deallocates the extra storage space that the setup routine allocated. Each call to the setup routine should be followed (after one or more calls to **part_scatter** and/or **part_vector_scatter**) by a **deallocate_part_scatter_setup** call.

NOTES

Layout and Performance. For best performance, axes *x_vector_axis* and *y_vector_axis* should be either the leftmost or the rightmost axes of their respective arrays. If you use these routines in conjunction with the **partition_mesh** routine, you should declare all axes of ***x*** to be local (that is, either **:serial** or laid out with **:procs** = 1) except the reordered axis, which should be non-local. This strategy achieves maximum locality and yields the best performance.

Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the partitioned gather and scatter utilities can be found on-line in the subdirectory

```
partitioning/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

15.14 Communication Compiler

CMSSL provides a set of routines, referred to collectively as the *communication compiler*, that compute and use message delivery optimizations for basic data motion and combining operations. The data motion and combining operations are similar to those performed by the CM Fortran utilities **CMF_SEND_***combiner* (for example, **CMF_SEND_ADD**), which are described in the CM Fortran documentation set.

A *message* is a piece of data that must get from a source location to a destination location. The set of all messages that must be delivered during an operation such as a get or a send is called the *communication pattern*. The communication compiler allows you to compute an optimization, or *trace*, for a communication pattern just once, and then use it many times in subsequent operations. This feature can yield significant time savings (with an associated memory cost) in applications that use the same communication pattern repeatedly. The process of computing a trace for later use is sometimes referred to as *compilation* of the trace.

The communication compiler offers a variety of methods for computing a trace. You can either select a method suited to your application, or allow the communication compiler to choose an appropriate method based on the number of times you plan to perform a specific operation.

15.14.1 Communication Compiler Routines

The communication compiler includes the following routines:

- Setup routine: **comm_setup**

  This routine computes a trace for a specified type of operation, using a selected trace compilation method. You must supply the setup routine with the shapes and layouts of the source and destination arrays on which you will be operating, and with the destination locations to which source elements are to be moved (for a send operation) or the source locations from which elements are to be moved to the destination (for a get operation). The setup routine computes a trace that can only be used on the current partition size. It allocates processing element and partition manager memory for storing the trace and related information. It returns an integer, *trace*, that contains a pointer to the partition manager memory where information about the trace (including pointers to processing element fields) is stored.

Copyright © 1994 Thinking Machines Corporation

- Data motion and combining routines:

  comm_get
  comm_send
  comm_send_add
  comm_send_and
  comm_send_max
  comm_send_min
  comm_send_or
  comm_send_xor

  Each of these routines uses a trace previously computed by the setup routine to perform the specified operation.

- Compilation option routine: **comm_set_option**

  This routine selects an option that prints error information. On the CM-200, this routine also selects options that can help optimize trace compilation and message delivery. The optimization options are not available in this CM-5 release, as they affect only those compilations performed using the FastGraph method (which is also unavailable in this release on the CM-5).

- Deallocation routine: **deallocate_comm_setup**

  This routine deallocates the memory that **comm_setup** allocated to store a trace. Once this memory is deallocated, the specified trace can no longer be used.

NOTE

The trace saving and restore routines, **comm_save_trace** and **comm_restore_trace**, available on the CM-200, are currently unavailable on the CM-5. Attempts to call these routines on the CM-5 result in an error code.

Copyright © 1994 Thinking Machines Corporation

15.14.2 How to Use the Communication Compiler

The trace computed by the setup routine can be used for any operation that satisfies the following conditions:

- The type of operation must match the operation type you specified in the setup call. (The setup routine uses the same operation type for the **comm_send_add**, **comm_send_and**, **comm_send_max**, **comm_send_min**, **comm_send_or**, and **comm_send_xor** operations, as these operations require the same setup action. A trace computed for any of these operations can be used for any of the others, assuming the other two conditions are met.)

- The source and destination arrays must have the same rank, axis extents, and layout directives as in the setup call.

- The source and destination arrays must have the same data type, and that data type must be valid for the operation to be performed.

You can follow a setup call with multiple calls to the data motion and combining routines, as long as the above conditions are met. The arguments you supplied in the setup call determine how data is moved in the associated data motion and combining routine calls. You can also have more than one trace allocated at a time; the only limit on the number of concurrently active traces is the amount of memory.

To compute a trace and use it in the same program run, follow these steps:

1. Call **comm_setup** to allocate processing element memory and compute the trace.

2. Call the desired data motion or combining routine. (You can repeat this step an arbitrary number of times.)

3. Call **deallocate_comm_setup**.

If you want to set a compilation option, call the **comm_set_option** routine prior to calling the setup routine.

Copyright © 1994 Thinking Machines Corporation

NOTE

The arrays you supply to the communication compiler must be one-based.

For detailed information about the communication compiler routines, see the man page that follows.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Communication Compiler

Given a source array, a destination array, and gathering or scattering coordinates, the routines described below compute and use message delivery optimizations for basic data motion and combining operations.

SYNTAX

trace = **comm_setup** (*y, p, x,* {*x*|*y*}***_mask****, operation, method, ier*)

comm_get (*y, trace, x, ier*)

comm_send (*y, trace, x, ier*)

comm_send_add (*y, trace, x, ier*)

comm_send_and (*y, trace, x, ier*)

comm_send_max (*y, trace, x, ier*)

comm_send_min (*y, trace, x, ier*)

comm_send_or (*y, trace, x, ier*)

comm_send_xor (*y, trace, x, ier*)

comm_set_option (*option, value, ier*)

deallocate_comm_setup (*trace*)

ARGUMENTS

In the following argument descriptions, the **comm_get**, **comm_send**, **comm_send_add**, **comm_send_and**, **comm_send_max**, **comm_send_min**, **comm_send_or**, and **comm_send_xor** routines are referred to as *data motion and combining routines*.

The message delivery optimization for a given operation is referred to as a *trace*.

trace — Scalar integer variable. Returned by **comm_setup**; contains the address of a scalar structure that contains information necessary to deliver the data, including CM addresses. When you call one of the data motion or combining routines, you must supply the *trace* value returned by **comm_setup**.

Copyright © 1994 Thinking Machines Corporation

When you call **deallocate_comm_setup**, supply the value representing the trace you want to deallocate.

y One-based CM array. When you call **comm_setup**, the contents of ***y*** are ignored; only the shape and layout are used. If you are computing a trace for a get operation, ***y*** can have any rank ≤ 6. If you are computing a trace for any of the send operations, ***y*** can have any rank ≤ 7.

When you call a data motion or combining routine, ***y*** must have the same rank, axis extents, and layout directives as the ***y*** you supplied in the **comm_setup** call that created the trace you are using. This is the array to which elements of ***x*** will be moved; it should have the same data type as ***x***. The valid data types depend on the operation you are performing; see the Description section for details.

p One-based integer CM array that defines the *gathering coordinates* (the source locations from which active destination elements fetch their new values) for a get operation, or the *scattering coordinates* (the destination locations to which source array elements are to be moved) for the various send operations.

If you are computing a trace for **comm_get**, ***p*** must satisfy the following conditions:

- The rank of ***p*** must be one greater than the rank of ***y***.
- The extra axis of ***p*** must be last, have extent greater than or equal to the rank of ***x***, and be local.
- The remaining axes of ***p*** must match the axes of ***y*** in order of declaration, extents, and layout directives.
- If ***y*** has rank n and ***x*** has rank m, then to gather data into $\boldsymbol{y}(y_1, y_2, \ldots, y_n)$ from $\boldsymbol{x}(x_1, x_2, \ldots, x_m)$, set $\boldsymbol{p}(y_1, y_2, \ldots, y_n, q) = x_q$ for $1 \leq q \leq m$.

If you are computing a trace for **comm_send**, **comm_send_add**, **comm_send_and**, **comm_send_max**, **comm_send_min**, **comm_send_or**, or **comm_send_xor**, ***p*** must satisfy the following conditions:

- The rank of ***p*** must be one greater than the rank of ***x***.

Copyright © 1994 Thinking Machines Corporation

- The extra axis of ***p*** must be last, have extent greater than or equal to the rank of ***y***, and be local.
- The remaining axes of ***p*** must match the axes of ***x*** in order of declaration, extents, and layout directives.
- If ***y*** has rank n and ***x*** has rank m, then to send data from $\boldsymbol{x}(x_1, x_2, \ldots, x_m)$ to $\boldsymbol{y}(y_1, y_2, \ldots, y_n)$, set $\boldsymbol{p}(x_1, x_2, \ldots, x_m, q) = y_q$ for $1 \leq q \leq n$.

x One-based CM array. When you call **comm_setup**, the contents of ***x*** are ignored; only the shape and layout are used. If you are computing a trace for a get operation, ***x*** can have any rank ≤ 7. If you are computing a trace for any of the send operations, ***x*** can have any rank ≤ 6.

When you call a data motion or combining routine, ***x*** must have the same rank, axis extents, and layout directives as the ***x*** you supplied in the **comm_setup** call that created the trace you are using. This is the array whose elements will be moved into ***y***; it should have the same data type as ***y***. The valid data types depend on the operation you are performing; see the Description section for details.

{x|y}_mask If you are computing a trace for a get operation, this argument is ***y_mask***, a mask for ***y***. It can be either a logical CM array with the same shape and layout as ***y***, or the scalar value **.TRUE.**. Only those elements of ***y*** that correspond to true ***y_mask*** values are overwritten.

If you are computing a trace for any of the send operations, this argument is ***x_mask***, a mask for ***x***. It can be either a logical CM array with the same shape and layout as ***x***, or the scalar value **.TRUE.**. Only those elements of ***x*** that correspond to true ***x_mask*** values are moved into ***y***.

operation Scalar integer variable that specifies the operation for which you plan to use the trace. Can be any of the following symbolic constants:

CMSSL_comm_by_get
CMSSL_comm_by_send
CMSSL_comm_by_send_overwrite
CMSSL_comm_by_send_with_op

Copyright © 1994 Thinking Machines Corporation

The values **CMSSL_comm_by_send** and **CMSSL_comm_by_send_overwrite** both represent the **comm_send** operation, but they request different behavior with respect to collisions. (In a **comm_send** operation, two or more source elements are said to *collide* if they are sent to the same destination array location.) If the scattering coordinates you supply contain collisions, **CMSSL_comm_by_send** informs you of this condition by returning *ier* = −4, while **CMSSL_comm_by_send_overwrite** does not return an error. The code *ier* = −4 is only a warning; the trace computed by **CMSSL_comm_by_send** is valid. For details about collision handling, see the Description section.

The value **CMSSL_comm_by_send_with_op** represents the **comm_send_add**, **comm_send_and**, **comm_send_max**, **comm_send_min**, **comm_send_or**, and **comm_send_xor** operations.

method

Usually a one-dimensional integer front-end array. The first element specifies the method to be used in computing the trace, and must have one of the values listed below. Some methods require additional information which you must supply in subsequent elements, as described below. The array may have any length; the elements beyond those that are required are ignored. If you choose a method that does not require any additional information, you may define *method* as a scalar integer with one of the values listed below.

CMSSL_method_automatic

Selects one of the trace compilation methods based on your estimate of how many times this trace will be used before it is discarded. You must specify this estimate in *method*(2). If *method*(2) is a large number, the setup routine will choose a method that is likely to require more time for trace generation, but yield faster communication. If *method*(2) is a small number, the setup routine will choose a method that is likely to minimize the trace generation time. In particular:

- If *method*(2) < 3, **CMSSL_method_nop** is used.
- If 3 ≤ *method*(2) ≤ 999, the method is chosen as follows:

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- If *operation* is **CMSSL_comm_by_get** and the maximum fan-out that any processing element will experience during the get is ≤ 2, **CMSSL_method_get_into_sends** is used. (For a definition of fan-out, see the description of **CMSSL_method_sort_and_scan**, below.)

- If *operation* is **CMSSL_comm_by_get** and the maximum fan-out that any processing element will experience during the get is > 2, **CMSSL_method_sort_and_scan** is used.

- If *operation* is **CMSSL_comm_by_send** or **CMSSL_comm_by_send_overwrite**, **CMSSL_method_nop** is used and collisions are stripped. (For details about collision stripping, see the Description section.)

- If *operation* is **CMSSL_comm_by_send_with_op**, **CMSSL_method_serial_combining** is used.

- If *method*(2) > 999, **CMSSL_method_nop** is used.

Note that these choices are likely to change in future releases. They may not yield the best results in all cases. It is recommended that you experiment with other methods to find the best one for your application.

method: Must be an integer front-end array of length ≥ 2. The second element must be your estimate of how many times the trace will be used.

Usage: Valid in any **comm_setup** call.

CMSSL_method_fastgraph

This method is currently unavailable on the CM-5. If you select this method, **CMSSL_method_nop** is substituted.

CMSSL_method_get_into_sends

Compiles a gathering pattern to be used with **comm_get** into a series of send operations.

Copyright © 1994 Thinking Machines Corporation

method: May be either an integer front-end array of length ≥ 1 or a scalar integer.

Usage: Valid only in **comm_setup** calls in which the operation is **CMSSL_comm_by_get**.

CMSSL_method_nop

Stores the ***p*** and {***x***|***y***}***_mask*** values with no additional processing. A subsequent **comm_get**, **comm_send**, or **comm_send_***combiner* call, supplied with the *trace* value returned by this method, will compute the trace and will run at about the same speed as the corresponding CM Fortran utility library routine. This method is useful for comparing the performance of operations with and without compiled traces, with minimal code changes.

method: May be either an integer front-end array of length ≥ 1 or a scalar integer.

Usage: Valid in any **comm_setup** call.

CMSSL_method_serial_combining

Treats the destination array as if it had an additional local axis. Elements that are sending to identical destinations (that is, elements with identical ***p*** entries) are directed to different locations along this extra axis. When the trace is used, elements are combined along the local axis.

method: May be either an integer front-end array of length ≥ 1 or a scalar integer.

Usage: Valid only in **comm_setup** calls in which the operation is **CMSSL_comm_by_send_with_op**.

CMSSL_method_sort_and_scan

Performs a sort to collect together elements that are to be combined. This method performs particularly well when there is a large variation in the fan-out or fan-in of the graph, or when the maximum fan-in or fan-out is large.

Fan-out applies only to get operations. The fan-out for a source location is the number of destination elements to

Copyright © 1994 Thinking Machines Corporation

which that location must send its data. A large variation in the fan-out means that some source locations are sending to few (or one) destination elements, while other source locations are sending to many destination elements.

Fan-in applies only to send operations. The fan-in for a destination location is the number of source elements being sent to it. A large variation in the fan-in means that some destination locations are receiving few (or one) source elements, while other destination locations are receiving many source elements.

method: May be either an integer front-end array of length ≥ 1 or a scalar integer.

Usage: Valid in **comm_setup** calls in which the operation is **CMSSL_comm_by_get** or **CMSSL_comm_by_send_with_op**.

option Scalar integer variable. In the current CM-5 release, only the value **CMSSL_comm_verbose** is valid; any other option results in an error message. The **CMSSL_comm_verbose** option determines whether the communication compiler will print any messages associated with returned *ier* codes. An associated *value* (see below) of 0 specifies that messages should not be printed; a non-zero *value* specifies that they should be printed. (Default: 0.)

value Variable associated with *option*. See *option* description. Must be an integer.

ier Scalar integer variable. Error code; set to 0 on successful return.

The following errors are returned by **comm_setup**:

−1 Invalid operation. The *operation* argument is not one of the allowed operations.

−2 Invalid method. The *method* argument is not allowed with the operation selected or is an unknown method.

Copyright © 1994 Thinking Machines Corporation

−4 The scattering coordinates supplied in the ***p*** argument contain collisions and the *operation* is **CMSSL_comm_by_send**. This message is only a warning; the computed trace is valid.

−8 The ***p*** array has an illegal type, shape, or layout.

The following errors are returned by **comm_get**, **comm_send**, **comm_send_add**, **comm_send_and**, **comm_send_max**, **comm_send_min**, **comm_send_or**, and **comm_send_xor**:

−1 Invalid data type. The ***x*** and/or ***y*** data type is invalid for this operation. For instance, complex data cannot be passed to **comm_send_max**. Only data types that are allowed in the combiner operation to be performed are allowed in the **comm_send_***combiner* routines.

−2 Invalid trace type. The *trace* value passed to this routine was of the wrong type for this operation. For instance, calling **comm_send_add** with a trace compiled with *operation* = **CMSSL_comm_by_get** will cause this error.

−4 Bad or unrecognized trace. The *trace* valuepassed to this routine was not usable. Possible reasons: the trace was never assigned the return value of a successful **comm_setup** call, the trace has already been deallocated, or the memory used by the trace has been corrupted.

The following errors are returned by **comm_set_option**:

−1 Invalid (unrecognized) *option* name.

−2 Invalid option value. The *value* supplied is not valid for the *option* selected.

DESCRIPTION

The communication compiler consists of the following routines:

comm_setup Computes a message delivery optimization, or *trace*, for the specified operation using the source and destination

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

shapes and layouts you have supplied. This trace can only be used on the current partition size. The setup routine allocates processing element and partition manager memory for storing the trace and related information. It returns an integer, *trace*, that contains a pointer to the partition manager memory where information about the trace (including pointers to processsing element fields) is stored. You must supply the value of *trace* in subsequent calls to the data motion and combining routines, and also to **deallocate_comm_setup** (when you want to deallocate the memory associated with the trace).

comm_get where (***y_mask***) ***y*** = ***x***(***p***)

Gathers selected source array elements into a destination array. Only those destination array elements for which ***y_mask*** was true in the setup call are overwritten. Each destination array location receives at most one source element. All data types are supported.

comm_send where (***x_mask***) ***y***(***p***) = ***x***

Scatters selected source array elements to a destination array. Only those source array elements for which ***x_mask*** was true in the setup call are moved. Each source array element is sent to at most one destination location. Source array elements overwrite the destination array elements to which they are sent. All data types are supported.

comm_send_*combiner* Scatters selected source array elements to a destination array. Only those source array elements for which ***x_mask*** was true in the setup call are moved. Source array elements are combined with the destination array elements to which they are sent. Colliding source elements are combined together with the destination element. The result overwrites the original destination element. The following combining operations are supported:

comm_send_add where (***x_mask***) ***y***(***p*** | +) = ***x***

Performs addition. Operates on integer, real, or complex data.

comm_send_and where (***x_mask***) ***y***(***p*** | **.and.**) = ***x***

Copyright © 1994 Thinking Machines Corporation

Performs a logical **AND** operation. Operates on logical or integer data. For integers, performs the operation on a bitwise basis.

comm_send_max where $(x_mask)\ y(p \mid \mathbf{max}) = x$

Selects the maximum value. Operates on integer and real data.

comm_send_min where $(x_mask)\ y(p \mid \mathbf{min}) = x$

Selects the minimum value. Operates on integer and real data.

comm_send_or where $(x_mask)\ y(p \mid \mathbf{.or.}) = x$

Performs a logical inclusive **OR** operation. Operates on logical or integer data. For integers, performs the operation on a bitwise basis.

comm_send_xor where $(x_mask)\ y(p \mid \mathbf{.xor.}) = x$

Performs a logical exclusive **OR** operation. Operates on logical or integer data. For integers, performs the operation on a bitwise basis.

comm_set_option Selects an option that prints error information. On the CM-200, this routine also selects options that can help optimize trace compilation and message delivery. The optimization options are not available in this CM-5 release, as they affect only those compilations performed using the FastGraph method (which is also unavailable in this release on the CM-5).

deallocate_comm_setup Deallocates the memory that **comm_setup** allocated to store a trace. When you call **deallocate_comm_setup**, you must supply the *trace* value returned by **comm_setup**. Once you have deallocated the memory associated with a trace, the trace can no longer be used.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Usage. The trace computed by the setup routine can be used for any operation that satisfies the following conditions:

- The type of operation must match the *operation* value you specified in the setup call. (Note that the setup routine uses the same *operation* value, **CMSSL_comm_by_send_with_op**, for the **comm_send_add**, **comm_send_and**, **comm_send_max**, **comm_send_min**, **comm_send_or**, and **comm_send_xor** operations, as these operations require the same setup action. A trace computed for any of these operations can be used for any of the others, assuming the other two conditions are met. Also, the *operation* values **CMSSL_comm_by_send** and **CMSSL_comm_by_send_overwrite** both represent the **comm_send** operation, but request different behavior with respect to collisions, as described in the argument list above.)

- The source and destination arrays must have the same rank, axis extents, and layout directives as in the setup call.

- The source and destination arrays must have the same data type, and that type must be valid for the operation to be performed.

You can follow a setup call with multiple calls to the data motion and combining routines, as long as the above conditions are met. The arguments you supplied in the setup call determine how data is moved in the associated data motion and combining routine calls. You can also have more than one trace allocated at a time; the only limit on the number of concurrently active traces is the amount of memory.

To compute and use a trace, follow these steps:

- Call **comm_setup** to allocate memory and compute the trace.

- Call the desired data motion or combining routine. (You can repeat this step an arbitrary number of times.)

- Call **deallocate_comm_setup**.

If you want to set a compilation option, call the **comm_set_option** routine prior to calling the setup routine.

Collision Handling. In a **comm_send** operation, two or more source elements are said to *collide* if they are sent to the same destination array location. The communication compiler handles collisions as follows:

Copyright © 1994 Thinking Machines Corporation

- If you specify *method*(1) = **CMSSL_method_automatic** and 3 $\leq$ *method*(2) $\leq$ 999, collisions are stripped. The communication compiler arbitrarily chooses one of the colliding source elements, and sends only that element to the destination element.
- In all other cases, all of the colliding source elements are sent to the destination processing element, which arbitrarily chooses one of them to overwrite the destination element.

If you specify *operation* = **CMSSL_comm_by_send**, the setup routine returns *ier* = −4 if it encounters collisions. The computed trace is valid. If you specify **CMSSL_comm_by_send_overwrite**, the setup routine does not issue the warning.

NOTES

Header File. The communication compiler routines use predefined symbolic constants. Therefore, you must include the statement `INCLUDE '/usr/include/cm/cmssl-cmf.h'` at the top of the main file of any program that uses these routines. This file defines symbolic constants and declares the type of the CMSSL functions.

Compilation Methods. The methods available for computing traces attempt to optimize communication, but each method involves a trade-off between compilation time and message delivery performance gain. (That is, some methods require more time for trace compilation but yield faster communication when the operation is performed; others require less compilation time but yield slower communication during message delivery.) Also, these techniques incur a cost in memory usage. Therefore, it is recommended that you experiment with different methods to find the one that works best for the operation you want to perform. Refer to the descriptions of the methods in the argument list, above.

Trace Deallocation. You can call **comm_setup** multiple times without calling **deallocate_comm_setup**. However, to conserve memory, it is good practice to deallocate a trace as soon as you are finished with it.

The trace is essentially a pointer to a region of partition manager memory; this region of memory contains pointers to additional regions of memory allocated on both the partition manager and the processing elements. If you destroy this pointer before deallocating the trace, there is no way for you to deallocate the trace subsequently. Therefore, be careful not to overwrite the trace pointer (for exam-

Copyright © 1994 Thinking Machines Corporation

ple, by allowing **comm_setup** to overwrite an allocated trace value, or by changing the value of the *trace* variable manually) unless you have deallocated the trace.

EXAMPLES

Sample CM Fortran code that uses the communication compiler can be found on-line in the subdirectory

```
comm-compiler/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.15 Vector Move (Extract and Deposit)

The **vector_move** routine moves a vector from each subgrid of a source array to a vector in a corresponding subgrid of a destination array. The source and destination arrays must have the same rank, data type, and processing element layout. The **vector_move_utils** routine returns processing element layout and subgrid shape information for any CM array.

Details about these routines are provided in the man page that follows.

Copyright © 1994 Thinking Machines Corporation

Vector Move (Extract and Deposit)

The **vector_move** routine moves a vector from each subgrid of a source array to a vector in a corresponding subgrid of a destination array. The source and destination arrays must have the same rank, data type, and processing element layout. The **vector_move_utils** routine returns processing element layout and subgrid shape information for any CM array.

SYNTAX

vector_move (***y, x,*** *y_coords, y_axis, x_coords, x_axis, ier*)

vector_move_utils (***array,*** *subgrid_extents, num_procs, ier*)

ARGUMENTS

y	One-based CM array of any data type.
x	One-based CM array of the same rank and data type as ***y***. The ***x*** and ***y*** arrays must have the same processing element layout; otherwise, the code −2 is returned in *ier*. *Processing element* refers to a slicewise processing element on a CM-200 or CM-2; a processing node on a CM-5 without vector units; or a vector unit on a CM-5 with vector units. *Processing element layout* refers to the number of processing elements along each array axis. The extent of axis *x_axis* in ***x*** must equal the extent of axis *y_axis* in ***y***.
y_coords	One-based integer front-end array of rank 1 and extent equal to the rank of ***y***. Defines the destination vectors to which the source vectors will be moved. The destination vectors are formed by holding the local coordinate along each axis *i* of ***y*** constant at the value *y_coords*(*i*), except the coordinate corresponding to axis *y_axis*. The value of *y_coords*(*y_axis*) is ignored. The array *y_coords* must satisfy $1 \leq y_coords(i) \leq$ subgrid length of axis *i* of ***y***, for each axis of ***y***.
y_axis	Scalar integer variable. Identifies the axis in ***y*** along which the destination vector lies. Must satisfy $1 \leq y_axis \leq$ rank(***y***).

Copyright © 1994 Thinking Machines Corporation

x_coords One-based integer front-end array of rank 1 and extent equal to the rank of ***x***. Defines the source vectors to be moved to ***y***. The source vectors are formed by holding the local coordinate along each axis *i* of ***x*** constant at the value *x_coords*(*i*), except the coordinate corresponding to axis *x_axis*. The value of *x_coords*(*x_axis*) is ignored. The array *x_coords* must satisfy $1 \leq$ *x_coords*(*i*) $\leq$ subgrid length of axis *i* of ***x***, for each axis of ***x***.

x_axis Scalar integer variable. Identifies the axis in ***x*** along which the source vector lies. Must satisfy $1 \leq$ *x_axis* $\leq$ rank(***x***).

array CM array whose processing element layout and subgrid shape is required.

subgrid_extents Integer front-end array of rank 1 and extent equal to the rank of ***array***. On successful return, *subgrid_extents*(*i*) contains the subgrid extent of ***array*** along axis *i*.

num_procs Integer front-end array of rank 1 and extent equal to the rank of ***array***. On successful return, *num_procs*(*i*) contains the number of processing elements along axis *i* of ***array***.

ier Scalar integer variable. Error code. Set to 0 on successful return, or to one of the following error codes:

−1 ***x*** and ***y*** do not have the same rank.

−2 ***x*** and ***y*** do not have the same processing element layout.

−4 *y_coords* does not satisfy $1 \leq$ *y_coords*(*i*) $\leq$ subgrid length of axis *i* of ***y***, for all axes of ***y***, or *x_coords* does not satisfy $1 \leq$ *x_coords*(*i*) $\leq$ subgrid length of axis *i* of ***x***, for all axes of ***x***.

DESCRIPTION

The **vector_move** routine moves a vector from each subgrid of ***x*** to a vector in a corresponding subgrid of ***y***. Each element in the destination vector is overwritten by the corresponding source value. The operation occurring within each subgrid can be expressed by

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

$$\boldsymbol{y}(y_coords(1), ..., y_coords(y_axis-1), i, y_coords(y_axis+1), ...) = \boldsymbol{x}(x_coords(1), ..., x_coords(x_axis-1), i, x_coords(x_axis+1), ...)$$
$(1 \leq i \leq$ extent of axis *y_axis*$)$.

The **vector_move_utils** routine returns processing element layout and subgrid shape information for any CM array. The *subgrid_extents* argument returns the subgrid extent of the array along each axis; the *num_procs* argument returns the number of processing elements along each axis. In this context, *processing element* refers to a slicewise processing element on a CM-200 or CM-2; a processing node on a CM-5 without vector units; or a vector unit on a CM-5 with vector units.

The on-line examples show how to use the **vector_move** and **vector_move_utils** routines to extract and deposit rows and columns of a two-dimensional array, and how to use **vector_move** to extract a face from a three-dimensional array.

Consider a 64 x 64 source array ***x*** and a 64 x 16 destination array ***y***. Further assume a 4 x 8 processing element layout underlying both ***x*** and ***y***. The subgrid of ***x*** on each processing element is 16 x 8; the subgrid of ***y*** on each processing element is 16 x 2. The following CM Fortran expression extracts the third column of the subgrid of ***x*** on each processing element and deposits it in the second column of the subgrid of ***y***:

```
y(:,2:16:2) = x(:,3:64:8)
```

The same functionality can be achieved by calling **vector_move** with the following arguments:

```
integer y_coords(2), x_coords(2)
integer y_axis, x_axis

y_coords(2) = 2
x_coords(2) = 3
y_axis = 1
x_axis = 1
vector_move(y, x, y_coords, y_axis, x_coords, x_axis, ier)
```

NOTES

One-Based Arrays. The arrays ***x***, ***y***, *x_coords*, and *y_coords* must be one-based.

Processing Element Layout. Be sure that the processing element layout is identical for ***x*** and ***y***. An error is signalled if this is not the case. (*Processing element layout* refers to the number of processing elements along each array axis.) The on-line

Copyright © 1994 Thinking Machines Corporation

examples illustrate the use of the CM Fortran detailed layout directives in this context. You can also obtain the processing element layout and subgrid shape using **vector_move_utils**.

EXAMPLES

Sample CM Fortran code that uses the extract and deposit routines can be found on-line in the subdirectory

```
vector-move/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.16 Computation of Block Cyclic Permutations

The **compute_fe_block_cyclic_perms** routine computes the permutations required to transform any matrix from normal (elementwise consecutive) order to block cyclic order, and vice versa. Many CMSSL routines use **compute_fe_block_cyclic_perms** internally because they operate on matrices in block cyclic order to achieve improved performance.

NOTE

The **compute_fe_block_cyclic_perms** routine only *computes* block cyclic permutations; it does not *apply* any permutations to the supplied matrices. To apply the block cyclic permutations, use the **permute_cm_matrix_axis_from_fe** routine, described in Section 15.17.

The *LU* and *QR* routines described in Chapter 5 are examples of CMSSL routines that operate on matrices in block cyclic order. As a result, the *L* and *U* factors produced by the **gen_lu_get_l** and **gen_lu_get_u** routines, and the *R* factor produced by the **gen_qr_get_r** routine, are in block cyclic order. The **compute_fe_block_cyclic_perms** routine is useful in conjunction with the *LU* and *QR* routines if you want to obtain the *L* and *U* factors or the *R* factor in elementwise consecutive order. An example of this use of **compute_fe_block_cyclic_perms** is given in Section 15.16.2.

The **compute_fe_block_cyclic_perms** routine supports multiple instances in the sense that you supply it with one or more matrices embedded in a CM array. The matrices must all have the same dimensions; therefore, they all have the same block cyclic permutations, and **compute_fe_block_cyclic_perms** computes only one set of permutations for all the instances. The "multiple-instance" format of the routine is provided for compatibility with the *QR* and *LU* operations, and with the **permute_cm_matrix_axis_from_fe** routine, which applies permutations to multiple matrices at a time.

Copyright © 1994 Thinking Machines Corporation

15.16.1 Blocking, Load Balancing, and Block Cyclic Ordering

One way to understand block cyclic ordering is to consider the CMSSL routines that use blocking and load balancing (which is achieved through cyclic ordering) to achieve improved performance. Examples of such routines are

- The *LU* (Gaussian elimination) routines (described in Chapter 5).
- The *QR* routines (described in the Chapter 5).
- The routines that perform Householder reduction of Hermitian matrices to real symmetric tridiagonal form and corresponding basis transformation (described in Chapter 8).

Blocking

In the strategy called *blocking*, routines operate on and transfer blocks of data rather than single data elements. Each block resides in the memory associated with a parallel processing element. Blocking results in fewer vector-vector operations and more matrix-vector (level 2 BLAS) operations, which, when local to a processing element, can yield very high performance.

When you call a routine that uses blocking, you supply a *blocking factor* (usually denoted by *nblock* in the argument list) that determines the block size. For example, if you call the *LU* factorization routine, **gen_lu_factor**, and supply a block size of *b*, then in its Gaussian elimination process, **gen_lu_factor** eliminates *b* variables at a time by subtracting multiples of *b* equations from all later equations. For more information about blocking, see reference 6 in Section 15.21.

Load Balancing

In an elimination operation that does not use load balancing, the elimination runs through the columns of the matrix in order. As a result, the number of active processing elements decreases as the factorization proceeds. For example, if A is an (m x n) matrix distributed across an array of processing elements by assigning a contiguous p x q chunk of A to each processing element, then each processing element contains p consecutive rows and q consecutive columns of A. Thus, the first column of processing elements contains the first q columns of A. After q elimination steps, all the columns of A in the first column of processing elements will have been pro-

Copyright © 1994 Thinking Machines Corporation

cessed. Thus, after q steps, an entire column of processing elements becomes inactive. Similarly, after p steps, the first row of processing elements becomes inactive.

Suppose instead that the columns are processed in cyclic order (column 1 of the first column of processing elements, then column 1 of the second column of processing elements, and so on). Then as columns are eliminated, the active subgrid on each processing element shrinks, but no processing element becomes completely inactive until the last column is eliminated from some column of processing elements. This strategy, called *cyclic ordering*, achieves *load balancing*: that is, it increases the number of processing elements that are active at any one time. For more information about cyclic ordering, see reference 6 in Section 15.21.

Blocking and Load Balancing Combined: Block Cyclic Ordering

When blocking and load balancing are combined, a routine computes on a block of columns or rows as a unit. For example, rather than updating a matrix a single column at a time, a routine performs a block update of a number of columns. That is, the load balancing scheme processes blocks, rather than single columns, in cyclic order. In fact, some CMSSL routines eliminate columns *and* rows in block cyclic order.

For example, if you choose the block size to be b, the *QR* or *LU* factorization routine eliminates columns and rows in the following order:

- Columns 1 through b of the first column of processing elements, along with rows 1 through b of the first row of processing elements.
- Columns 1 through b of the second column of processing elements, along with rows 1 through b of the second row of processing elements; and so on.

After eliminating b columns from each column of processing elements, the routine returns to columns $b + 1$ through $2b$ of processing element column 1, and so on.

The permutations computed by the **compute_fe_block_cyclic_perms** routine are those required to transform the rows and columns of a given matrix from normal (elementwise consecutive) order to block cyclic order, and vice versa. For example, the permutation that transforms the columns from elementwise consecutive order to block cyclic order (with a blocking factor of b) does the following:

Copyright © 1994 Thinking Machines Corporation

- Moves the first b columns of the original matrix into columns 1 through b of the first column of processing elements.
- Moves the next b columns of the original matrix into columns 1 through b of the second column of processing elements; and so on until the first b columns of each column of processing elements have been filled.
- Moves the next b columns of the original matrix to columns $b + 1$ through $2b$ of processing element column 1.
- Continues this pattern until all columns of the original matrix have been moved.

This description is approximate; the exact details involve operations on quantities of data that are not exact multiples of b rows and columns.

Choosing the Blocking Factor

As mentioned above, the CMSSL routines that operate on matrices in block cyclic order require you to supply the blocking factor in the *nblock* argument. The optimum block size depends on the routine you are calling and the details of your application. Specific requirements for *nblock* are documented in the man pages for the various routines.

The blocking factor you supply when you call **compute_fe_block_cyclic_perms** depends on the context of the call. If you are using **compute_fe_block_cyclic_perms** to "undo" the block cyclic ordering of an L, U, or R factor, supply the same blocking factor you supplied in the original LU or QR factorization call.

15.16.2 Obtaining *L*, *U*, and *R* Factors in Elementwise Consecutive Order

Because the LU and QR routines operate in block cyclic order, the L, U, and R factors produced by the **gen_lu_get_l**, **gen_lu_get_u**, and **gen_qr_get_r** routines, respectively, are stored in block cyclic order. The **compute_fe_block_cyclic_perms** routine is useful if you want to permute L, U, or R to "undo" the block cyclic ordering. To obtain a factor in elementwise consecutive order, you must follow these steps:

Copyright © 1994 Thinking Machines Corporation

1. Before factoring the matrices embedded in a CM array ***A***, compute their block cyclic permutations by supplying ***A*** to **compute_fe_block_cyclic_perms**.

2. Use **permute_cm_matrix_axis_from_fe** to permute the columns and rows of the matrices embedded in ***A*** to block cyclic order.

3. Use **gen_lu_factor** or **gen_qr_factor** to factor the matrices in ***A***. (The factorization routine works in block cyclic order, but since you already permuted the matrices to block cyclic order in Step 1, the factorization actually occurs in elementwise consecutive order.)

4. Use **permute_cm_matrix_axis_from_fe** to permute the columns and rows of the factors from block cyclic order back to elementwise consecutive order.

For the *LU* routines, the block cyclic permutations and the exact form of the *L* and *U* factors returned by **gen_lu_get_l** and **gen_lu_get_u** depend on whether you have specified pivoting. Similarly, for the *QR* routines, the exact form of the *R* factor returned by **gen_qr_get_r** depends on whether you specified pre- or post-permutation in the *back_solve_strategy* argument. For exact definitions of the factors, refer to the sections on the *LU* and *QR* routines in Chapter 5.

For an example of how to use the strategy outlined above, refer to the online sample code (see the man page at the end of this section for the pathname).

Copyright © 1994 Thinking Machines Corporation

Computation of Block Cyclic Permutations

Given a block size and a CM array containing one or more embedded matrices, the **compute_fe_block_cyclic_perms** routine computes the permutations required to transform the rows and columns of the matrices from elementwise consecutive order to block cyclic order, and vice versa.

SYNTAX

compute_fe_block_cyclic_perms (***A***, *n1, n2, row_axis, col_axis, nblock, f, f_inv, g, g_inv, ier*)

ARGUMENTS

A	CM array of any data type and rank greater than or equal to 2, containing one or more instances of a matrix for which you want to compute the block cyclic permutations. Only the rank, axis extents, and layout directives of this array are used; its contents are ignored.
n1	Scalar integer variable. The number of rows in each matrix embedded in ***A***. Must be $\geq$ *n2*.
n2	Scalar integer variable. The number of columns in each matrix embedded in ***A***. Must be $\leq$ *n1*.
row_axis	Scalar integer variable. Identifies the axis that counts the rows of each matrix embedded in ***A***. Axis *row_axis* must have length $\geq$ *n1*. (For definitions of row and column axes, see Chapter 1.)
col_axis	Scalar integer variable. Identifies the axis that counts the columns of each matrix embedded in ***A***. Axis *col_axis* must have length $\geq$ *n2*. (For definitions of row and column axes, see Chapter 1.)
nblock	Scalar integer variable. The blocking factor upon which the block cyclic permutations are to be based.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

f Front-end integer array of rank 1 and length $\geq n1$. Upon return, the first *n1* elements contain the permutation that transforms the rows of each matrix from elementwise consecutive order to block cyclic order. Note: this permutation has no effect on the element ordering within a given row.

f_inv Front-end integer array of rank 1 and length $\geq n1$. Upon return, the first *n1* elements contain the permutation that transforms the rows of each matrix from block cyclic order to elementwise consecutive order. Note: this permutation has no effect on the element ordering within a given row.

g Front-end integer array of rank 1 and length $\geq n2$. Upon return, the first *n2* elements contain the permutation that transforms the columns of each matrix from elementwise consecutive order to block cyclic order. Note: this permutation has no effect on the element ordering within a given column.

g_inv Front-end integer array of rank 1 and length $\geq n2$. Upon return, the first *n2* elements contain the permutation that transforms the columns of each matrix from block cyclic order to elementwise consecutive order. Note: this permutation has no effect on the element ordering within a given column.

ier Scalar integer variable. Set to 0 upon successful completion. The **compute_fe_block_cyclic_perms** routine can fail under any of the following conditions:

−1	The ***A*** you supplied is not a CM array.
−2	The rank of ***A*** is < 2.
−3	*row_axis* has length < *n1*.
−4	*col_axis* has length < *n2*.
−20	One of the permutation arrays has length ≤ 0.
−30	*row_axis* or *col_axis* is ≤ 0 or > rank(***A***), or *row_axis* = *col_axis*.
−40	*nblock* is less than or equal to 0.

Copyright © 1994 Thinking Machines Corporation

DESCRIPTION

Given a block size and a CM array ***A*** containing one or more embedded matrices, the **compute_fe_block_cyclic_perms** routine computes the permutations required to transform the rows and columns of the matrices from elementwise consecutive order to block cyclic order, and from block cyclic order to elementwise consecutive order.

The permutations *f* and *f_inv* are called *row permutations*. Each row moves as a whole unit; the elements within each column are reordered. The arrays *f* and *f_inv* must have length at least *n1*.

The permutations *g* and *g_inv* are called *column permutations*. Each column moves as a whole unit; the elements within each row are reordered. The arrays *g* and *g_inv* must have length at least *n2*.

The values in the front-end arrays indicate the *destinations* of the corresponding row or column elements, and assume an offset of 0. For example, the row permutation *into* block cyclic order is given by the mapping

$$A(i, :) \longrightarrow A(f(i)+1, :)$$

while the column permutation *out of* block cyclic order is given by the mapping

$$A(:, i) \longrightarrow A(:, g_inv(i)+1)$$

For a detailed description of block cyclic order, refer to Section 15.16.1.

EXAMPLES

Sample CM Fortran code that uses the **compute_fe_block_cyclic_perms** routine can be found on-line in the subdirectory

```
block-cyclic/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.17 Permutation Along an Axis

The **permute_cm_matrix_axis_from_fe** routine permutes the rows or columns of one or more matrices, using a permutation that is supplied in a front-end array.

The man page on the next page describes this routine and its arguments in detail, and defines row and column permutations. Figure 56 illustrates a row permutation where

$n_p = 4, \quad n_q = 6, \quad f = [\,1 \;\; 3 \;\; 0 \;\; 2]$.

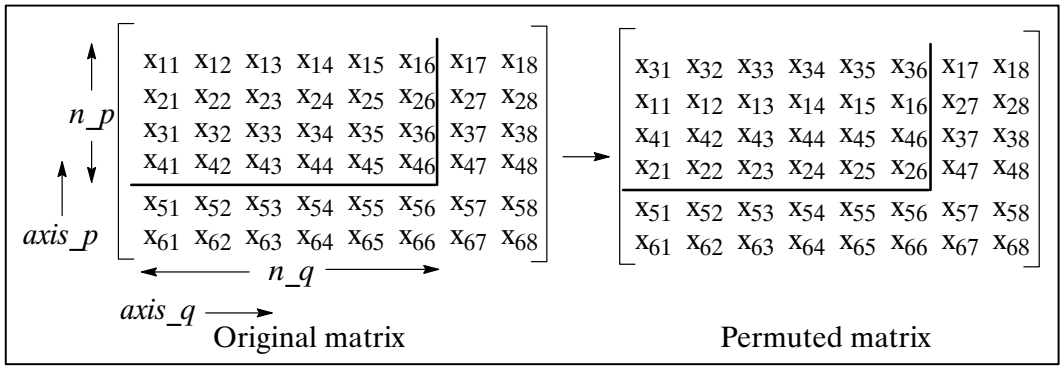

Figure 56. A row permutation.

Figure 57 illustrates a column permutation where

$n_p = 6, \quad n_q = 4, \quad f = [\,2 \;\; 4 \;\; 1 \;\; 5 \;\; 0 \;\; 3]$.

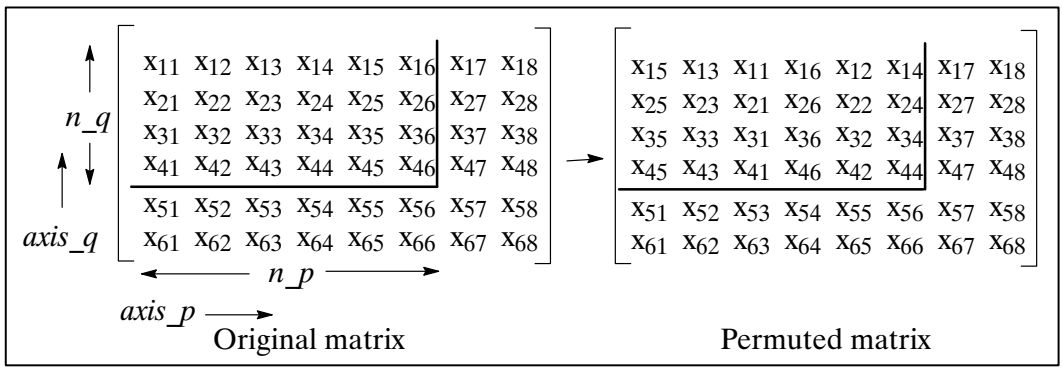

Figure 57. A column permutation.

Copyright © 1994 Thinking Machines Corporation

Permutation Along an Axis

Given a CM array containing one or more embedded matrices, a front-end array containing a permutation, and a choice of row or column axis, the **permute_cm_matrix_axis_from_fe** routine permutes each matrix along the specified axis.

SYNTAX

permute_cm_matrix_axis_from_fe (*A, n_p, n_q, axis_p, axis_q, f, ier*)

ARGUMENTS

A	CM array of any data type and rank greater than or equal to 2, containing one or more embedded matrices whose rows or columns you want to permute.
n_p	Scalar integer variable. The active length of axis *axis_p*. Also the length of the permutation to be applied to axis *axis_p*.
n_q	Scalar integer variable. The active length of axis *axis_q*.
axis_p	Scalar integer variable. Identifies the axis along which elements are to be permuted. This axis counts either the rows or the columns of the matrices embedded in *A*.
axis_q	Scalar integer variable. Identifies the matrix axis other than *axis_p*.
f	Integer front-end array of rank 1 and length $\geq$ *n_p*. The first *n_p* elements of *f* must contain the new locations to which the first *n_p* elements along *axis_p* of each matrix are to be mapped, assuming an offset of 0. That is, if $f(i) = k$, then along axis *axis_p* of each matrix, element i is mapped to element $k+1$.
ier	Scalar integer variable. Set to 0 upon successful completion. The **permute_cm_matrix_axis_from_fe** routine can fail under any of the following conditions: -1 The *A* you have supplied is not a CM array. -2 The rank of *A* is < 2.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

−3	*axis_p* has length < *n_p*.
−4	*axis_q* has length < *n_q*.
−20	The *f* array has length ≤ 0.
−30	*axis_p* or *axis_q* is ≤ 0 or > rank(***A***), or *axis_p* = *axis_q*.

DESCRIPTION

Given a CM array ***A*** containing one or more embedded matrices, a front-end array *f* containing a permutation, and a choice of row or column axis (*axis_p*), the **permute_cm_matrix_axis_from_fe** routine permutes each matrix along *axis_p*.

If axis *axis_p* counts the rows, then the permutation is applied to the first *n_p* elements of each of the first *n_q* columns. This operation is called a *row permutation* because the first *n_q* elements of each row (up to row *n_p*, inclusive) move together as a whole unit.

If axis *axis_p* counts the columns, then the permutation is applied to the first *n_p* elements of each of the first *n_q* rows. This operation is called a *column permutation* because the first *n_q* elements of each column (up to column *n_p*, inclusive) move together as a whole unit.

The operation performed by this routine can be summarized as follows:

$A(,,,i,,,j,,,,,) = A(,,,f(i)+1,,,j,,,,,);$
$1 \le i \le n1$ along *axis_p*
$1 \le j \le n2$ along *axis_q*

A matrix element is not used in or affected by the permutation if its CM Fortran subscript in the *axis_p* dimension is greater than *n_p*, or if its CM Fortran subscript in the *axis_q* dimension is greater than *n_q*.

If *f* is the permutation *f* or *f_inv* computed by the **compute_fe_block_cyclic_perms** routine, then to transform the rows into or out of block cyclic order, you must set *axis_p*, *axis_q*, *n_p* and *n_q* to the same values as *row_axis*, *col_axis*, *n1*, and *n2*, respectively, in the **compute_fe_block_cyclic_perms** call.

Similarly, if *f* is the permutation *g* or *g_inv* computed by the **compute_fe_block_cyclic_perms** routine, then to transform the columns into or out of block cyclic order, you must set *axis_p*, *axis_q*, *n_p*, and *n_q* to the same values as *col_axis*, *row_axis*, *n2*, and *n1*, respectively, in the **compute_fe_block_cyclic_perms** call.

Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the **permute_cm_matrix_axis_from_fe** routine can be found on-line in the subdirectory

```
block-cyclic/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

15.18 Combination of Permutations

Given two source front-end arrays containing permutations, the **combine_fe_perms** routine generates a destination front-end array containing the combination of the permutations.

Details are provided in the man page that follows.

Copyright © 1994 Thinking Machines Corporation

Combination of Permutations

Given two source front-end arrays containing permutations, the **combine_fe_perms** routine generates a destination front-end array containing the combination of the permutations.

SYNTAX

combine_fe_perms (*n2, n1, dest, src2, src1*)

ARGUMENTS

n2	Scalar integer variable. The length of the permutation contained in *src2*.
n1	Scalar integer variable. The length of the permutation contained in *src1*.
dest	Integer front-end array of rank 1 and length greater than or equal to the maximum of *n1* and *n2*. Contains the permutation $P = P_2P_1$, where P_2 is the permutation contained in *src2* and P_1 is the permutation contained in *src1*. If P_2 and P_1 are not the same size, the smaller permutation is extended to the larger size by embedding it as the first elements of the identity permutation of the larger size.
src2	Integer front-end array of rank 1 and length greater than or equal to *n2*. Must contain a permutation P_2 of *n2* coordinates; see the Description section for details.
src1	Integer front-end array of rank 1 and length greater than or equal to *n1*. Must contain a permutation P_1 of *n1* coordinates; see the Description section for details..

DESCRIPTION

Given a one-dimensional front-end array *src1* containing a permutation P_1 of length *n1*, and a one-dimensional front-end array *src2* containing a permutation P_2 of length *n2*, the **combine_fe_perms** routine generates the one-dimensional front-end array *dest* containing the permutation P_2P_1. If *n1* is not equal to *n2*, the smaller

Copyright © 1994 Thinking Machines Corporation

permutation is extended to the larger size by embedding it as the first elements of the identity permutation of the larger size.

The permutations are assumed to be 0-based in the following sense. If $P_1(i) = k$, then it is assumed that P_1 maps i to $k+1$. The same is true for P_2 and P. Furthermore, given $P_1(i) = k$, the combination $P(i) = P_2(k+1)$. For example, if

$P_1(1) = 8$ (P_1 maps 1 to 9)
$P_2(9) = 4$ (P_2 maps 9 to 5)

then $P(1) = 4$ and P maps 1 to 5.

EXAMPLES

Sample CM Fortran code that uses the **combine_fe_perms** routine can be found on-line in the subdirectory

```
block-cyclic/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.19 Zeroing Routine

The **zero_elements** routine zeroes a CM array. This routine provides faster performance than the equivalent CM Fortran code, especially for single-precision real or complex data. The man page that follows provides the syntax.

Copyright © 1994 Thinking Machines Corporation

Zeroing Routine

Given a CM array, the routine described below returns the zeroed array.

SYNTAX

zero_elements (***A***, *ier*)

ARGUMENTS

A	CM array of any rank and any numeric data type. On return, all elements of ***A*** are set to zero.
ier	Scalar integer variable. Error code. On return, *ier* is set to 0 unless the CMSSL safety mechanism is on and the descriptor for ***A*** is invalid, in which case *ier* is set to −32.

DESCRIPTION

Given the CM array ***A***, the **zero_elements** routine sets all elements of ***A*** to zero.

NOTES

This routine provides faster performance than the equivalent CM Fortran code, especially for single-precision real or complex data.

EXAMPLES

Sample CM Fortran code that uses the zeroing routine can be found on-line in the subdirectory

```
zeroing/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

15.20 Send-to-NEWS and NEWS-to-Send Reordering

On the CM-200, the **send_to_news** and **news_to_send** routines allow you to change the ordering of specified axes of a CM array from send to NEWS ordering or vice-versa.

On the CM-5, these routines have no effect, since NEWS and send ordering are the same. They are provided only for compatibility with the CM-200. (Refer to the CM Fortran documentation set for information about send and NEWS ordering.)

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Send-to-NEWS and NEWS-to-Send Reordering

On the CM-200, given a CM array of any rank or type, the **send_to_news** and **news_to_send** routines change the ordering of specified axes of the array by shuffling the data in place. On the CM-5, these routines have no effect because send and NEWS ordering are the same. They are provided only for compatibility with the CM-200.

SYNTAX

send_to_news (*A, **B**, xform_vector*)

news_to_send (*A, **B**, xform_vector*)

ARGUMENTS

A	Integer front-end array of rank 1 and length **CMSSL_DESC_LENGTH** (a symbolic constant defined in the include file **`cmssl-cmf.h`**).
B	CM array of any rank and type. Each ***B*** axis to be reordered must be a power of 2 in length.
xform_vector	Logical front-end vector of length equal to the rank of ***B***, indicating which axes of ***B*** are to be reordered.

DESCRIPTION

On the CM-200, the **send_to_news** and **news_to_send** routines reorder the CM array ***B*** in place and, by creating a CM array descriptor in *A*, allow you to use the reordered ***B*** array in a subroutine.

On the CM-5, these routines have no effect because send and NEWS ordering are the same.

Copyright © 1994 Thinking Machines Corporation

NOTES

Header File Required. Be sure to include the header file **`cmssl-cmf.h`** in any program that uses the **send_to_news** and **news_to_send** routines. This header file defines the symbolic constant **CMSSL_DESC_LENGTH**.

Copyright © 1994 Thinking Machines Corporation

15.21 References

For more information on the all-to-all rotation, see the following references:

1. Brunet, J.-Ph., and S. L. Johnsson. All-to-All Broadcast and Applications on the Connection Machine. *Int. J. Sup. App.* **6**, no. 3 (1992): 241−56.

2. Brunet, J.-Ph., A. Edelman, and J. P. Mesirov. An Optimal Hypercube Direct n-Body Solver on the Connection Machine. In *Supercomputing 90*, ICS Press, 1990. Pp. 748−52.

3. Johnsson, S. L., and C.-T. Ho. Spanning Graphs for Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Trans. Computers* **38**, no. 9 (1989): 1249−68.

4. Mathur, K. K. and S. L Johnsson. *All-to-All Communication on the Connection Machine CM-200*. Thinking Machines Corporation Technical Report TR-243, 1992.

5. Brunet, J.-Ph., A. Edelman, and J. P. Mesirov. Hypercube Algorithms for Direct *N*-Body Solvers for Different Granularities. To be published in *SIAM J. Sci. Stat. Comput.*

For information about block cyclic ordering, refer to

6. Lichtenstein, W. and S. L. Johnsson. Block Cyclic Dense Linear Algebra. *SIAM J. Sci. Comp.* **14**, no. 6 (1993): 1257−86.

For further information about the FastGraph trace compilation method of the communication compiler, see the following reference:

7. Dahl, E. D. Mapping and Compiled Communication on the Connection Machine System. *Proceedings of the Fifth Distributed Memory Computing Conference*, *IEEE*, DMCC 1990, Charleston, South Carolina, April 8−10, 1990. Pp. 756−66.

For information about the partitioning algorithm implemented on the CM, see the following references:

8. Pothen, A., H. D. Simon, and K.−P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications* **11** (1990): 430−52.

Copyright © 1994 Thinking Machines Corporation

9. Simon, H. D. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering* **2** (1991): 135−48.

10. Johan, Z., K. K. Mathur, S. L. Johnsson, and T. J. R. Hughes. *An efficient communication strategy for finite element methods on the Connection Machine CM-5 system*. Thinking Machines Corporation Technical Report TR-256, 1993.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation