Chapter 4

Sparse Matrix Operations

This chapter describes the CM Fortran interface to the basic linear algebra operations for sparse matrices. One section is devoted to each of the following topics:

- introduction
- arbitrary elementwise sparse matrix operations
- arbitrary block sparse matrix operations
- grid sparse matrix operations
- references

4.1 Introduction

CMSSL provides routines for basic linear algebra operations on sparse matrices representing structured and unstructured grids. Both elementwise and block sparse matrices are supported. The following operations are provided for arbitrary elementwise sparse matrices, arbitrary block sparse matrices, and grid sparse matrices:

- sparse matrix x vector
- vector x sparse matrix
- sparse matrix x dense matrix
- dense matrix x sparse matrix

Copyright © 1994 Thinking Machines Corporation

NOTE

The sparse matrix routines described in this chapter are intended for general sparse matrices and (in the case of the grid sparse matrix operations) for certain banded sparse matrices. For banded sparse matrices that cannot be handled by the grid sparse matrix operations, you can improve performance significantly by writing your own multiplication routine that exploits the band structure. More routines for banded sparse matrices are planned for future CMSSL releases.

4.1.1 Arbitrary Sparse Matrix Operations

The primary intent of the arbitrary sparse matrix operations is to provide the basic building blocks for more complex sparse applications — for example, a sparse iterative solver, or computation of the eigenvalues of the sparse matrices by the Lanczos or Arnoldi method.

For applications that do not perform explicit sparse linear algebra operations, but want to make use of some communication primitives used by the sparse basic linear algebra functions, CMSSL provides gather and scatter routines, as well as partitioned gather and scatter routines and an associated partitioning routine. These routines, which are described in Chapter 15, are intended for use in applications such as the solution of partial differential equations on unstructured discretizations, and optimization problems represented by sparse matrices occurring in network flow problems. A communication compiler is also provided.

Storage Representations

Two separate storage representations of the sparse matrix are supported (see references 2 and 3 listed in Section 4.5). These data mappings are referred to as the elementwise sparse matrix mapping and the block sparse matrix mapping. In the elementwise data mapping, the zero data values of

Copyright © 1994 Thinking Machines Corporation

the matrix are ignored and the non-zero data values are stored row-wise. In the block sparse mapping, the sparse matrix is stored as a collection of dense block matrices. In its full matrix representation, this block matrix storage scheme is extremely flexible. The dense blocks need not be composed of contiguous rows and columns, and may overlap in any way. One possible application for the block sparse representation is the finite element method. Structured finite element grids lead to a grid block sparse data layout; unstructured grids result in an arbitrary block sparse layout. The two storage schemes are described in more detail in Sections 4.2 and 4.3.

Gathering and Scattering

The CMSSL sparse matrix operations can be described briefly in three steps (see reference 1). First, the source vector (or matrix) elements are "gathered" into local vectors. The relevant local operation (matrix vector or matrix matrix multiplication) is then performed. Finally, the results of the local operations are "scattered" back to the destination vector (or matrix). If there is collision at the destination, the colliding data values are added. (Note that in this context, "local" means local to a block, and does not refer to the lower-level implementation or to processing elements.) Examples illustrating the gathering and scattering processes are provided in Sections 4.2 and 4.3.

Optimization Switches

The arbitrary sparse matrix functions described in this chapter provide two optimization switches. These optimizations are based on the premise that applications will use these sparse functions repeatedly. A marginal setup cost can therefore be incurred before the first call to the sparse functions. The setup cost is amortized over several calls to the sparse matrix functions.

The first optimization switch allows you to choose the communication algorithm used in the gather and scatter phases of the operation. This switch determines what kind of preprocessing the setup routine performs, how much time the preprocessing takes, and how much the performance of the multiplication routine improves as a result. Three algorithms are available (see references 4 and 5):

- One that uses get, send, and scan-add operations (in the arbitrary elementwise sparse routines) or get and send-add operations (in the block sparse routines)

Copyright © 1994 Thinking Machines Corporation

- One that uses the partitioned gather and scatter routines (**part_gather** and **part_scatter**), described in Chapter 15.
- One that uses the sparse gather and scatter routines (**sparse_util_gather** and **sparse_util_scatter**), described in Chapter 15.

Guidelines for choosing the communication algorithm are included in the descriptions of the routines.

The setup phase requires additional processing element storage. The amount of storage required is a strong function of the sparsity of the matrix, and is determined at run time by the setup routine. We strongly recommend that you free this additional storage as soon as the application is finished with the sparse matrix routines. The deallocation routines are described in this chapter.

The second optimization feature provided by the sparse matrix operations is the ability to permute the vector elements, either randomly (see references 4, 6, 7, and 8) or, in the case of the elementwise routines, based on the partitioning routine, which is described in Chapter 15 and which uses recursive spectral bisection (see reference 10). Permutations of the vector elements are particularly useful in reducing the routing conflicts that occur, and can reduce the time for data motion significantly. The setup routines compute the permutations and return the relevant masks and the locations of the vector elements after the permutations. Most applications use the sparse matrix vector products to produce vectors that are used in other operations such as inner products and local arithmetic. With the proper use of the masks, these other operations are invariant to the location of the vector elements. Thus, the products need not be permuted back in the inner loop of an application. We strongly recommend that applications use this optimization switch.

Optimizing Array Layout

As with most other CMSSL operations, the performance of the sparse matrix operations is a very strong function of the compiler layout directives used by the application. In particular, the block sparse functions perform significantly better when each dense block composing the sparse matrix is local to (contained within) a processing element. You can achieve this result by using the detailed axis descriptors of the CM Fortran **CMF$LAYOUT** directive.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

4.1.2 Grid Sparse Matrix Operations

The grid sparse matrix routines operate on data from grid-based applications. Coefficient matrix elements residing at each grid point *P* are multiplied by vector or matrix elements residing at point *P* and its nearest-neighbor points. The result is placed in product vector or matrix elements residing at point *P*. These routines support multiple instances and block matrices.

Like the arbitrary sparse matrix routines, the grid sparse routines are designed with the assumption that the application will use these functions repeatedly. A marginal setup cost can therefore be incurred before the first call to the functions. The setup cost is then amortized over several calls.

Copyright © 1994 Thinking Machines Corporation

4.2 Arbitrary Elementwise Sparse Matrix Operations

This section introduces the arbitrary elementwise sparse matrix operations. For detailed information about the routines and their arguments, refer to the man page at the end of this section.

4.2.1 The Arbitrary Elementwise Sparse Matrix Routines

Given a sparse matrix and a vector or dense matrix, the arbitrary elementwise sparse matrix routines compute the product of the sparse matrix with the vector or dense matrix. The following routines are provided:

sparse_matvec_mult	Multiplies a sparse matrix by a vector.
sparse_vecmat_mult	Multiplies a vector by a sparse matrix.
sparse_mat_gen_mat_mult	Multiplies a sparse matrix by a dense matrix.
gen_mat_sparse_mat_mult	Multiplies a dense matrix by a sparse matrix.

The two routines in which the sparse matrix is the left-hand operand (**sparse_matvec_mult** and **sparse_mat_gen_mat_mult**) use the following setup and deallocation routines:

sparse_matvec_setup
deallocate_sparse_matvec_setup

The two routines in which the sparse matrix is the right-hand operand (**sparse_vecmat_mult** and **gen_mat_sparse_mat_mult**) use the following setup and deallocation routines:

sparse_vecmat_setup
deallocate_sparse_vecmat_setup

For information about setup and deallocation, refer to the Description section of the man page following this section.

Copyright © 1994 Thinking Machines Corporation

4.2.2 Storage of Sparse Matrices

Before calling the arbitrary elementwise sparse matrix routines, you must create a vector (one-dimensional CM array) ***A*** to represent the sparse matrix. You must also supply associated vectors, ***rows***, ***cols***, ***row_segments***, and ***A_mask***. Refer to the man page following this section for descriptions of these arguments. The following example is based on the argument definitions in the man page.

Example

The sparse matrix

$$\begin{bmatrix} 1 & 0 & 4 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 \\ 5 & 0 & 1 & 0 \end{bmatrix}$$

is represented by the vector

A = [1 4 2 3 5 1]

along with associated vectors

rows = [1 1 2 3 4 4]

cols = [1 3 2 4 1 3]

row_segments = [T F T T T F]

In this case, since you need not mask any elements of ***A***, you can supply the scalar logical value **.true.** for the ***A_mask*** argument.

If you defined ***A*** to have extent 10, that is,

A = [1 4 2 3 5 1 0 0 0 0]

then the corresponding vectors would be

rows = [1 1 2 3 4 4 0 0 0 0]

cols = [1 3 2 4 1 3 0 0 0 0]

Copyright © 1994 Thinking Machines Corporation

row_segments = [T F T T T F F F F F]

A_mask = [T T T T T T F F F F]

Although other representations are possible, the current implementation requires that the elements of each row be contiguous with one another.

For discussions of this common method of storing sparse matrices, see references 2 and 3 in the list in Section 4.5.

4.2.3 Communication Algorithms

The **sparse_matvec_setup** and **sparse_vecmat_setup** routines always preprocess the communication that will be required by the multiplication routines. The *motion* argument allows you to choose the communication algorithm used, as follows:

- If you supply *motion* = 0, the communication uses get, send, and scan-add operations.
- If you supply *motion* = 1, the communication uses the partitioned gather and scatter routines (**part_gather** and **part_scatter**), which are described in Chapter 15.
- If you supply *motion* = 2, the communication uses the sparse gather and scatter routines (**sparse_util_gather** and **sparse_util_scatter**), which are described in Chapter 15.

With *motion* = 0, the setup routine uses the least amount of preprocessing time and has the smallest effect on the performance of the multiplication routine. In contrast, with *motion* = 1, the setup routine uses the most preprocessing time but yields the highest performance improvement in the multiplication phase. If you are calling the multiplication routine a small number of times, you may do best to choose *motion* = 0; if you are performing many multiplications with the same sparsity, you will probably achieve the best performance improvement with *motion* = 1. For further guidelines, see the man page at the end of this section.

The setup routine allocates the extra processing element storage needed for the communication. The amount of storage needed is a strong function of the sparsity of the matrix. You specify the sparsity of the matrix in the call to the setup routine; if the sparsity changes, you must call the setup routine

Copyright © 1994 Thinking Machines Corporation

again. To free the extra storage, you must call **deallocate_sparse_matvec_setup** or **deallocate_sparse_vecmat_setup** after all of the sparse matrix calls associated with one setup call have finished.

4.2.4 Permutation of Source and Destination Array Element Locations

Along with the sparse matrix ***A***, you must supply the arbitrary elementwise sparse matrix routines with a source array, ***x***, containing one or more vectors or dense matrices to be multiplied by the sparse matrix; and a destination array, ***y***, containing corresponding vectors or dense matrices into which the results of the multiplication are to be placed. The source and destination arrays may be the same variable. They have rank 1 if you are calling **sparse_matvec_mult** or **sparse_vecmat_mult**, or rank 2 if you are calling **sparse_mat_gen_mat_mult** or **gen_mat_sparse_mat_mult**. When you call one of the setup routines, the contents of ***x*** and ***y*** are ignored; only the geometry is examined.

The **sparse_matvec_setup** or **sparse_matvec_setup** argument *mapping*, if set to 1 or 2, activates an option that uses either an internal random permutation generator (if *mapping* = 1) or the partitioning routine described in Chapter 15 (if *mapping* = 2) to return permutations of the source and destination array element locations. (The value *mapping* = 2 requires that the sparsity of the matrix be symmetric. Symmetric sparsity is defined in the man page.) The permutations of the source and destination arrays affect all subsequent multiplication calls associated with the setup call, as follows:

- Before calling the multiplication routines, you must permute the elements of ***x*** using the source array permutation returned by the setup routine.
- The multiplication routine permutes the elements of ***y*** using the destination array permutation returned by the setup routine. Thus, the product array is returned in permuted form.

Note that the *source* array permutation must be applied by your application, while the *destination* array permutation is applied by the multiplication routine.

This feature involves a marginal preprocessing cost, but is extremely useful for minimizing the routing conflicts that occur during the data motion phase of the multiplication. In some cases, the permutations can reduce the com-

Copyright © 1994 Thinking Machines Corporation

munication time and thus improve performance significantly. If you set *mapping* to 0, an identity permutation is returned for both arrays.

The setup routine returns the source and destination array permutations in the integer arrays ***where_is_x*** and ***where_is_y***, respectively. If the source and destination arrays have rank 2, each permutation moves elements within columns only; each location remains in its original column. If the source and destination arrays are the same variable, the same permutation is returned in ***where_is_x*** and ***where_is_y***. If you set *mapping* to 0, you may conserve memory by declaring ***where_is_x*** and ***where_is_y*** as scalar integers with the value 0.

Along with the source and destination arrays, you must supply the routines with two integer arguments, *x_length* and *y_length*, containing the true extents of the first axes of ***x*** and ***y***, respectively. That is, *x_length* contains the number of active elements (for rank 1) or rows (for rank 2) in ***x***, and *y_length* contains the number of active elements (for rank 1) or rows (for rank 2) in ***y***. In the permuted source array that you provide to the multiplication routine, it is possible that the active elements will no longer be confined to the first *x_length* locations (or rows, for rank 2).

If your source or destination array contains elements that must be masked, you must supply the setup routine with a corresponding logical array, ***x_mask*** or ***y_mask***, that has the same axis extents and layout directives as ***x*** or ***y,*** respectively. The setup routine ignores the initial contents of the masks. On return from the setup, the values of the masks reflect the permutations returned in ***where_is_x*** and ***where_is_y***. If the source or destination array requires no masking, you may provide the scalar logical value **.true.** for ***x_mask*** or ***y_mask***, respectively. (An example is provided below.)

Copyright © 1994 Thinking Machines Corporation

NOTE

Product elements resulting from the multiplication are sent to the permuted ***y*** locations; thus, the ***y*** returned by each multiplication routine is the permuted destination array. Optionally, you may use the information in ***y_mask*** and ***where_is_y*** to permute the elements of ***y*** back to their original positions after the multiplication occurs. However, most applications do not require you to do this.

For detailed definitions of the returned values of ***x_mask***, ***y_mask***, ***where_is_x***, and ***where_is_y***, refer to the man page at the end of this section. The example below is based on the argument definitions in the man page.

For a discussion of random permutation of source vector element locations, see references 4, 6, 7, and 8 listed in Section 4.5. For a discussion of the partitioning routine, see reference 10.

Example

Suppose you want to multiply the sparse matrix

$$\begin{bmatrix} 1 & 0 & 4 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 \\ 5 & 0 & 1 & 0 \end{bmatrix}$$

by a vector of length 4. In this example, ***x*** is one-dimensional with declared extent 6, and *x_length* is 4:

$$\boldsymbol{x} = [x_1 \ x_2 \ x_3 \ x_4 \ - \ -]$$

Copyright © 1994 Thinking Machines Corporation

(The symbol − indicates a masked data element.) If you set *mapping* = 0, you can supply the scalar value 0 for ***where_is_x*** and ***where_is_y***; you need not permute your source array before supplying it to the multiplication routine; and the multiplication routine does not permute the destination elements.

However, suppose you set *mapping* to 1 or 2, and **sparse_matvec_setup** assigns ***where_is_x*** the values

where_is_x = [6 1 3 2 4 5]

and ***x_mask*** the values

x_mask = [T T T F F T].

In this case, you must permute the source array element locations as follows:

$\boldsymbol{x} = [x_2 \; x_4 \; x_3 \; - \; - \; x_1]$

That is, you must use this template when permuting the elements of each ***x*** you supply in subsequent **sparse_matvec_mult** calls associated with this setup.

In this same example, suppose ***y*** has declared extent 4 and true extent *y_length* = 4. In this case, since there is no need for a destination mask, you can supply the single scalar value ***y_mask*** = **.true.**. If **sparse_matvec_setup** assigned ***where_is_y*** the values

where_is_y = [2 1 4 3],

then the destination array is

$\boldsymbol{y} = [2x_2 \quad x_1 + 4x_3 \quad 5x_1 + x_3 \quad 3x_4]$.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Arbitrary Elementwise Sparse Matrix Operations

Given a sparse matrix and a vector or dense matrix, the routines described below compute the product of the sparse matrix with the vector or dense matrix.

SYNTAX

sparse_matvec_setup (***A_mask, row_segments, rows, cols, y, x, y_mask, x_mask, where_is_y, where_is_x***, *y_length, x_length, mapping, motion, setup, ier*)

sparse_matvec_mult (***y, A, x, rows, cols, row_segments, y_mask, A_mask, x_mask,*** *motion, setup, ier*)

sparse_mat_gen_mat_mult (***y, A, x, rows, cols, row_segments, y_mask, A_mask, x_mask,*** *motion, setup, ier*)

deallocate_sparse_matvec_setup (*setup, motion*)

sparse_vecmat_setup (***A_mask, row_segments, rows, cols, y, x, y_mask, x_mask, where_is_y, where_is_x***, *y_length, x_length, mapping, motion, setup, ier*)

sparse_vecmat_mult (***y, A, x, rows, cols, row_segments, y_mask, A_mask, x_mask,*** *motion, setup, ier*)

gen_mat_sparse_mat_mult (***y, A, x, rows, cols, row_segments, y_mask, A_mask, x_mask,*** *motion, setup, ier*)

deallocate_sparse_vecmat_setup (*setup, motion*)

ARGUMENTS

y CM array of the same rank as ***x*** and the same data type (real or complex) as ***A***. May be the same variable as ***x***. The contents of this array are ignored by the setup routines. Upon return from one of the multiplication routines, contains the product of the sparse matrix and ***x***.

Copyright © 1994 Thinking Machines Corporation

A Real or complex CM array of rank 1 containing — in packed storage — the non-zero elements of the sparse matrix. The elements of each row must be contiguous with one another. The extent of ***A*** may be larger than the number of non-zero elements in the sparse matrix.

A_mask If ***A*** contains elements that need masking, ***A_mask*** must be a logical CM array of rank 1 with the same extent and layout directives as ***A***; it is used as a mask for ***A***. Set an element of ***A_mask*** to **.true.** if the corresponding element of ***A*** is to be treated as a non-zero element of the sparse matrix. Supply values for ***A_mask*** before calling the setup routine; then use the same ***A_mask*** when calling the associated multiplication routine.

If ***A*** does not contain elements that need masking, you can conserve processing element memory by supplying the scalar logical value **.true.** for ***A_mask***.

row_segments Logical CM array of rank 1 with the same extent and layout directives as ***A***. Contains information about the sparsity of the matrix. Set an element of ***row_segments*** to **.true.** if and only if the corresponding element of ***A*** is the first non-zero element in a row of the sparse matrix. Supply values for ***row_segments*** before calling the setup routine. The setup routine does not alter the values of ***row_segments***. You must supply the same ***row_segments*** values when calling the associated multiplication routine; do not modify ***row_segments*** between the setup call and the associated multiplication call.

rows Integer, *one-based* CM array of rank 1 with the same extent and layout directives as ***A***. When you call the setup routine, each element of ***rows*** must contain the row number, in the sparse matrix, of the corresponding element of ***A***. Do not modify ***rows*** after the setup routine returns; you must supply the multiplication routine with the values contained in ***rows*** upon return from the associated setup routine.

Copyright © 1994 Thinking Machines Corporation

cols Integer, *one-based* CM array of rank 1 with the same extent and layout directives as ***A***. When you call the setup routine, each element of ***cols*** must contain the column number, in the sparse matrix, of the corresponding element of ***A***. Do not modify ***cols*** after the setup routine returns; you must supply the multiplication routine with the values contained in ***cols*** upon return from the associated setup routine.

x CM array of rank 1 or 2 and of the same data type and precision as ***A***. May be the same variable as ***y***. The contents of this array are ignored by the setup routines. Before calling the multiplication routine, you must apply to ***x*** the permutation, if any, indicated by the values that the setup routine assigned to ***where_is_x***.

y_mask If ***y*** contains elements that need masking, ***y_mask*** must be a logical CM array with the same rank, axis extents, and layout directives as ***y***. May be the same variable as ***x_mask***. Used as a mask for the destination array. The setup routine ignores the initial contents. On return from the setup routine, ***y_mask*** has the following values:

- If *mapping* = 0 and ***y*** has rank 1, ***y_mask*** (1:*y_length*) = **.true.**; all other elements of ***y_mask*** are **.false.**.
- If *mapping* = 0 and ***y*** has rank 2, then within each column of ***y_mask***, ***y_mask***(1:*y_length, l*) = **.true.** and all other elements of the column are **.false.**.
- If *mapping* = 1 or 2 and ***y*** has rank 1, then ***y_mask***(***where_is_y*** (1:*y_length*)) = **.true.**; all other elements of ***y_mask*** are **.false.**.
- If *mapping* = 1 or 2 and ***y*** has rank 2, then within each column of ***y_mask***, ***y_mask***(***where_is_y*** (*k*,*l*), *l*) = **.true.** for *k* = 1:*y_length*; all other elements of the column are **.false.**.

Do not modify ***y_mask*** between the setup call and the associated multiplication call(s). When you call one of the multiplication routines, you must supply the values assigned to ***y_mask*** by the setup routine.

Copyright © 1994 Thinking Machines Corporation

If ***y*** does not contain elements that need masking, you can conserve processing element memory by supplying the scalar logical value **.true.** for ***y_mask***.

x_mask If ***x*** contains elements that need masking, ***x_mask*** must be a logical CM array with the same rank, axis extents, and layout directives as ***x***. May be the same variable as ***y_mask***. Used as a mask for the source array. The setup routine ignores the initial contents. On return from the setup routine, ***x_mask*** helps you identify the active elements of ***x*** (after the permutation has been applied, if *mapping* = 1 or 2). In particular, ***x_mask*** has the following values:

- If *mapping* = 0 and ***x*** has rank 1, ***x_mask***(1:*x_length*) = **.true.** and all other elements of ***x_mask*** are **.false.**.
- If *mapping* = 0 and ***x*** has rank 2, then within each column of ***x_mask***, ***x_mask***(1:*x_length*, *j*) = **.true.**; all other elements of the column are **.false.**.
- If *mapping* = 1 or 2 and ***x*** has rank 1, then ***x_mask***(***where_is_x*** (1:*x_length*)) = **.true.**; all other elements of ***x_mask*** are **.false.**.
- If *mapping* = 1 or 2 and ***x*** has rank 2, then within each column of ***x_mask***, ***x_mask***(***where_is_x***(*k*, *l*), *l*)) = **.true.** for *k* = 1:*x_length*; all other elements of the column are **.false.**.

Do not modify ***x_mask*** between the setup call and the associated multiplication call(s). When you call one of the multiplication routines, you must supply the values assigned to ***x_mask*** by the setup routine.

If ***x*** does not contain elements that need masking, you can conserve processing element memory by supplying the scalar logical value **.true.** for ***x_mask***.

where_is_y If you set *mapping* = 0, you can conserve processing element memory by supplying the scalar integer value 0 for ***where_is_y***.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

If you set *mapping* = 1 or 2, ***where_is_y*** must be an integer, one-based CM array with the same rank, axis extents, and layout directives as ***y***. May be the same variable as ***where_is_x***. The initial contents are ignored. On return from the setup routine, ***where_is_y*** has the following values:

- If *mapping* = 0, ***where_is_y***(*k*) (for rank 1) or ***where_is_y***(*k*,*l*) (for rank 2) is simply *k*.
- If *mapping* = 1 or 2 and ***y*** has rank 1, ***where_is_y***(*k*) is the location to which the *k*th destination array location will be mapped by the multiplication routine.
- If *mapping* = 1 or 2 and ***y*** has rank 2, ***where_is_y***(*k*,*l*) is the row number to which location (*k, l*) of the destination array will be mapped by the multiplication routine.

where_is_x If you set *mapping* = 0, you can conserve processing element memory by supplying the scalar integer value 0 for ***where_is_x***.

If you set *mapping* = 1 or 2, ***where_is_x*** must be an integer, one-based CM array with the same rank, axis extents, and layout directives as ***x***. May be the same variable as ***where_is_y***. The initial contents are ignored. On return from the setup routine, ***where_is_x*** has the following values:

- If *mapping* = 0, ***where_is_x***(*k*) (for rank 1) or ***where_is_x***(*k*, *l*) (for rank 2) is simply *k*.
- If *mapping* = 1 or 2 and ***x*** has rank 1, ***where_is_x***(*k*) is the location to which the *k*th source array location must be mapped.
- If *mapping* = 1 or 2 and ***x*** has rank 2, ***where_is_x***(*k*,*l*) is the row number to which location (*k, l*) of the source array must be mapped.

y_length Scalar integer variable. The true extent of the first axis of ***y***.

x_length Scalar integer variable. The true extent of the first axis of ***x***.

Copyright © 1994 Thinking Machines Corporation

mapping Scalar integer variable. Must be 0, 1, or 2. If you set *mapping* to 1, the setup routine returns in ***where_is_x*** and ***where_is_y*** random permutations of the source and destination array element locations, respectively. If you set *mapping* to 2, the setup routine returns in ***where_is_x*** and ***where_is_y*** permutations generated internally using the **partition_mesh** routine. If you set *mapping* to 0, the setup routine returns identity permutations in ***where_is_x*** and ***where_is_y***. Note that *mapping* = 2 requires symmetric sparsity. This means that the locations of the non-zero elements of the sparse matrix must form a pattern that is symmetric across the diagonal (however, the contents of reflected elements need not be the same). The Notes section, below, includes guidelines for choosing a *mapping* value.

motion Scalar integer variable. Must be 0, 1, or 2. If you supply *motion* = 0, the communication required for the multiplication uses get, send, and scan-add operations. If you supply *motion* = 1, the communication uses the partitioned gather and scatter routines (**part_gather** and **part_scatter**). If you supply *motion* = 2, the communication uses the sparse gather and scatter routines (**sparse_util_gather** and **sparse_util_scatter**). The Notes section, below, includes guidelines for choosing a *motion* value.

The setup routine modifies the contents of *motion*. Do not modify *motion* after the setup routine returns; you must supply the associated multiplication and deallocation routines with the value contained in *motion* upon return from the setup routine.

setup Scalar integer variable. Internal variable. Do not modify *setup* after the setup routine returns; you must supply the associated multiplication and deallocation routines with the value contained in *setup* upon return from the setup routine.

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

0	Normal return.
−1	*mapping* is not equal to 0, 1, or 2; or *mapping* = 2 and the sparse matrix does not have symmetric sparsity.
−2	*motion* is not equal to 0, 1, or 2.

Copyright © 1994 Thinking Machines Corporation

−4 *x_length* is greater than the extent of the first axis of ***x***, or *y_length* is greater than the extent of the first axis of ***y***.

−8 ***x***, ***x_mask***, and ***where_is_x*** do not have the same shape, or ***y***, ***y_mask***, and ***where_is_y*** do not have the same shape.

−16 ***A_mask***, ***row_segments***, ***rows***, and ***cols*** do not have the same shape.

−64 *setup* is too large to fit in available memory.

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

0 Normal return.

−1 ***A***, ***x***, or ***y*** does not contain real or complex data.

DESCRIPTION

The arbitrary elementwise sparse matrix routines perform the operations listed below. (In the formulas below, x and y denote vectors while X and Y denote matrices. However, lowercase letters are used for both cases everywhere else in this text.)

sparse_matvec_mult	$y = Ax$	multiplies a sparse matrix by a vector
sparse_vecmat_mult	$y^{\mathrm{T}} = x^{\mathrm{T}}A$	multiplies a vector by a sparse matrix
sparse_mat_gen_mat_mult	$Y = AX$	multiplies a sparse matrix by a dense matrix
gen_mat_sparse_mat_mult	$Y^{\mathrm{T}} = X^{\mathrm{T}}A$	multiplies a dense matrix by a sparse matrix

The sparse matrix and the vector must be of the same data type (real or complex) and the same precision; the sparse matrix is stored in packed form (vector argument ***A***), as described in the argument list.

To multiply a sparse matrix by a vector or dense matrix, follow these steps:

1. Call **sparse_matvec_setup**.

Copyright © 1994 Thinking Machines Corporation

2. Call **sparse_matvec_mult** or **sparse_mat_gen_mat_mult**.

   To compute more than one product using sparse matrices that all have identical sparsities, follow one call to **sparse_matvec_setup** with multiple calls to **sparse_matvec_mult** or **sparse_mat_gen_mat_mult**. If the sparsity changes, start with Step 1 again.

3. After all **sparse_matvec_mult** or **sparse_mat_gen_mat_mult** calls associated with the same **sparse_matvec_setup** call, call **deallocate_sparse_matvec_setup** to deallocate the processing element storage space required by the setup routine.

To multiply a vector or dense matrix by a sparse matrix, follow these steps:

1. Call **sparse_vecmat_setup**.

2. Call **sparse_vecmat_mult** or **gen_mat_sparse_mat_mult**.

   To compute more than one product using sparse matrices that all have identical sparsities, follow one call to **sparse_vecmat_setup** with multiple calls to **sparse_vecmat_mult** or **gen_mat_sparse_mat_mult**. If the sparsity changes, start with Step 1 again.

3. After all **sparse_vecmat_mult** or **gen_mat_sparse_mat_mult** calls associated with the same **sparse_vecmat_setup** call, call **deallocate_sparse_vecmat_setup** to deallocate the processing element storage space required by the setup routine.

More than one setup may be active at a time. That is, you may call the setup routine more than once without calling the deallocation routine.

Setup Phase. The setup routine analyzes the sparsity of the matrix, allocates processing element storage space for the multiplication, and places appropriate values in variables required by the multiplication routines.

The setup routine provides two options that may improve performance significantly:

- The *motion* argument allows you to choose a communication algorithm. Your choice determines what kind of preprocessing the setup routine performs, how much time the preprocessing takes, and how much the performance of the multiplication routine improves as a result. See the Notes section for guidelines.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- If you set *mapping* = 1 or 2, the setup routine returns permutations of the source and destination array element locations in ***where_is_x*** and ***where_is_y***, respectively. This feature involves a marginal preprocessing cost, but is extremely useful for minimizing the routing conflicts that occur during the data motion phase of the multiplication. In some cases, the permutations can reduce the communication time and thus improve performance significantly. The value *mapping* = 2 requires that the sparse matrix have symmetric sparsity. Also, if the source and destination arrays are the same variable, the same permutation is applied to both arrays. You must apply the permutation indicated in ***where_is_x*** to the source arrays you supply in subsequent multiplication calls. The permutation indicated in ***where_is_y*** is applied to the destination array by the multiplication routine.

Multiplication Phase. Given a source CM array, ***x***, and a sparse matrix represented as a packed vector ***A***, each multiplication routine computes the product of the sparse matrix with ***x*** and returns the product in the CM array ***y***.

Deallocation Phase. The **deallocate_sparse_matvec_setup** and **deallocate_sparse_vecmat_setup** routines deallocate the memory that was allocated in a previous call to **sparse_matvec_setup** or **sparse_vecmat_setup**, respectively. Each setup call should be followed (after one or more associated calls to the multiplication routines) by a deallocation call.

NOTES

Argument Values. Do not alter the contents of *setup*, ***rows***, ***cols***, ***row_segments***, ***x_mask***, ***y_mask***, ***where_is_x***, ***where_is_y***, or *motion* between a call to the setup routine and a subsequent, associated call to a multiplication routine, for the following reasons:

- You must supply the multiplication routine with the values that the setup routine assigns to *setup*, ***rows***, ***cols***, ***row_segments***, and *motion*.
- You must supply the deallocation routine with the values that the setup routine assigns to *setup* and *motion*.
- If you set *mapping* to 1 or 2 when calling the setup routine, you must use the values that the setup routine assigns to ***x_mask*** and ***where_is_x*** to permute the elements of each ***x*** you supply in subsequent multiplication calls. (Refer to the on-line sample code for an example.) The values that the setup rou-

Copyright © 1994 Thinking Machines Corporation

tine assigns to ***y_mask*** and ***where_is_y*** determine the permutation that the multiplication routine will apply to the destination elements.

If the setup routine permutes the source array element locations (*mapping* = 1 or 2), it also alters the contents of ***rows*** (and of ***cols***, if *motion* = 0) appropriately to reflect the permutation so that the multiplication will occur correctly. (The multiplication routines use the contents of ***rows*** to perform the communication for the multiplication. If you supplied *motion* = 0 to the setup routine, the multiplication routines also use the information stored in ***cols***.)

The product array ***y*** is the only argument updated by a call to one of the multiplication routines.

Guidelines for Choosing *mapping* and *motion* Values. The *mapping* and *motion* arguments have the meanings indicated below.

Value of *mapping*	Permutation returned in ***where_is_x*** and ***where_is_y***
0	identity
1	random
2	based on recursive spectral bisection (partitioning)

Value of *motion*	Operations Used by Communication algorithm
0	get, send, scan-add
1	**part_gather**, **part_scatter**
2	**sparse_util_gather**, **sparse_util_scatter**

Use the following guidelines when choosing *mapping* and *motion* values:

- All combinations of *mapping* and *motion* values are valid.
- The value of *motion* determines how much time the preprocessing takes, and how much the performance of the multiplication routine improves as a result. With *motion* = 0, the setup routine uses the least amount of preprocessing time and has the smallest effect on the performance of the multiplication routine. With *motion* = 1, the setup routine uses the most preprocessing time but yields the highest performance improvement in the multiplication phase. If you are calling the multiplication routine a small number of times, you may do best to choose *motion* = 0. However, if you are performing many multiplications with the same sparsity, you will probably achieve the best performance improvement with *motion* = 1.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- If you are performing more than one multiplication operation with the same sparsity, *mapping* = 1 or 2 usually yields better performance (at a cost of preprocessing time and temporary storage) than *mapping* = 0.
- The value *mapping* = 2 requires symmetric sparsity. This means that the locations of the non-zero elements of the sparse matrix must form a pattern that is symmetric across the diagonal (however, the contents of reflected elements need not be the same).
- If the sparse matrix has symmetric sparsity, then *mapping* = 2 yields better performance than *mapping* = 0 or 1; and the best performance is achieved when *mapping* = 2 and *motion* = 1.
- The only way to take full advantage of *motion* = 1 is to set *mapping* = 2.
- Setting *motion* = 2 is best when there is little locality — for example, when the pointers are random (which is true when *mapping* = 1).

Overlapping Variables. For square matrices, you can use the same variable for ***x*** as for ***y***, the same variable for ***where_is_x*** as for ***where_is_y***, and the same variable for ***x_mask*** as for ***y_mask***.

Numerical Stability. The arbitrary elementwise sparse matrix operations are stable.

Numerical Complexity. If the vector ***A*** has length *n*, the sparse matrix vector and vector sparse matrix multiplication operations require approximately 2*n* floating-point operations if ***A*** is real, or approximately 8*n* floating-point operations if ***A*** is complex.

If the vector ***A*** has length *n* and ***x*** has *r* columns, the sparse matrix dense matrix and dense matrix sparse matrix operations require approximately 2*nr* floating-point operations if ***A*** is real, or 8*nr* floating-point operations if ***A*** is complex.

Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the arbitrary elementwise sparse matrix routines can be found on-line in the subdirectory

```
sparse-matrix-vector/cmf
```

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

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

4.3 Arbitrary Block Sparse Matrix Operations

This section introduces the arbitrary block sparse matrix operations. For detailed information about the routines and their arguments, refer to the man page at the end of this section.

4.3.1 The Arbitrary Block Sparse Matrix Routines

Given a block sparse matrix, a vector or dense matrix, and gathering and scattering pointer arrays, the arbitrary block sparse matrix routines compute the product of the block sparse matrix with the vector or dense matrix. The following routines are provided:

block_sparse_setup	Allocates processing element memory for the operation.
block_sparse_matrix_vector_mult	Multiplies a block sparse matrix by a vector.
vector_block_sparse_matrix_mult	Multiplies a vector by a block sparse matrix.
block_sparse_mat_gen_mat_mult	Multiplies a block sparse matrix by a dense matrix.
gen_mat_block_sparse_mat_mult	Multiplies a dense matrix by a block sparse matrix.
deallocate_block_sparse_setup	Deallocates memory allocated by **block_sparse_setup**.

For information about setup and deallocation, refer to the Description section of the man page following this section.

4.3.2 Block Representation, Gathering, and Scattering

Each block of data in a block sparse matrix is identified by a set of *m* row numbers and *n* column numbers. A block may overlap itself or other blocks. Blocks need not be contiguous, and the rows and columns within a block need not be contiguous.

Copyright © 1994 Thinking Machines Corporation

When you call the block sparse matrix routines, you must embed the blocks of the block sparse matrix in a three-dimensional CM array, ***A***, with declared extents ***A***(*dim_1*, *dim_2*, *dim_3*) and true extents ***A***(m, n, p). The first two axes represent the rows and columns of the blocks (which are assumed to be dense); the third axis counts the blocks. Thus, each of p blocks is represented by an m x n dense matrix within ***A***. Rows and columns must be preserved in this representation; that is, elements of a block that occur in the same row (or column) in the block sparse matrix must occur in the same row (or column) when embedded in ***A***.

The source array, ***x,*** and destination array, ***y,*** may be of rank 1 or 2 (with axes of any lengths), may be the same variable, and are assumed to be dense. The elements to be multiplied with each block of ***A*** are gathered from the source array, and the results of each block multiplication are scattered to form the destination array. You must supply the **block_sparse_setup** routine with two arrays, ***x_pointers*** and ***y_pointers***, containing pointers for gathering elements from the source array and scattering elements to the destination array, respectively.

The ***x_pointers*** and ***y_pointers*** arrays indicate the locations of the blocks within the block sparse matrix. The location of element ***A***(i, j, k) within the block sparse matrix is given by (***y_pointers***(i, k), ***x_pointers***(j, k)). (See Example 1 in Section 4.3.5.)

The elements of ***x_pointers*** identify the ***x*** elements that are to be multiplied by the blocks of ***A***; the elements of ***y_pointers*** identify the ***y*** locations to which the resulting product elements are to be scattered.

For **block_sparse_matrix_vector_mult**, the gather operation can be expressed in array notation as

$$\text{forall}(i = 1{:}n,\ j = 1{:}p)\ \ u(i, j)\ =\ \boldsymbol{x}(\boldsymbol{x\_pointers}(i, j))$$

and the scatter operation can be expressed as

$$\text{forall}(i = 1{:}m,\ j = 1{:}p)\ \boldsymbol{y}(\boldsymbol{y\_pointers}(i, j))\ =\ \boldsymbol{y}(\boldsymbol{y\_pointers}(i, j)) + v(i, j)$$

where $u(i,j)$ (for $i = 1{:}n, j = 1{:}p$) contains the source array elements to be multiplied with the jth block of ***A***, and $v(i,j)$ (for $i = 1{:}m, j = 1{:}p$) contains the resulting product elements to be scattered to the destination array. For **vector_block_sparse_matrix_mult**, the same definitions apply, but m and n are switched. For **block_sparse_mat_gen_mat_mult** and **gen_mat_block_sparse_mat_mult**, these definitions are extended by one dimension.

Copyright © 1994 Thinking Machines Corporation

Detailed definitions of ***x_pointers*** and ***y_pointers*** are provided in the man page. Section 4.3.5 presents examples of how gathering and scattering work in block sparse matrix vector multiplication and dense matrix block sparse matrix multiplication.

4.3.3 Communication Algorithms

The **block_sparse_setup** routine always preprocesses the communication that will be required by the multiplication routines. The *motion* argument allows you to choose the communication algorithm used, as follows:

- If you supply *motion* = 0, the communication uses get and send-add operations.
- If you supply *motion* = 1, the communication uses the partitioned gather and scatter routines (**part_gather** and **part_scatter**), which are described in Chapter 15.
- If you supply *motion* = 2, the communication uses the sparse gather and scatter routines (**sparse_util_gather** and **sparse_util_scatter**), which are described in Chapter 15.

With *motion* = 0, the setup routine uses the least amount of preprocessing time and has the smallest effect on the performance of the multiplication routine. In contrast, with *motion* = 1, the setup routine uses the most preprocessing time but yields the highest performance improvement in the multiplication phase. If you are calling the multiplication routine a small number of times, you may do best to choose *motion* = 0; if you are performing many multiplications with the same sparsity, you will probably achieve the best performance improvement with *motion* = 1. For further guidelines, see the man page at the end of this section.

The setup routine allocates the extra processing element storage needed for the communication. The amount of storage needed depends on the contents of the gathering pointers arrays, ***x_pointers***. You supply this array in the call to the setup routine; if the pointer values change, you must call the setup routine again. (The contents of ***y_pointers*** must also remain constant for all multiplication calls following a single **block_sparse_setup** call.) To free the extra storage, you must call **deallocate_block_sparse_setup** after all of the sparse matrix calls associated with one setup call have finished.

Copyright © 1994 Thinking Machines Corporation

4.3.4 Random Permutation of Source and Destination Array Element Locations

The **block_sparse_setup** argument *mapping*, if set to 1, activates an option that uses an internal random permutation generator to return permutations of the source and destination array element locations. These permutations affect all subsequent block sparse multiplication calls associated with the setup call, as follows:

- Before calling the multiplication routines, you must permute your source array elements, using the source array permutation returned by the setup routine.
- The multiplication routine permutes the elements of the destination array using the destination array permutation returned by the setup routine. Thus, the destination array is returned in permuted form.

Note that the *source* array permutation must be applied by your application, while the *destination* array permutation is applied by the multiplication routine.

This feature involves a marginal preprocessing cost, but is extremely useful for minimizing the routing conflicts that occur during the data motion phase of the multiplication. In some cases, the permutations can reduce the communication time and thus improve performance significantly. If you set *mapping* to 0, an identity permutation is returned for both arrays.

The setup routine returns the source and destination array permutations in the integer arrays ***where_is_x*** and ***where_is_y***, respectively. If the source and destination arrays have rank 2, each permutation moves elements within columns only; each location remains in its original column.

Along with the source and destination arrays, you must supply the block sparse matrix routines with two integer arguments, *x_length* and *y_length*, containing the true extents of the first axes of ***x*** and ***y***, respectively. That is, *x_length* contains the number of active elements (for rank 1) or rows (for rank 2) in ***x***, and *y_length* contains the number of active elements (for rank 1) or rows (for rank 2) in ***y***. In the permuted source array that you provide to the multiplication routine, it is possible that the active elements will no longer be confined to the first *x_length* locations (or rows, for rank 2).

When you call the setup routine, you must also supply two logical arrays, ***x_mask*** and ***y_mask***, that have the same axis extents and layout directives as ***x*** and ***y,*** respectively. The setup routine ignores the initial contents of the

Copyright © 1994 Thinking Machines Corporation

masks. On return from the setup, the values of the masks reflect the permutations returned in ***where_is_x*** and ***where_is_y***. (Examples are provided in Section 4.3.5.)

NOTE

Product elements from the block multiplications are scattered to the permuted ***y*** locations; thus, the ***y*** returned by each multiplication routine is the permuted destination array. Optionally, you may use the information in ***y_mask*** and ***where_is_y*** to permute the elements of ***y*** back to their original locations after the multiplication occurs. However, most applications do not require you to do this; for example, inner product computations on destination vectors are invariant under random permutation.

For detailed definitions of the returned values of ***x_mask***, ***y_mask***, ***where_is_x***, and ***where_is_y***, refer to the man page at the end of this section. The examples below are based on the argument definitions in the man page.

For a discussion of random permutation of source vector element locations, see references 4, 6, 7, and 8 listed in Section 4.5.

4.3.5 Examples

The following two examples show how gathering, scattering, and random permutation of the source and destination arrays work in

- block sparse matrix vector multiplication
- dense matrix block sparse matrix multiplication

These examples are based on the argument descriptions in the man page following this section. They use letters instead of numbers for array element values in some cases for clarity.

Copyright © 1994 Thinking Machines Corporation

Example 1: Block Sparse Matrix Vector Multiplication

In this example, $m = 5$, $n = 4$, and $p = 3$. The coefficient block sparse matrix contains three blocks, each of size (5 x 4). It is represented by the CM array ***A***, which has declared extents (10, 10, 3) and true extents (5, 4, 3). The first (5 x 4) elements of each dense matrix within ***A*** have the following values:

$$A(:,:,1) = \begin{bmatrix} a & f & k & p \\ b & g & l & q \\ c & h & m & r \\ d & i & n & s \\ e & j & o & t \end{bmatrix} \quad A(:,:,2) = \begin{bmatrix} u & z & e & j \\ v & a & f & k \\ w & b & g & l \\ x & c & h & m \\ y & d & i & n \end{bmatrix} \quad A(:,:,3) = \begin{bmatrix} o & t & y & j \\ p & u & z & k \\ q & v & a & l \\ r & w & b & m \\ s & x & c & n \end{bmatrix}$$

The ***x*** argument is a vector with declared extent 8 and true extent *x_length* = 6. The elements to be gathered from ***x*** are assumed to be, originally, in the first six locations of the vector:

$$\boldsymbol{x} = [\, x_1 \;\; x_2 \;\; x_3 \;\; x_4 \;\; x_5 \;\; x_6 \;\; - \;\; - \,]$$

(The symbol − indicates a masked data element.) In this example, when you call **block_sparse_setup**, you set *mapping* = 1. The setup routine permutes the source array element locations, assigns ***x_mask*** the values

$$\boldsymbol{x\_mask} = [\, \mathrm{T} \;\; \mathrm{F} \;\; \mathrm{T} \;\; \mathrm{F} \;\; \mathrm{T} \;\; \mathrm{T} \;\; \mathrm{T} \;\; \mathrm{T} \,]$$

and assigns ***where_is_x*** the values

$$\boldsymbol{where\_is\_x} = [\, 7 \;\; 3 \;\; 5 \;\; 6 \;\; 1 \;\; 8 \;\; 4 \;\; 2 \,]$$

These values indicate that the source array element locations must be permuted as follows:

$$\boldsymbol{x} = [\, x_5 \;\; - \;\; x_2 \;\; - \;\; x_3 \;\; x_4 \;\; x_1 \;\; x_6 \,]$$

That is, you must use this template when permuting the elements of each ***x*** you supply in subsequent multiplication calls associated with this setup call.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

The ***x_pointers*** array must have declared extents (10 x 3) and true extents (n x p) or (4 x 3). In this example, you supply the following values in the first (4 x 3) locations of ***x_pointers*** when you call **block_sparse_setup**:

$$\boldsymbol{x\_pointers} = \begin{bmatrix} 1 & 4 & 1 \\ 5 & 2 & 1 \\ 2 & 6 & 3 \\ 4 & 3 & 2 \end{bmatrix}$$

The ***x_pointers*** array determines the contents of the vectors of length $n = 4$ that will be multiplied on the left by the blocks of $\boldsymbol{A}$. Element (i, j) of ***x_pointers*** contains the original location of the ***x*** element that is to be multiplied by the ith column of the jth block of $\boldsymbol{A}$.

Note that the values of the ***x_pointers*** elements are all less than or equal to *x_length* = 6, since the elements to be gathered from ***x*** originally reside in the first six locations of ***x***.

Given that you supplied the above ***x_pointers*** values to **block_sparse_setup**, the **block_sparse_matrix_vector_mult** routine will multiply the blocks $\boldsymbol{A}(:, :, 1)$, $\boldsymbol{A}(:, :, 2)$, and $\boldsymbol{A}(:, :,3)$ shown above by the following vectors, respectively:

$$u(:, 1) = \begin{bmatrix} x_1 \\ x_5 \\ x_2 \\ x_4 \end{bmatrix} \qquad u(:, 2) = \begin{bmatrix} x_4 \\ x_2 \\ x_6 \\ x_3 \end{bmatrix} \qquad u(:, 3) = \begin{bmatrix} x_1 \\ x_1 \\ x_3 \\ x_2 \end{bmatrix}$$

The results are the product vectors $v(:, 1)$, $v(:, 2)$, and $v(:, 3)$. For example,

$$v(:, 1) = \boldsymbol{A}(:, :, 1)\, u(:, 1) = \begin{bmatrix} a & f & k & p \\ b & g & l & q \\ c & h & m & r \\ d & i & n & s \\ e & j & o & t \end{bmatrix} \begin{bmatrix} x_1 \\ x_5 \\ x_2 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_1a + x_5f + x_2k + x_4p \\ x_1b + x_5g + x_2l + x_4q \\ x_1c + x_5h + x_2m + x_4r \\ x_1d + x_5i + x_2n + x_4s \\ x_1e + x_5j + x_2o + x_4t \end{bmatrix}$$

In this example, ***y*** has declared extent 8 and true extent *y_length* = 6, but is a different variable than ***x*** (and therefore undergoes a different permutation

Copyright © 1994 Thinking Machines Corporation

than ***x***). The contents of *v* are scattered to form ***y*** using the pointers you supplied in the ***y_pointers*** argument when you called **block_sparse_setup**.

The ***y_pointers*** array must have declared extents (10 x 3) and true extents (*m* x *p*) or (5 x 3). Suppose you supplied the following values in the first (5 x 3) locations of ***y_pointers***:

$$\boldsymbol{y\_pointers} = \begin{bmatrix} 3 & 5 & 1 \\ 4 & 2 & 3 \\ 1 & 4 & 2 \\ 5 & 3 & 6 \\ 1 & 2 & 1 \end{bmatrix}$$

Element (*i, j*) of ***y_pointers*** contains the original location of the ***y*** element to which element *v*(*i,j*) is to be scattered.

Note that the values of the ***y_pointers*** elements are all less than or equal to *y_length* = 6, since the locations of ***y*** to receive scattered product elements are originally the first six locations.

If you had set *mapping* to 0 when calling **block_sparse_setup**, the ***y_pointers*** values shown above would have caused **block_sparse_matrix_vector_mult** to assign ***y*** the values

$$y = \begin{bmatrix} v(3,1) + v(5,1) + v(1,3) + v(5,3) \\ v(2,2) + v(5,2) + v(3,3) \\ v(1,1) + v(4,2) + v(2,3) \\ v(2,1) + v(3,2) \\ v(4,1) + v(1,2) \\ v(4,3) \\ - \\ - \end{bmatrix}$$

Note that colliding values are added.

However, since you set *mapping* to 1 when calling **block_sparse_setup**, the contents of *v* are sent, not to the ***y*** locations specified in ***y_pointers***, but to the *new* locations to which those locations are mapped during the random permutation. If the setup routine assigned ***y_mask*** the values

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

Copyright © 1994 Thinking Machines Corporation

and assigned ***where_is_y*** the values

where_is_y = [8 6 7 4 5 2 1 3]

then **block_sparse_matrix_vector_mult** assigns ***y*** the values

$$y = \begin{bmatrix} - \\ v(4,3) \\ - \\ v(2,1) + v(3,2) \\ v(4,1) + v(1,2) \\ v(2,2) + v(5,2) + v(3,3) \\ v(1,1) + v(4,2) + v(2,3) \\ v(3,1) + v(5,1) + v(1,3) + v(5,3) \end{bmatrix}$$

Recall that the location of element ***A***(*i*, *j, k*) within the block sparse matrix is given by (***y_pointers***(*i*, *k*), ***x_pointers***(*j*, *k*)). Applying this formula to the elements ***A***(:, :, 1), we see that this block is positioned as shown below in the original block sparse matrix:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} c,e & m,o & & r,t & h,j \\ & & & & \\ a & k & & p & f \\ b & l & & q & g \\ d & n & & s & i \end{bmatrix} \end{array}$$

This example illustrates the fact that blocks can be self-overlapping and can have non-contiguous rows and columns.

Example 2: Dense Matrix Block Sparse Matrix Multiplication

In this example, $m = 4$, $n = 3$, and $p = 2$. The coefficient block sparse matrix contains two blocks, each of size (4 x 3), and is embedded in the CM array ***A***. ***A*** has declared extents (10, 10, 2) and true extents (4, 3, 2). The first (4 x 3) elements of each block have the following values:

Copyright © 1994 Thinking Machines Corporation

$$A(:,:,1) = \begin{bmatrix} a & e & i \\ b & f & j \\ c & g & k \\ d & h & l \end{bmatrix} \qquad A(:,:,2) = \begin{bmatrix} m & q & u \\ n & r & v \\ o & s & w \\ p & t & x \end{bmatrix}$$

The ***x*** argument is a matrix with declared extents (8 x 8) and true extents (6 x 8); thus, *x_length* = 6. The elements to be gathered from ***x*** are assumed to be, originally, in the first (6 x 8) locations of the matrix:

$$\boldsymbol{x} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & x_{17} & x_{18} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} & x_{27} & x_{28} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} & x_{48} \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & x_{57} & x_{58} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} & x_{67} & x_{68} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

In this example, when you call **block_sparse_setup**, you set *mapping* = 1. The setup routine permutes the source array element locations within each column, assigns ***x_mask*** the values

Copyright © 1994 Thinking Machines Corporation

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

and assigns ***where_is_x*** the values

$$\boldsymbol{where\_is\_x} = \begin{bmatrix} 2 & 8 & 1 & 2 & 7 & 5 & 3 & 1 \\ 5 & 4 & 4 & 6 & 6 & 8 & 2 & 7 \\ 1 & 5 & 6 & 8 & 2 & 3 & 8 & 8 \\ 8 & 2 & 2 & 4 & 1 & 2 & 5 & 2 \\ 4 & 1 & 8 & 5 & 3 & 6 & 4 & 5 \\ 7 & 3 & 5 & 3 & 8 & 1 & 6 & 6 \\ 3 & 6 & 7 & 1 & 5 & 7 & 1 & 4 \\ 6 & 7 & 3 & 7 & 4 & 4 & 7 & 3 \end{bmatrix}$$

These values indicate that the source array element locations must be permuted as follows:

$$\boldsymbol{x} = \begin{bmatrix} x_{31} & x_{52} & x_{13} & 0 & x_{45} & x_{66} & 0 & x_{18} \\ x_{11} & x_{42} & x_{43} & x_{14} & x_{35} & x_{46} & x_{27} & x_{48} \\ 0 & x_{62} & 0 & x_{64} & x_{55} & x_{36} & x_{17} & 0 \\ x_{51} & x_{22} & x_{23} & x_{44} & 0 & 0 & x_{57} & 0 \\ x_{21} & x_{32} & x_{63} & x_{54} & 0 & x_{16} & x_{47} & x_{58} \\ 0 & 0 & x_{33} & x_{24} & x_{25} & x_{56} & x_{67} & x_{68} \\ x_{61} & 0 & 0 & 0 & x_{15} & 0 & 0 & x_{28} \\ x_{41} & x_{12} & x_{53} & x_{34} & x_{65} & x_{26} & x_{37} & x_{38} \end{bmatrix}$$

That is, you must use this template when permuting the elements of each ***x*** you supply in subsequent multiplication calls associated with this setup call.

Copyright © 1994 Thinking Machines Corporation

The ***x_pointers*** array must have declared extents (10 x 2) and true extents (m x p) or (4 x 2). In this example, you supply the following values in the first (4 x 2) locations of ***x_pointers*** when you call **block_sparse_setup**:

$$\boldsymbol{x\_pointers} = \begin{bmatrix} 2 & 3 \\ 1 & 5 \\ 6 & 4 \\ 3 & 1 \end{bmatrix}$$

The ***x_pointers*** array determines the contents of the matrices of extent (n x m) = (3 x 4) that will be multiplied on the right by the blocks of $\boldsymbol{A}$. If element (i, j) of ***x_pointers*** contains the value k, then the **gen_mat_block_sparse_mat_mult** routine will multiply the ith row of the jth block of $\boldsymbol{A}$ by $\boldsymbol{x}((\boldsymbol{where\_is\_x}(k, l),l)$, when computing the lth row of the product.

Note that the values of the ***x_pointers*** elements are all less than or equal to *x_length* = 6, since the elements to be gathered from ***x*** originally reside in the first six rows of ***x***.

Given that you supplied the above ***x_pointers*** values to **block_sparse_setup**, the **gen_mat_block_sparse_mat_mult** routine will multiply the blocks $\boldsymbol{A}$(:, :, 1) and $\boldsymbol{A}$(:, :,2) shown above by the following matrices, respectively:

$$u(:, :, 1) = \begin{bmatrix} x_{21} & x_{11} & x_{61} & x_{31} \\ x_{22} & x_{12} & x_{62} & x_{32} \\ x_{23} & x_{13} & x_{63} & x_{33} \end{bmatrix} \qquad u(:, :, 2) = \begin{bmatrix} x_{31} & x_{51} & x_{41} & x_{11} \\ x_{32} & x_{52} & x_{42} & x_{12} \\ x_{33} & x_{53} & x_{43} & x_{13} \end{bmatrix}$$

The results are the product matrices v(:, :, 1) and v(:, :, 2). For example,

$$v(:, :, 1) = u(:, :, 1)\,\boldsymbol{A}(:, :, 1) = \begin{bmatrix} x_{21} & x_{11} & x_{61} & x_{31} \\ x_{22} & x_{12} & x_{62} & x_{32} \\ x_{23} & x_{13} & x_{63} & x_{33} \end{bmatrix} \begin{bmatrix} a & e & i \\ b & f & j \\ c & g & k \\ d & h & l \end{bmatrix}$$

In this example, ***y*** has declared extents (8 x 8) and true extents (6 x 8); thus, *y_length* = 6. Also, ***y*** is defined to be the same variable as ***x***, and therefore undergoes the same random permutation as ***x***. The contents of v are scat-

Copyright © 1994 Thinking Machines Corporation

tered to form ***y*** using the pointers you supplied in the ***y_pointers*** argument when you called **block_sparse_setup**.

The ***y_pointers*** array must have declared extents (10 x 2) and true extents (n x p) or (3 x 2). Suppose you supplied the following values in the first (3 x 2) locations of ***y_pointers***:

$$\boldsymbol{y\_pointers} = \begin{bmatrix} 3 & 2 \\ 1 & 3 \\ 6 & 2 \end{bmatrix}$$

If element (i, j) of ***y_pointers*** contains the value k, then element $v(i, l, j)$ is scattered to location $\boldsymbol{y}(\boldsymbol{where\_is\_y}((k, l),l)$.

Note that the values of the ***y_pointers*** elements are all less than or equal to *y_length* = 6, since the locations of ***y*** to receive scattered product elements are originally the first six rows of ***y***.

If you had set *mapping* to 0 when calling **block_sparse_setup**, the ***y_pointers*** values shown above would have caused **gen_mat_block_sparse_mat_mult** to scatter the contents of v to the first (m x n) or (4 x 3) locations of ***y*** as follows:

$$\boldsymbol{y} = \begin{bmatrix} v_{211} & v_{221} & v_{231} & 0 & 0 & 0 & 0 & 0 \\ v_{112} + v_{312} & v_{122} + v_{322} & v_{132} + v_{332} & 0 & 0 & 0 & 0 & 0 \\ v_{111} + v_{212} & v_{121} + v_{222} & v_{131} + v_{232} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_{311} & v_{321} & v_{331} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Note that colliding values are added; and that since, in this example, each matrix $v(:, :, j)$ has dimensions (3 x 3), only the first three columns of ***y*** receive scattered product elements.

However, since you set *mapping* to 1 when calling **block_sparse_setup**, the contents of v are sent, not to the ***y*** locations specified in ***y_pointers***, but to the *new* locations to which those locations were mapped during the random permutation. Since the setup routine applied the same permutation to the

Copyright © 1994 Thinking Machines Corporation

source array and destination array element locations, it assigned ***y_mask*** the values

$$\textit{y\_mask} = \begin{bmatrix} T & T & T & F & T & T & F & T \\ T & T & T & T & T & T & T & T \\ F & T & F & T & T & T & T & F \\ T & T & T & T & F & F & T & F \\ T & T & T & T & F & T & T & T \\ F & F & T & T & T & T & T & T \\ T & F & F & F & T & F & F & T \\ T & T & T & T & T & T & T & T \end{bmatrix}$$

and assigned ***where_is_y*** the values

$$\textit{where\_is\_y} = \begin{bmatrix} 2 & 8 & 1 & 2 & 7 & 5 & 3 & 1 \\ 5 & 4 & 4 & 6 & 6 & 8 & 2 & 7 \\ 1 & 5 & 6 & 8 & 2 & 3 & 8 & 8 \\ 8 & 2 & 2 & 4 & 1 & 2 & 5 & 2 \\ 4 & 1 & 8 & 5 & 3 & 6 & 4 & 5 \\ 7 & 3 & 5 & 3 & 8 & 1 & 6 & 6 \\ 3 & 6 & 7 & 1 & 5 & 7 & 1 & 4 \\ 6 & 7 & 3 & 7 & 4 & 4 & 7 & 3 \end{bmatrix}$$

Therefore, **gen_mat_block_sparse_mat_mult** assigns ***y*** the values

$$\mathbf{y} = \begin{bmatrix} v_{111} + v_{212} & 0 & v_{231} & 0 & 0 & 0 & 0 & 0 \\ v_{211} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & v_{321} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & v_{122} + v_{322} & v_{132} + v_{332} & 0 & 0 & 0 & 0 & 0 \\ v_{112} + v_{312} & v_{121} + v_{222} & v_{331} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & v_{131} + v_{232} & 0 & 0 & 0 & 0 & 0 \\ v_{311} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & v_{221} & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Copyright © 1994 Thinking Machines Corporation

Arbitrary Block Sparse Matrix Operations

Given a block sparse matrix, a vector or dense matrix, and gathering and scattering pointer arrays, the routines described below compute the product of the block sparse matrix with the vector or dense matrix.

SYNTAX

block_sparse_setup (***x_mask***, ***y_mask***, ***where_is_x***, ***where_is_y***, ***x_pointers***, ***y_pointers***, *m, n, p, x_length, y_length, mapping, motion, setup1, setup2, setup3, ier*)

block_sparse_matrix_vector_mult (***y***, ***A***, ***x***, ***x_pointers***, ***y_pointers***, ***y_mask***, *m, n, p, x_length, y_length, setup3, setup1, setup2, ier*)

vector_block_sparse_matrix_mult (***y***, ***A***, ***x***, ***x_pointers***, ***y_pointers***, ***y_mask***, *m, n, p, x_length, y_length, setup3, setup1, setup2, ier*)

block_sparse_mat_gen_mat_mult (***y***, ***A***, ***x***, ***x_pointers***, ***y_pointers***, ***y_mask***, *m, n, p, x_length, y_length, setup3, setup1, setup2, ier*)

gen_mat_block_sparse_mat_mult (***y***, ***A***, ***x***, ***x_pointers***, ***y_pointers***, ***y_mask***, *m, n, p, x_length, y_length, setup3, setup1, setup2, ier*)

deallocate_block_sparse_setup (*setup1, setup2*)

ARGUMENTS

The following definitions assume that the coefficient block sparse matrix is embedded in a three-dimensional CM array, ***A***; that ***A*** is declared with extents (*dim_1*, *dim_2*, *dim_3*); that the portion of ***A*** containing valid data has extents (m, n, p); and that the pointer arrays ***x_pointers*** and ***y_pointers*** are one-based.

y CM array of type real or complex. Destination array. May be the same variable as ***x***. The rank of ***y*** is 1 for **block_sparse_matrix_vector_mult** and **block_sparse_vector_matrix_mult**, and 2 for **block_sparse_mat_gen_mat_mult** and **gen_mat_block_sparse_mat_mult**. These multiplication routines assign values to ***y*** by using the pointers supplied in ***y_pointers*** to scatter elements from the block products. If you set *mapping* = 1 when calling **block_sparse_setup**, the product elements are scattered to the per-

Copyright © 1994 Thinking Machines Corporation

muted locations of ***y***. (The permutation is given by the values of ***where_is_y*** assigned by the setup routine.)

Any initial values of ***y*** are overwritten by the multiplication routines. If the pointer array ***y_pointers*** causes more than one block product element to update the same ***y*** element during the scatter operation, the colliding values are added.

A CM array of rank 3 and the same data type and precision as ***y***. Represents the block sparse matrix. The first two axes have true extents m and n, respectively, and count the rows and columns of the dense blocks. The third axis has true extent p and counts the blocks. Thus, ***A*** contains p dense blocks, each of size $m \times n$. The location of element ***A***(i, j, k) within the block sparse matrix is given by (***y_pointers***(i, k), ***x_pointers***(j, k)).

x CM array of the same data type and precision as ***y***. Source array. Assumed to be dense. May be the same variable as ***y***. The rank of ***x*** is 1 for **block_sparse_matrix_vector_mult** and **vector_block_sparse_matrix_mult**, and 2 for **block_sparse_mat_gen_mat_mult** and **gen_mat_block_sparse_mat_mult**. If you set *mapping* = 1 when calling **block_sparse_setup**, then before calling any of the multiplication routines, you must permute the elements of ***x*** using the permutation returned in ***where_is_x***.

x_mask Logical CM array with the same rank (1 or 2), axis extents, and layout directives as ***x***; used as a mask for the source array. May be the same variable as ***y_mask***. The setup routine ignores the initial contents. On return from **block_sparse_setup**, ***x_mask*** helps you identify the active elements of ***x*** (after the permutation has been applied, if *mapping* = 1). In particular, ***x_mask*** has the following values:

- If *mapping* = 0 and ***x*** has rank 1, ***x_mask***(1:*x_length*) = **.true.**; all other elements of ***x_mask*** are **.false.**.
- If *mapping* = 0 and ***x*** has rank 2, then within each column of ***x_mask***, ***x_mask***(1:*x_length, l*) = **.true.**; all other elements of the column are **.false.**.
- If *mapping* = 1 and ***x*** has rank 1, then ***x_mask***(***where_is_x***(1:*x_length*)) = **.true.**; all other elements of ***x_mask*** are **.false.**.

Copyright © 1994 Thinking Machines Corporation

- If *mapping* = 1 and ***x*** has rank 2, then within each column of ***x_mask***, ***x_mask***(***where_is_x***(*k*,*l*), *l*) = **.true.** for *k* = 1:*x_length*; all other elements of the column are **.false.**.

y_mask Logical CM array with the same rank (1 or 2), axis extents, and layout directives as ***y***; used as a mask for the destination array. May be the same variable as ***x_mask***. The setup routine ignores the initial contents. On return from **block_sparse_setup**, ***y_mask*** has the following values:

- If *mapping* = 0 and ***y*** has rank 1, ***y_mask*** (1:*y_length*) = **.true.**; all other elements of ***y_mask*** are **.false.**.
- If *mapping* = 0 and ***y*** has rank 2, then within each column of ***y_mask***, ***y_mask***(1:*y_length, l*) = **.true.** and all other elements of the column are **.false.**.
- If *mapping* = 1 and ***y*** has rank 1, then ***y_mask***(***where_is_y***(1:*y_length*)) = **.true.**; all other elements of ***y_mask*** are **.false.**.
- If *mapping* = 1 and ***y*** has rank 2, then within each column of ***y_mask***, ***y_mask***(***where_is_y*** (*k*,*l*), *l*) = **.true.** for *k* = 1:*y_length*; all other elements of the column are **.false.**.

Do not modify ***y_mask*** between the setup call and the associated multiplication call(s). When you call one of the multiplication routines, you must supply the values assigned to ***y_mask*** by **block_sparse_setup**.

where_is_x Integer, one-based CM array. Must have the same axis extents and layout directives as ***x*** (rank 1 or 2). The initial values are ignored. On return from **block_sparse_setup**, ***where_is_x*** has the following values:

- If *mapping* = 0, ***where_is_x***(*k*) (for rank 1) or ***where_is_x***(*k*,*l*) (for rank 2) is simply *k*.
- If *mapping* = 1 and ***x*** has rank 1, ***where_is_x***(*k*) is the location to which the *k*th source array location must be mapped.

Copyright © 1994 Thinking Machines Corporation

- If *mapping* = 1 and ***x*** has rank 2, ***where_is_x***(*k*,*l*) is the row number to which location (*k, l*) of the source array must be mapped.

If ***x_mask*** and ***y_mask*** are the same variable, then ***where_is_x*** and ***where_is_y*** must be the same variable.

where_is_y Integer, one-based CM array. Must have the same axis extents and layout directives as ***y*** (rank 1 or 2). The initial values are ignored. On return from **block_sparse_setup**, ***where_is_y*** has the following values:

- If *mapping* = 0, ***where_is_y***(*k*) (for rank 1) or ***where_is_y***(*k*,*l*) (for rank 2) is simply *k*.
- If *mapping* = 1 and ***y*** has rank 1, ***where_is_x***(*k*) is the location to which the *k*th destination array location will be mapped by the multiplication routine.
- If *mapping* = 1 and ***y*** has rank 2, ***where_is_x***(*k*,*l*) is the row number to which location (*k, l*) of the destination array will be mapped by the multiplication routine.

If ***x_mask*** and ***y_mask*** are the same variable, then ***where_is_x*** and ***where_is_y*** must be the same variable.

x_pointers Integer CM array of rank 2. *Must be one-based*. The elements of ***x_pointers*** identify the original locations of the ***x*** elements that are to be gathered into vectors (if ***x*** has rank 1) or matrices (if ***x*** has rank 2) to be multiplied by the blocks of ***A***.

If ***x_mask*** and ***y_mask*** are the same variable, then ***x_pointers*** and ***y_pointers*** must be the same variable.

Before calling **block_sparse_setup**, use the following guidelines to create ***x_pointers***:

- If you are planning to use **block_sparse_matrix_vector_mult** or **block_sparse_mat_gen_mat_mult** (in which ***A*** is the left-hand operand), declare ***x_pointers*** with extents (*dim_2, dim_3*) and place valid data in the subarray whose axes have extents *n* and *p*.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- If you are planning to use **vector_block_sparse_matrix_mult** or **gen_mat_block_sparse_mat_mult** (in which ***A*** is the right-hand operand), declare ***x_pointers*** with extents (*dim_1*, *dim_3*) and place valid data in the subarray whose axes have extents *m* and *p*.
- Assign the elements of ***x_pointers*** values less than or equal to *x_length*.

If ***x*** has rank 1 and ***x_pointers***$(i,j) = k$, then

- **block_sparse_matrix_vector_mult** multiplies the *i*th column of the *j*th block of ***A*** by ***x***(***where_is_x***(k)).
- **vector_block_sparse_matrix_mult** multiplies the *i*th row of the *j*th block of ***A*** by ***x***(***where_is_x***(k)).

If ***x*** has rank 2 and ***x_pointers***$(i,j) = k$, then

- **block_sparse_mat_gen_mat_mult** multiplies the *i*th column of the *j*th block of ***A*** by ***x***(***where_is_x***$(k,l),l$) when computing the *l*th column of the block product.
- **gen_mat_block_sparse_mat_mult** multiplies the *i*th row of the *j*th block of ***A*** by ***x***(***where_is_x*** $(k,l),l$) when computing the *l*th row of the block product.

When defining ***x_pointers***, refer to the rules for using the same variable for two arguments, presented in the description below.

The ***x_pointers*** values you supply to the **block_sparse_setup** routine should refer to the original (unpermuted) locations of ***x***. If you set *mapping* = 1, **block_sparse_setup** modifies the values of ***x_pointers*** so that the gathering operation occurs correctly. Do not modify the contents of ***x_pointers*** between the setup call and the associated multiplication call(s). When you call one of the multiplication routines, you must supply the values assigned to ***x_pointers*** by **block_sparse_setup**.

y_pointers Integer CM array of rank 2. *Must be one-based*. The elements of ***y_pointers*** identify the original (unpermuted) locations of the ***y*** elements that are to receive scattered product elements.

If ***x_mask*** and ***y_mask*** are the same variable, then ***x_pointers*** and ***y_pointers*** must be the same variable.

Copyright © 1994 Thinking Machines Corporation

Before calling **block_sparse_setup**, use the following guidelines to create ***y_pointers***:

- If you are planning to use **block_sparse_matrix_vector_mult** or **block_sparse_mat_gen_mat_mult** (in which ***A*** is the left-hand operand), declare ***y_pointers*** with extents (*dim_1, dim_3*) and place valid data in the subarray whose axes have extents *m* and *p*.
- If you are planning to use **gen_mat_block_sparse_mat_mult** or **vector_block_sparse_matrix_mult** (in which ***A*** is the right-hand operand), declare ***y_pointers*** with extents (*dim_2*, *dim_3*) and place valid data in the subarray whose axes have extents *n* and *p*.
- Assign the elements of ***y_pointers*** values less than or equal to *y_length*.

The values of ***y_pointers*** determine the scattering pattern as follows:

- If ***y*** has rank 1 and ***y_pointers***$(i,j) = k$, then the ith element from the jth block product is scattered to ***y***(***where_is_y***(k)).
- If ***y*** has rank 2 and ***y_pointers***$(i,j) = k$, then element (i, l) from the jth block product is scattered to ***y***(***where_is_y***$(k,l),l)$.

When defining ***y_pointers***, refer to the rules for using the same variable for two arguments, presented in the description below.

The ***y_pointers*** values you supply to the **block_sparse_setup** routine should refer to the original locations of ***y***. If you set *mapping* = 1, **block_sparse_setup** modifies the values of ***y_pointers*** so that the scattering operation occurs correctly. Do not modify the contents of ***y_pointers*** between the setup call and the associated multiplication call(s). When you call the multiplication routines, you must supply the values that **block_sparse_setup** assigned to ***y_pointers***.

m Scalar integer variable. The true extent of the first axis of ***A***. Also the true extent of the first axis of ***y_pointers*** (for operations in which ***A*** is the left-hand operand) or of ***x_pointers*** (for operations in which ***A*** is the right-hand operand).

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

n — Scalar integer variable. The true extent of the second axis of ***A***. Also the true extent of the first axis of ***x_pointers*** (for operations in which ***A*** is the left-hand operand) or of ***y_pointers*** (for operations in which ***A*** is the right-hand operand).

p — Scalar integer variable. The true extent of the third axis of ***A***, and the true extent of the second axis of ***x_pointers*** and ***y_pointers***.

x_length — Scalar integer variable. The true extent of the first axis of the source array ***x***. Prior to permuting the data elements of ***x***, you must arrange your data so that the first *x_length* locations (for rank 1) or rows (for rank 2) of ***x*** contain the elements to be gathered.

y_length — Scalar integer variable. The true extent of the first axis of the destination array ***y***. The multiplication routines assume that the original, unpermuted locations of ***y*** that are to receive scattered product elements are the first *y_length* locations (for rank 1) or first *y_length* rows (for rank 2).

mapping — Scalar integer variable. Must be 0 or 1. Setting *mapping* to 1 causes the setup routine to return random permutations of the source and destination array element locations. If *mapping* is 0, identity permutations are returned.

motion — Scalar integer variable. Must be 0, 1, or 2. If you supply *motion* = 0, the communication required for the multiplication uses get and send-add operations. If you supply *motion* = 1, the communication uses the partitioned gather and scatter routines (**part_gather** and **part_scatter**). If you supply *motion* = 2, the communication uses the sparse gather and scatter routines (**sparse_util_gather** and **sparse_util_scatter**). The Notes section, below, includes guidelines for choosing a *motion* value.

The setup routine modifies the contents of *motion*. Do not modify *motion* after the setup routine returns; you must supply the associated multiplication and deallocation routines with the value contained in *motion* upon return from the setup routine.

Copyright © 1994 Thinking Machines Corporation

setup1 Scalar integer variable. Internal variable. The initial value you supply when you call **block_sparse_setup** is ignored. Do not modify *setup1* after the setup routine returns; you must supply the associated multiplication and deallocation routines with the value contained in *setup1* upon return from the setup routine.

setup2 Scalar integer variable. Internal variable. The initial value you supply when you call **block_sparse_setup** is ignored. Do not modify *setup2* after the setup routine returns; you must supply the associated multiplication and deallocation routines with the value contained in *setup2* upon return from the setup routine.

setup3 Scalar integer variable. Internal variable. The initial value you supply when you call **block_sparse_setup** is ignored. Do not modify *setup3* after the setup routine returns; you must supply the associated multiplication and deallocation routines with the value contained in *setup3* upon return from the setup routine.

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

0	Successful return.
−1	The supplied arguments had mismatched shapes or did not follow the rules for using the same variable for two arguments, presented in the description below.

Upon return from one of the multiplication routines, contains one of the following codes:

0	Successful return.
−1	The supplied arguments had mismatched shapes or did not follow the rules for using the same variable for two arguments, presented in the description below.

DESCRIPTION

Setup and Deallocation. Follow these steps to perform one multiplication operation (or multiple operations, sequentially):

1. Call **block_sparse_setup**.

Copyright © 1994 Thinking Machines Corporation

2. Call one or more of the multiplication routines listed below. (In the formulas below, x and y denote vectors while X and Y denote matrices. However, lowercase letters are used for both cases everywhere else in this text.)

   block_sparse_matrix_vector_mult $y = Ax$
   block sparse matrix x vector

   vector_block_sparse_matrix_mult $y^T = x^T A$
   vector x block sparse matrix

   block_sparse_mat_gen_mat_mult $Y = AX$
   block sparse matrix x dense matrix

   gen_mat_block_sparse_mat_mult $Y^T = X^T A$
   dense matrix x block sparse matrix

   To compute more than one product using the same gathering and scattering pointer arrays, follow one call to **block_sparse_setup** with multiple calls to the multiplication routines. If the pointer arrays change, start with Step 1 again.

3. After all multiplication calls associated with the same **block_sparse_setup** call, call **deallocate_block_sparse_setup** to deallocate the processing element storage space required by the setup routine.

More than one setup may be active at a time. That is, you may call the setup routine more than once without calling the deallocation routine.

Setup Phase. Given the sparsity of the block sparse matrix ***A***, the gathering and scattering pointers (***x_pointers*** and ***y_pointers***) to be used in subsequent multiplications, and the shapes of the source array ***x*** and destination array ***y***, **block_sparse_setup** initializes masks and location vectors for ***x*** and ***y*** and assigns appropriate values to internal variables required by the multiplication routines. The setup routine modifies the contents of ***x_pointers*** and ***y_pointers***.

The setup routine provides two options that may improve performance significantly:

- The *motion* argument allows you to choose a communication algorithm. Your choice determines what kind of preprocessing the setup routine performs, how much time the preprocessing takes, and how much the performance of the multiplication routine improves as a result. See the Notes section for guidelines.

Copyright © 1994 Thinking Machines Corporation

- If you set *mapping* = 1, **block_sparse_setup** returns random permutations of the source and destination array element locations in ***where_is_x*** and ***where_is_y***, respectively, and returns the new masks for the source and destination arrays in ***x_mask*** and ***y_mask***, respectively. (If the source and destination arrays are the same variable, the same permutation is applied to both arrays.) You must apply the permutation indicated in ***where_is_x*** to the source arrays you supply in subsequent multiplication calls. The permutation indicated in ***where_is_y*** is applied to the destination array by the multiplication routine.

Multiplication Phase. Given a sparse matrix, ***A***, represented in block form, and source and destination arrays ***x*** and ***y***, respectively, the multiplication routines compute the product of ***A*** with ***x*** and return the product in ***y***.

Deallocation Phase. The **deallocate_block_sparse_setup** routine deallocates the memory that was allocated in a previous call to **block_sparse_setup**. Each setup call should be followed (after one or more associated calls to the multiplication routines) by a deallocation call.

Overlapping Variables. Several possibilities exist for using the same variable for two arguments. However, the current release supports the following two cases only:

- Each argument uses a different variable.
- ***x_mask*** and ***y_mask*** are the same variable. In this case, ***where_is_x*** and ***where_is_y*** must be the same variable, and ***x_pointers*** and ***y_pointers*** must be the same variable. If you do not follow this rule, an error code (*ier* = −1) is returned.

NOTES

Argument Values. The internal variables *setup1*, *setup2*, and *setup3* are required for communicating information between the setup phase and the multiplication phase. The application must not modify the contents of these variables. Similarly, after a call to the setup routine, the application should not modify the contents of pointers ***x_pointers*** and ***y_pointers***.

The destination array ***y*** is the only argument updated by the multiplication routines.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Use of Setup Routine. The setup routine must be called whenever the sparsity of the sparse system, represented by pointer arrays ***x_pointers*** and ***y_pointers***, changes. For performance reasons, the cost of the setup phase should be amortized over several multiplications.

Guidelines for Choosing *motion* Values. The *motion* argument has the meanings indicated below.

Value of *motion*	Operations Used by Communication algorithm
0	get, send-add
1	**part_gather**, **part_scatter**
2	**sparse_util_gather**, **sparse_util_scatter**

The value of *motion* determines how much time the preprocessing takes, and how much the performance of the multiplication routine improves as a result. With *motion* = 0, the setup routine uses the least amount of preprocessing time and has the smallest effect on the performance of the multiplication routine. With *motion* = 1, the setup routine uses the most preprocessing time but yields the highest performance improvement in the multiplication phase. If you are calling the multiplication routine a small number of times, you may do best to choose *motion* = 0. However, if you are performing many multiplications with the same sparsity, you will probably achieve the best performance improvement with *motion* = 1.

Numerical Stability. The block sparse matrix operations are stable.

Numerical Complexity. Each block sparse matrix operation requires approximately 2*mnp* floating-point operations for real operands, or 8*mnp* floating-point operations for complex operands.

Performance Hints. Performance is best when the blocks are local to a processing element. You may meet this condition by using the **:serial** layout directive on axes 1 and 2, by using a very high weight on these axes, or by using the detailed layout axis descriptors, **:procs** and **:block**. Typical examples are as follows:

```
cmf$layout a(:serial, :serial, :news)
cmf$layout src_pointers(:serial, :news)
cmf$layout dest_pointers(:serial, :news)
     real a(24, 24, 16000)
```


Copyright © 1994 Thinking Machines Corporation

```
      integer src_pointers(24, 16000)
      integer dest_pointers(24, 16000)

cmf$layout a(:serial, 100000:news, :news)
cmf$layout src_pointers(100000:news, :news)
cmf$layout dest_pointers(100000:news, :news)
      real a(81, 81, 8000)
      integer src_pointers(81, 8000)
      integer dest_pointers(81, 8000)
```

EXAMPLES

Sample CM Fortran code that uses the block sparse matrix operations can be found on-line in the subdirectory

`block-sparse/cmf/`

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

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

4.4 Grid Sparse Matrix Operations

This section introduces the grid sparse matrix operations. For detailed information about the routines and their arguments, refer to the man page at the end of this section.

4.4.1 The Grid Sparse Matrix Routines

Given coefficient arrays, an operand array, and a product array on a 1-, 2-, or 3-dimensional grid, the grid sparse matrix routines compute the product of the grid sparse matrix represented by the coefficient arrays with the vector or dense matrix represented by the operand array. The following routines are provided:

grid_sparse_setup	Sets up the multiplication operation and allocates the necessary partition manager workspace.
grid_sparse_matrix_vector_mult	Multiplies a grid sparse matrix by a vector.
vector_grid_sparse_matrix_mult	Multiplies a vector by a grid sparse matrix.
grid_sparse_mat_gen_mat_mult	Multiplies a grid sparse matrix by a dense matrix.
gen_mat_grid_sparse_mat_mult	Multiplies a dense matrix by a grid sparse matrix.
deallocate_grid_sparse_setup	Deallocates the partition manager workspace allocated by **grid_sparse_setup**.

For information about setup and deallocation, refer to the Description section of the man page at the end of this section.

4.4.2 Grid Sparse Matrix Representation

The grid sparse matrix routines operate on data that is arranged on a grid. Coefficient matrix elements residing at each grid point *P* are multiplied by

Copyright © 1994 Thinking Machines Corporation

vector or matrix elements residing at point *P* and its nearest-neighbor points. The result is placed in product vector or matrix elements residing at point *P.* This section describes these grid operations in detail. Section 4.4.3 describes the matrix representation of these operations. The matrix representation is provided for informational purposes only; applications must use the grid representation described in this section.

Grid Representation

The grid sparse matrix routines assume that you are working with the arrays listed below, and that these arrays are arranged in a 1-, 2-, or 3-dimensional grid.

- Three, five, or seven c*oefficent arrays*:
  - Three arrays, ***a, b,*** and ***c***, if the grid is 1-dimensional.
  - Five arrays, ***a, b, c, d,*** and ***e***, if the grid is 2-dimensional.
  - Seven arrays, ***a, b, c, d, e, f,*** and ***g***, if the grid is 3-dimensional.
- An *operand array*, ***x***.
- A *product array*, ***y***.

Each grid point is associated with either an *element* or a dense *block* of each array.

For example, Figure 6 shows a 1-dimensional grid with 7 points. Each grid point is associated with one element of each array. In this example, ***a, b, c, x,*** and ***y*** all have rank 1.

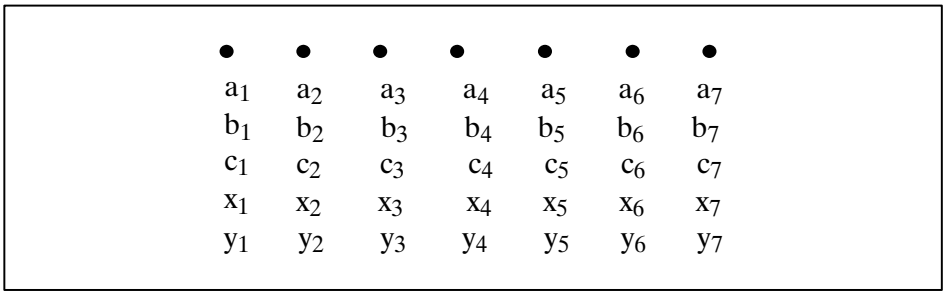

Figure 6. 1-dimensional grid; each grid point is associated with one element of each array.

Copyright © 1994 Thinking Machines Corporation

Figure 7 shows a 2-dimensional grid with dimensions (3 x 3). Again, each grid point is associated with one element of each array. In this example, ***a, b, c, d, e, x,*** and ***y*** all have rank 2. The axes of each array correspond to the axes of the grid.

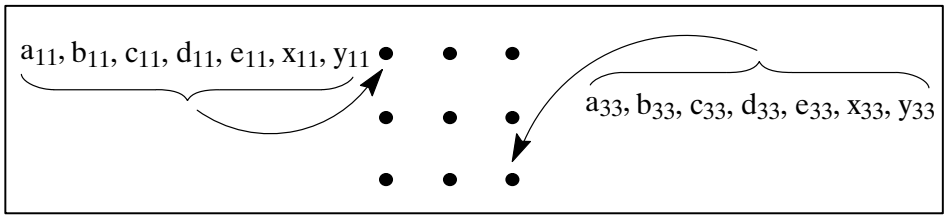

Figure 7. 2-dimensional grid; each grid point is associated with one element of each array.

In contrast, Figure 8 shows a 1-dimensional, 4-point grid, each point of which is associated with a dense *block* of each array. Specifically, each grid point is associated with a (2 x 2) block of each of the coefficient arrays ***a, b,*** and ***c***; a length-2 block of ***x***; and a length-2 block of ***y***. In this example, the arrays ***a, b,*** and ***c*** have rank 3; the first two axes define the block and the third axis corresponds to the grid. The arrays ***x*** and ***y*** have rank 2; the first axis defines the block (which is a vector) and the second corresponds to the grid. In CM Fortran, you would declare these arrays as ***a***(2, 2, 4), ***b***(2, 2, 4), ***c***(2, 2, 4), ***x***(2, 4), and ***y***(2, 4).

Copyright © 1994 Thinking Machines Corporation

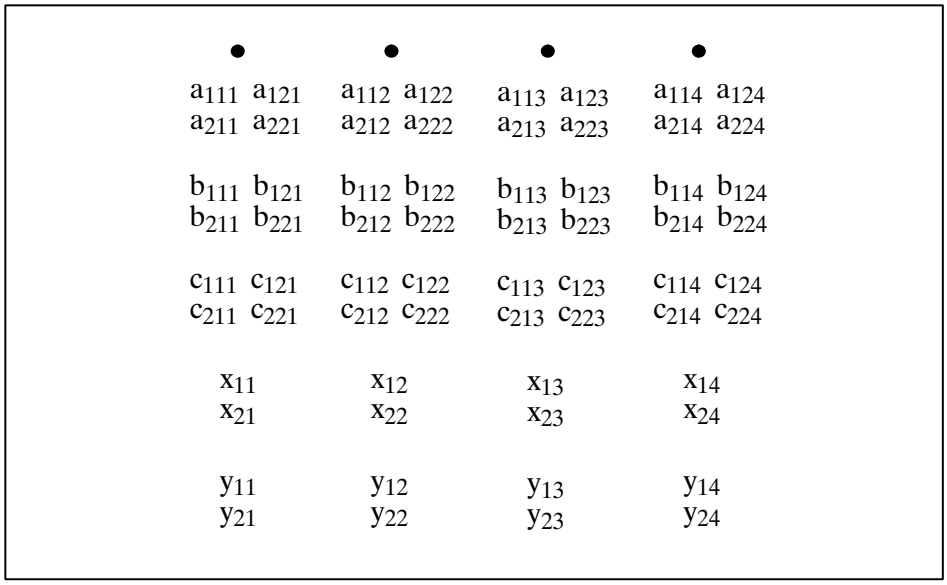

Figure 8. 1-dimensional grid; 2-dimensional coefficient blocks, 1-dimensional operand and product blocks.

Finally, Figure 9 shows the same 1-dimensional grid of length 4, but this time each grid point is associated with a (2 x 2) block of each of the coefficient arrays ***a, b,*** and ***c***; a (2 x 2) block of ***x***; and a (2 x 2) block of ***y***. In this example, the arrays ***a, b, c, x,*** and ***y*** all have rank 3. In CM Fortran, you would declare the arrays as follows:

```
real, array (2,2,4): a, b, c, x, y
```


Copyright © 1994 Thinking Machines Corporation

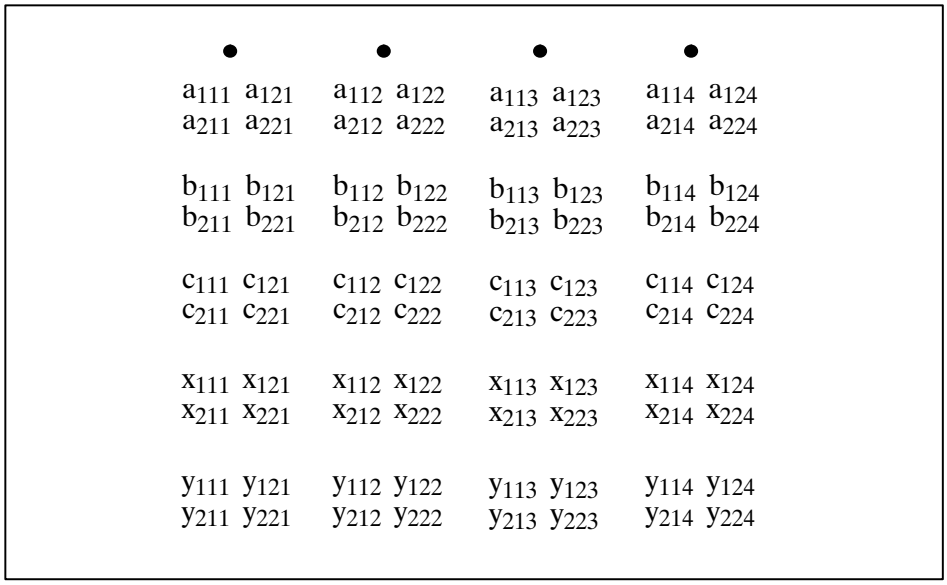

Figure 9. 1-dimensional grid; 2-dimensional coefficient, operand and product blocks.

Grid Axes, Block Axes, and Instance Axes

As the examples in Figure 6 through Figure 9 illustrate, each of the arrays ***a, b, c***[, ***d, e***[, ***f, g***]], ***x***, and ***y*** has one, two, or three axes corresponding to the grid. These axes are called the *grid axes*.

In addition, each of the arrays may have one or two axes that define the dense block associated with each grid point. These axes are called the *block axes*.

Finally, each array may have multiple instances, defined by any number of *instance axes*. (For a discussion of instance axes, refer to Chapter 1.)

Grid Multiplication

The grid sparse matrix routines multiply the elements of the arrays ***a, b, c***[, ***d, e***[, ***f, g***]] by the elements of ***x*** and place the results in ***y***. To compute this product, the routines multiply the elements or blocks of ***a, b, c***[, ***d, e***[, ***f, g***]] at a given grid point by the elements of ***x*** that reside at the same grid point and its nearest-neighbor grid points.

Copyright © 1994 Thinking Machines Corporation

Figure 10 illustrates this process for a 1-dimensional grid. Each point (except the boundary points) has two nearest neighbors. At each point *P*,

- The block of ***a*** is multiplied by the block of ***x*** at point $P-1$.
- The block of ***b*** is multiplied by the block of ***x*** at point *P*.
- The block of ***c*** is multiplied by the block of ***x*** at point $P+1$.

The sum of the results is placed in the block of ***y*** at point *P*. The boundary conditions are under user control. For example, in Figure 10, elements a_{111}, a_{121}, a_{211}, a_{221}, c_{114}, c_{124}, c_{214}, and c_{224} would normally be 0; but you may wish to supply other values, depending on your application.

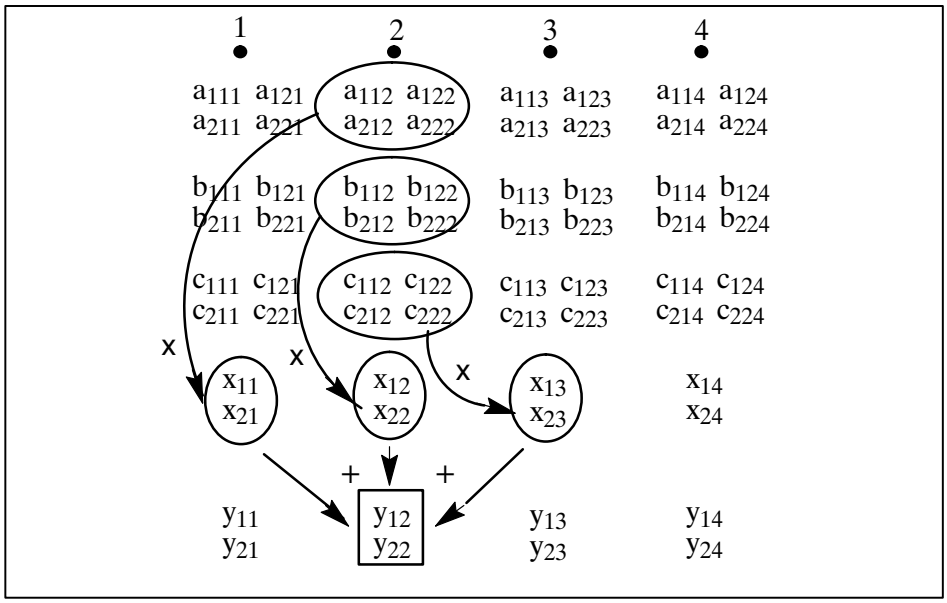

Figure 10. Grid multiplication at a non-boundary point.

Functionally, the routines perform a circular shift (**CSHIFT** in CM Fortran) of the array ***x***. Thus, the high boundary point takes the place of the missing nearest neighbor to the low boundary point, as shown in Figure 11.

Copyright © 1994 Thinking Machines Corporation

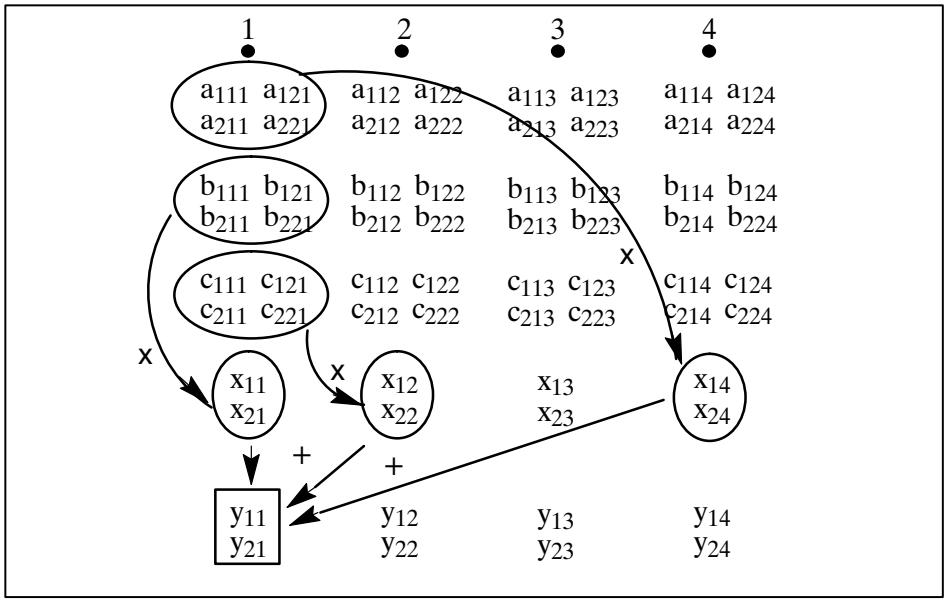

Figure 11. Grid multiplication at a boundary point.

In a 2-dimensional grid, each non-boundary point has four nearest neighbors. An element of ***c*** is multiplied by the element of ***x*** at the same point; elements of ***a, b, d,*** and ***e*** are multiplied by the elements of ***x*** at the nearest neighbors.

Similarly, in a 3-dimensional grid, each non-boundary point has six nearest neighbors. An element of ***d*** is multiplied by the element of ***x*** at the same point; elements of ***a, b, c, e,*** and ***f*** are multiplied by the elements of ***x*** at the nearest neighbors.

The man page at the end of this section includes formulas for grid sparse matrix multiplication; rules for grid, block, and instance axes; and performance hints.

4.4.3 Matrix Representation of the Grid Sparse Matrix Operations

This section describes the grid sparse matrix operations in terms of matrices. This matrix representation is provided for informational purposes only; applications must use the grid representation described in Section 4.4.2.

Copyright © 1994 Thinking Machines Corporation

A grid sparse matrix is a sparse matrix that represents data originating on a grid. The non-zero elements of the matrix contain the data from the grid; the *positions* of the non-zero elements reflect the numbering scheme used to order the points of the grid.

When the grid multiplication described above is represented in matrix form, the elements of the arrays ***a, b, c***[, ***d, e***[, ***f, g***]] form a grid sparse matrix A, and the elements of ***x*** form a vector or matrix. The routines compute the product $A\boldsymbol{x}$ or $\boldsymbol{x}^{\mathrm{T}}A$ and place the results in the vector or matrix ***y***.

The elements of ***a, b, c***[, ***d, e***[, ***f, g***]] associated with one grid point appear on the same row of A; the positions of these non-zero elements in the row ensure that each element is multiplied by the correct element of ***x***.

The exact locations of non-zero elements in the grid sparse matrix depend on the numbering scheme used to label the points of the original grid. Typical labeling schemes may result in the patterns shown in Figure 12 (tridiagonal, 5-diagonal, and 7-diagonal matrices for 1-, 2-, and 3-dimensional grids, respectively). The dots indicate the positions of non-zero elements or dense blocks.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

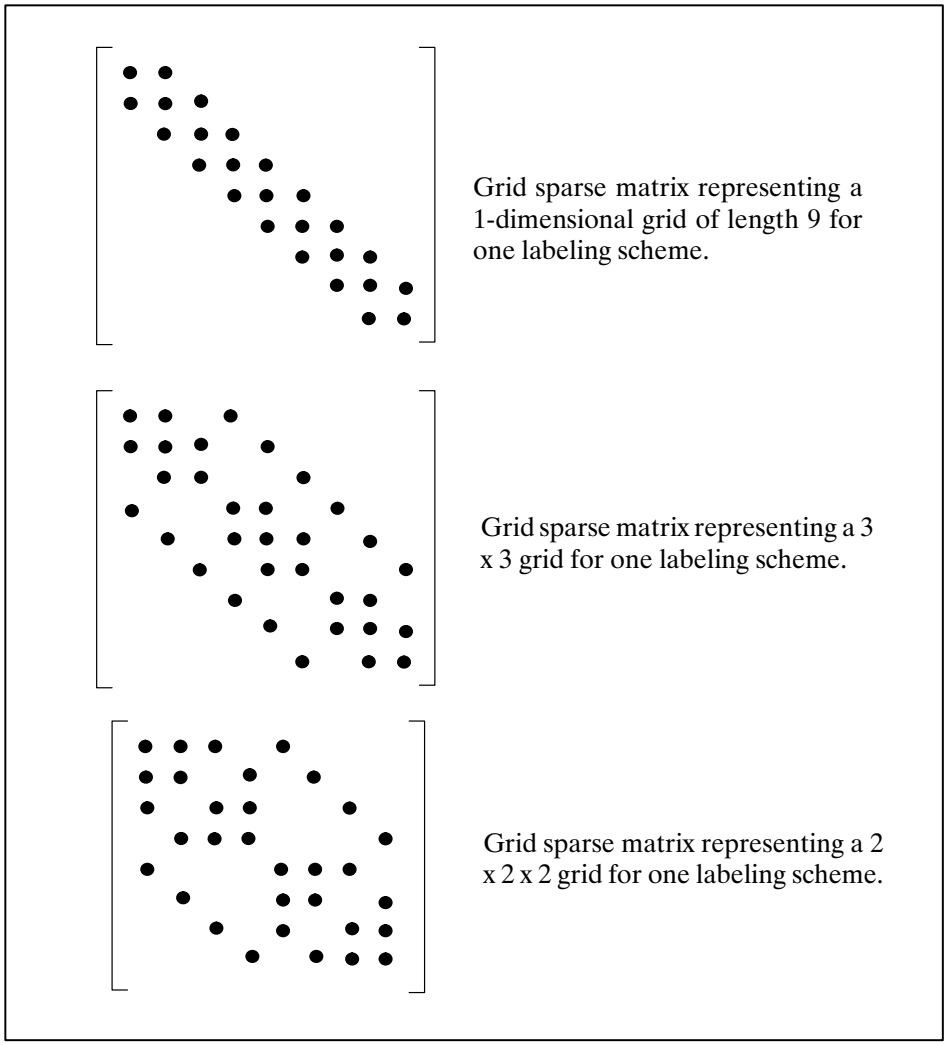

Figure 12. Matrix forms for 1-, 2-, and 3-dimensional grids using a common numbering scheme. Dots indicate the positions of non-zero elements or dense blocks.

Copyright © 1994 Thinking Machines Corporation

For grid sparse matrix representations with the labeling schemes shown in Figure 12, each coefficient array ***a, b, c***[***, d, e***[***, f, g***]] consists of the elements of a diagonal, as shown in Figure 13.

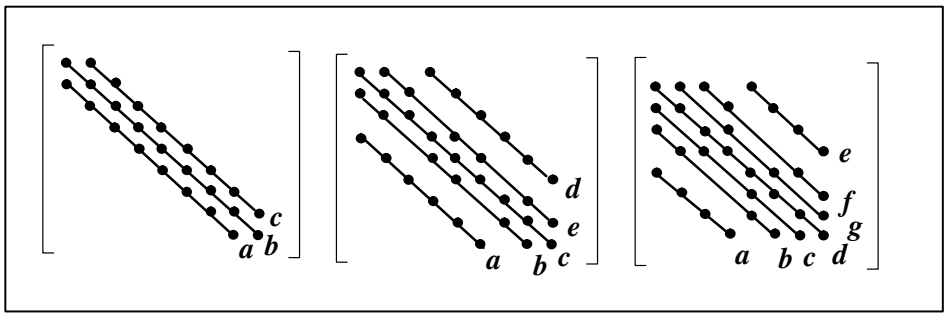

Figure 13. Coefficient array representation of common forms of grid sparse matrices.

Note that some of the elements of ***a, b, c***[***, d, e***[***, f, g***]] associated with boundary points on the grid do not appear in these matrices. The arrays you supply must include these boundary elements; be sure to set the boundary values appropriately for your application.

Figure 14 shows the matrix representations of the grid sparse matrix operations.

Copyright © 1994 Thinking Machines Corporation

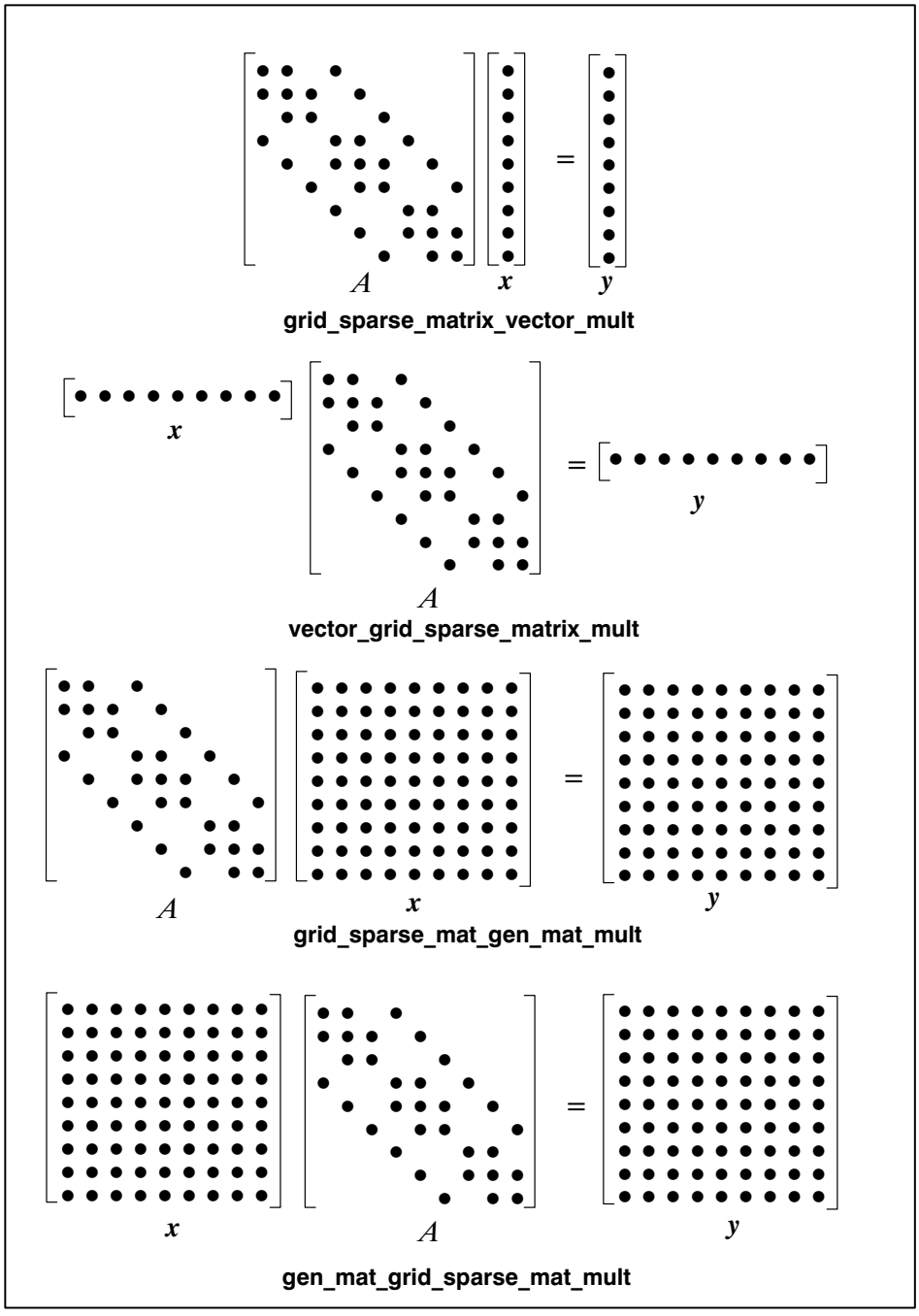

Figure 14. The grid sparse matrix operations in matrix form.

Copyright © 1994 Thinking Machines Corporation

The **grid_sparse_matrix_vector_mult** and **vector_grid_sparse_matrix_mult** routines support the following possibilities:

- Each dot of the grid sparse matrix is a single element; each dot of the vector is a single element.
- Each dot of the grid sparse matrix is a p x q block; each dot of the vector is a vector of length q (for **grid_sparse_matrix_vector_mult**) or length p (for **vector_grid_sparse_matrix_mult**).

The **grid_sparse_mat_gen_mat_mult** and **gen_mat_grid_sparse_mat_mult** routines support the following possibility:

- Each dot of the grid sparse matrix is a p x q block. For **grid_sparse_mat_gen_mat_mult**, each dot of the dense matrix $\boldsymbol{x}$ is a q x r block and each dot of the dense matrix $\boldsymbol{y}$ is an p x r block. For **gen_mat_grid_sparse_mat_mult**, each dot of the dense matrix $\boldsymbol{x}$ is an r x p block and each dot of the dense matrix $\boldsymbol{y}$ is an r x q block.

Each operation can occur in multiple instances.

Copyright © 1994 Thinking Machines Corporation

Grid Sparse Matrix Operations

Given coefficient arrays, an operand array, and a product array on a 1-, 2-, or 3-dimensional grid, the routines described below compute the product of the grid sparse matrix represented by the coefficient arrays with the vector or dense matrix represented by the operand array.

SYNTAX

grid_sparse_setup (***x***, *x_axis_types, setup, ier*)

grid_sparse_matrix_vector_mult (*ier, setup, y_axis_types, coeff_axis_types, x_axis_types,* ***y, x, a, b, c***`[`, ***d, e***`[`, ***f, g***`]]`)

vector_grid_sparse_matrix_mult (*ier, setup, y_axis_types, coeff_axis_types, x_axis_types,* ***y, x, a, b, c***`[`, ***d, e***`[`, ***f, g***`]]`)

grid_sparse_mat_gen_mat_mult (*ier, setup, coeff_axis_types,* ***y, x, a, b, c***`[`, ***d, e***`[`, ***f, g***`]]`)

gen_mat_grid_sparse_mat_mult (*ier, setup, coeff_axis_types,* ***y, x, a, b, c***`[`, ***d, e***`[`, ***f, g***`]]`)

deallocate_grid_sparse_setup (*setup*)

ARGUMENTS

x — CM array of type real or complex. Represents the operand vector or dense matrix in the operation $\boldsymbol{y} = A\boldsymbol{x}$ or $\boldsymbol{y} = \boldsymbol{x}^{\mathrm{T}}A$, where A is a grid sparse matrix represented by the coefficient arrays ***a, b, c***[, ***d, e***[, ***f, g***]].

x_axis_types — Front-end integer vector. The length of *x_axis_types* must be equal to the rank of ***x***. Each element of *x_axis_types* is one of the following symbolic constants, describing the corresponding axis of ***x***:

CMSSL_block_axis
CMSSL_grid_axis
CMSSL_instance_axis

Copyright © 1994 Thinking Machines Corporation

y_axis_types Front-end integer vector. The length of *y_axis_types* must be equal to the rank of ***y***. Each element of *y_axis_types* is one of the following symbolic constants, describing the corresponding axis of ***y***:

CMSSL_block_axis
CMSSL_grid_axis
CMSSL_instance_axis

coeff_axis_types Front-end integer vector. The length of *coeff_axis_types* must be equal to the rank of the arrays ***a, b, c***[, ***d, e***[, ***f, g***]]. Each element of *coeff_axis_types* is one of the following symbolic constants, describing the corresponding axis of ***a, b, c***[, ***d, e***[, ***f, g***]]:

CMSSL_block_axis
CMSSL_grid_axis
CMSSL_instance_axis

y CM array of the same data type and precision as ***x***. Represents the product vector or dense matrix in the operation $\boldsymbol{y} = A\boldsymbol{x}$ or $\boldsymbol{y} = \boldsymbol{x}^{\mathrm{T}}A$, where A is a grid sparse matrix represented by the coefficient arrays ***a, b, c***[, ***d, e***[, ***f, g***]].

a, b, c[, ***d, e***[, ***f, g***]] Coefficient CM arrays of the same data type and precision as ***x***. Must have the same rank, axis extents, and layout directives. See description below.

setup Scalar integer. Internal variable. When you call a multiplication routine or the deallocation routine, you must supply the *setup* value assigned by the associated setup call.

ier Scalar integer. Return code; set to 0 upon successful return, or to −1 if any of the restrictions on axis labels are violated.

DESCRIPTION

Follow these steps to perform one multiplication operation, or multiple operations, sequentially:

1. Call **grid_sparse_setup**.
2. Call one or more of the following multiplication routines:
   - **grid_sparse_matrix_vector_mult** (grid sparse matrix x vector)

Copyright © 1994 Thinking Machines Corporation

- **vector_grid_sparse_matrix_mult** (vector x grid sparse matrix)
- **grid_sparse_mat_gen_mat_mult** (grid sparse matrix x dense matrix)
- **gen_mat_grid_sparse_mat_mult** (dense matrix x grid sparse matrix)

To compute more than one product in which the shape, axis types, and axis declaration order of the operand vectors (or dense matrices) remain the same, follow one call to **grid_sparse_setup** with multiple calls to the multiplication routines. If the shape, axis types, or axis declaration order of the operand vector (or dense matrix) changes, you must start with Step 1 again.

3. After all multiplication calls associated with the same **grid_sparse_setup** call, call **deallocate_block_sparse_setup** to deallocate the partition manager work space allocated by the setup routine.

More than one setup may be active at a time. That is, you may call the setup routine more than once without calling the deallocation routine.

The grid sparse matrix routines support multiple instances. By specifying instance axes, you may perform multiple concurrent operations with each multiplication call.

Axis Types. Each of the arrays ***a, b, c***[, ***d, e***[, ***f, g***]], ***x***, and ***y*** has one, two, or three axes corresponding to the grid. These axes are called the *grid axes*.

In addition, each of the arrays may have one or two axes that define the block associated with each grid point. These axes are called the *block axes*.

Finally, each array may have multiple instances, defined by any number of *instance axes*. (For a discussion of instance axes, refer to Chapter 1.)

Grid Multiplication. The grid sparse matrix routines multiply the elements of the arrays ***a, b, c***[, ***d, e***[, ***f, g***]] by the elements of ***x*** and place the results in ***y***. To compute this product, the routines multiply the elements or blocks of ***a, b, c***[, ***d, e***[, ***f, g***]] at a given grid point by the elements of ***x*** that reside at the same grid point and its nearest-neighbor grid points. The boundary conditions are under user control. Functionally, the routines perform a circular shift (**CSHIFT** in CM Fortran) of the array ***x***. Thus, the high boundary point takes the place of the missing nearest neighbor to the low boundary point.

The formulas for grid sparse matrix multiplication are shown below. In these formulas, *a1, a2,* and *a3* are the grid axes of ***x***.

Copyright © 1994 Thinking Machines Corporation

For **grid_sparse_matrix_vector_mult** and **grid_sparse_mat_gen_mat_mult**:

$\ominus$ is one of the following operations, performed with respect to the block axes:

- multiplication of single elements
- matrix vector multiplication
- matrix matrix multiplication

On a 1-dimensional grid (one grid axis labeled a_1):

$$\begin{aligned} y = \; & a \ominus \textbf{CSHIFT}\,(x, \text{shift} = -1, \text{dim} = a1) + \\ & b \ominus x + \\ & c \ominus \textbf{CSHIFT}\,(x, \text{shift} = +1, \text{dim} = a1) \end{aligned}$$

On a 2-dimensional grid (two grid axes labeled a_1 and a_2):

$$\begin{aligned} y = \; & a \ominus \textbf{CSHIFT}\,(x, \text{shift} = -1, \text{dim} = a1) + \\ & b \ominus \textbf{CSHIFT}\,(x, \text{shift} = -1, \text{dim} = a2) + \\ & c \ominus x + \\ & d \ominus \textbf{CSHIFT}\,(x, \text{shift} = +1, \text{dim} = a1) + \\ & e \ominus \textbf{CSHIFT}\,(x, \text{shift} = +1, \text{dim} = a2) \end{aligned}$$

On a 3-dimensional grid (three grid axes labeled a_1, a_2, and a_3):

$$\begin{aligned} y = \; & a \ominus \textbf{CSHIFT}\,(x, \text{shift} = -1, \text{dim} = a1) + \\ & b \ominus \textbf{CSHIFT}\,(x, \text{shift} = -1, \text{dim} = a2) + \\ & c \ominus \textbf{CSHIFT}\,(x, \text{shift} = -1, \text{dim} = a3) + \\ & d \ominus x + \\ & e \ominus \textbf{CSHIFT}\,(x, \text{shift} = +1, \text{dim} = a1) + \\ & f \ominus \textbf{CSHIFT}\,(x, \text{shift} = +1, \text{dim} = a2) + \\ & g \ominus \textbf{CSHIFT}\,(x, \text{shift} = +1, \text{dim} = a3) \end{aligned}$$

For **vector_grid_sparse_matrix_mult** and **gen_mat_grid_sparse_mat_mult**:

$\ominus$ is one of the following operations, performed with respect to the block axes:

- multiplication of single elements
- vector matrix multiplication
- matrix matrix multiplication

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

On a 1-dimensional grid (one grid axis labeled a_1):

$$\begin{aligned} \boldsymbol{y} = \; & \textbf{CSHIFT}\,(\boldsymbol{a} \ominus \boldsymbol{x}, \text{shift} = +1, \text{dim} = a1) + \\ & \boldsymbol{b} \ominus \boldsymbol{x} + \\ & \textbf{CSHIFT}\,(\boldsymbol{c} \ominus \boldsymbol{x}, \text{shift} = -1, \text{dim} = a1) \end{aligned}$$

On a 2-dimensional grid (two grid axes labeled a_1 and a_2):

$$\begin{aligned} \boldsymbol{y} = \; & \textbf{CSHIFT}\,(\boldsymbol{a} \ominus \boldsymbol{x}, \text{shift} = +1, \text{dim} = a1) + \\ & \textbf{CSHIFT}\,(\boldsymbol{b} \ominus \boldsymbol{x}, \text{shift} = +1, \text{dim} = a2) + \\ & \boldsymbol{c} \ominus \boldsymbol{x} + \\ & \textbf{CSHIFT}\,(\boldsymbol{d} \ominus \boldsymbol{x}, \text{shift} = -1, \text{dim} = a1) + \\ & \textbf{CSHIFT}\,(\boldsymbol{e} \ominus \boldsymbol{x}, \text{shift} = -1, \text{dim} = a2) \end{aligned}$$

On a 3-dimensional grid (three grid axes labeled a_1, a_2, and a_3):

$$\begin{aligned} \boldsymbol{y} = \; & \textbf{CSHIFT}\,(\boldsymbol{a} \ominus \boldsymbol{x}, \text{shift} = +1, \text{dim} = a1) + \\ & \textbf{CSHIFT}\,(\boldsymbol{b} \ominus \boldsymbol{x}, \text{shift} = +1, \text{dim} = a2) + \\ & \textbf{CSHIFT}\,(\boldsymbol{c} \ominus \boldsymbol{x}, \text{shift} = +1, \text{dim} = a3) + \\ & \boldsymbol{d} \ominus \boldsymbol{x} + \\ & \textbf{CSHIFT}\,(\boldsymbol{e} \ominus \boldsymbol{x}, \text{shift} = -1, \text{dim} = a1) + \\ & \textbf{CSHIFT}\,(\boldsymbol{f} \ominus \boldsymbol{x}, \text{shift} = -1, \text{dim} = a2) + \\ & \textbf{CSHIFT}\,(\boldsymbol{g} \ominus \boldsymbol{x}, \text{shift} = -1, \text{dim} = a3) \end{aligned}$$

Rules for Grid Axes, Block Axes, and Instance Axes. The grid sparse matrix routines impose the following requirements with regard to the structure of the arrays ***a, b, c***[, ***d, e***[, ***f, g***]], ***x***, and ***y***:

- If the grid is 1-dimensional with N points, then each array has one grid axis of extent N.
- If the grid is 2-dimensional with $N_1 \times N_2$ points, then each array has two grid axes of extents N_1 and N_2, respectively.
- If the grid is 3-dimensional with $N_1 \times N_2 \times N_3$ points, then each array has three grid axes of extents N_1, N_2, and N_3, respectively.

Copyright © 1994 Thinking Machines Corporation

- The valid combinations of block axes are as follows:
  - For **grid_sparse_matrix_vector_mult** or **vector_grid_sparse_matrix_mult**:

	Number of block axes
a, b, c[, ***d, e***[, ***f, g***]]	0 or 2
x	0 or 1
y	0 or 1

    The first combination (number of block axes = 0) is for multiplication of single elements. The second combination (number of block axes = 2 for coefficients matrix and 1 for vectors) is for matrix vector or vector matrix multiplication.

  - For **grid_sparse_mat_gen_mat_mult** or **gen_mat_grid_sparse_mat_mult**:

	Number of block axes
a, b, c[, ***d, e***[, ***f, g***]]	2
x	2
y	2

    This combination is for matrix matrix multiplication.

- The grid axes, block axes, and instance axes can occur in any order (which you specify when calling the routines), with the condition that the grid, block, and instance axes must occur in the same order in the arrays ***a, b, c***[, ***d, e***[, ***f, g***]]. That is, the arrays ***a, b, c***[, ***d, e***[, ***f, g***]] must all have the same shape (axis declaration order and extents). They must also have the same layout directives.

NOTES

Include the CMSSL Header File. The grid sparse matrix routines use symbolic constants. Therefore, you must include the line

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

at the top of any program module that calls these routines. This file declares the types of the CMSSL functions and symbolic constants.

Argument Values. The internal variable *setup* is required for communicating information between the setup phase and the multiplication phase. The application must not modify the contents of this variable.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Use of Setup Routine. The setup routine must be called whenever the shape, axis types, or axis declaration order of ***x*** changes. For performance reasons, the cost of the setup phase should be amortized over several multiplications.

Use of Deallocation Routine. Be sure to call **deallocate_grid_sparse_setup** to deallocate work space after all of the grid sparse matrix operations associated with the setup call have finished.

Numerical Stability. The grid sparse matrix operations are stable.

Numerical Complexity. If I is the number of instances (the product of the extents of the instance axes), N is the product of the extents of the grid axes, and each block of the coefficient arrays has axis extents $p \times q$, then the **grid_sparse_matrix_vector_mult** and **vector_grid_sparse_matrix_mult** operations require $2pqNI$ floating-point operations for real operands, or $8pqNI$ floating-point operations for complex operands.

If I is the number of instances, N is the product of the extents of the grid axes, each block of the coefficient arrays has axis extents $p \times q$, each block of ***x*** is $q \times r$, and each block of ***y*** is $p \times r$, then the **grid_sparse_mat_gen_mat_mult** and **gen_mat_grid_sparse_mat_mult** operations require $2pqrNI$ floating-point operations for real operands, or $8pqrNI$ floating-point operations for complex operands.

Performance Hints. Performance is strongly dependent on layout, and is best when the axes representing the vectors and matrices are local to a processing element. You may meet this condition by using the **:serial** layout directive or using the detailed axis descriptors of the CM Fortran **CMF$LAYOUT** directive. Otherwise, the routines reshape the arrays, incurring a performance cost.

EXAMPLES

Sample CM Fortran code that uses the grid sparse matrix operations can be found on-line in the subdirectory **`grid-sparse/cmf/`** of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

4.5 References

The following references provide further information about sparse matrix operations:

1. Blelloch, G. *Vector Models for Data Parallel Computing*. Cambridge, MA: MIT Press, 1990.

2. Duff, I. S., et al. *Direct Methods for Sparse Matrices*. Oxford Science Publications, 1986.

3. George, A., and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

4. Mathur, K. K. *On the Use of Randomized Address Maps in Unstructured Three-Dimensional Finite Element Simulation.* Thinking Machines Corporation Technical Report CS90−4, 1990.

5. McKenna, M., and Mesirov, J. Personal communication, 1991.

6. Ranade, A. How to Emulate Shared Memory. *Proc. 28th Annual Symp. Theory of Comp.* ACM (1981): 263−77.

7. Ranade, A., et al. The Fluent Abstract Machine. *Advanced Research in VLSI*. MIT Press, 1988. Pp. 71−94.

8. Valiant, L. G. A Scheme for Fast Parallel Communication. *SIAM Journal of Computing* **11** (1982): 350−61.

9. Young, D. and D. Kincaid. The ITPACK Package for Large Sparse Linear Systems. In *Elliptic Problem Solvers,* ed. M. Schkultz. New York: Academic Press, 1981. Pp. 163−85.

The following reference discusses the partitioning routine used internally with certain preprocessing options of the sparse matrix routines. The partitioning routine is also described in Chapter 15.

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.

Copyright © 1994 Thinking Machines Corporation