Chapter 6

Communication Primitives

6.1 Overview

This chapter describes the C* interfaces for the following CMSSL communication primitives:

- matrix transpose
- sparse gather and scatter utilities
- sparse vector gather and vector scatter utilities
- block gather and scatter utilities
- mesh partitioning and reordering of mappings
- partitioned gather and scatter utilities
- partitioned vector gather and scatter utilities
- vector move (extract and deposit)

Sections 6.2 through 6.11 introduce these operations. Each section is followed by a man page that describes the routine(s) in detail. Section 6.12 provides references.

Copyright © 1993 Thinking Machines Corporation

6.2 Matrix Transpose

The matrix transpose routine, **CMSSL_gen_matrix_transpose**, transposes two axes of a parallel variable. In the current release, this routine is designed specifically to achieve enhanced performance when one of the axes to be exchanged is local (resides within a single processing element) and the other is non-local (spans multiple processing elements). The man page following this section describes the other requirements for achieving enhanced performance. In cases where these requirements are not met, **CMSSL_gen_matrix_transpose** performs the transpose with a simple send operation.

6.2.1 Example

The following code fragment is based on the syntax and requirements presented in the man page. This example occurs on a CM-5 with 128 vector units (32 processing nodes). Axes 1 and 2 of the parallel variable **A** are exchanged, and the resulting parallel variable is returned in **A_T**. For a complete example, see the CMSSL examples directory.

```
unsigned long  ext_a[3] = {64, 256, 512};
unsigned long  ext_at[3] = {64, 512, 256};

CMC_axis_order_t orderings[3] = {CMC_serial_order,
                 CMC_serial_order, CMC_news_order};

shape sh_a = allocate_detailed_shape(
              &sh_a, 3, ext_a, NULL, orderings,
             NULL, NULL, NULL);

shape sh_at = allocate_detailed_shape(
               &sh_at, 3, ext_at, NULL, axis_orderings,
               NULL, NULL, NULL);

float A:sh_a, A_T:sh_at;
...
CMSSL_gen_matrix_transpose(&A_T, &A, 1, 2);
```


Copyright © 1993 Thinking Machines Corporation

Matrix Transpose

The matrix transpose routine exchanges two axes of a parallel variable.

SYNTAX

```
#include <cm/cmssl.h>

void CMSSL_gen_matrix_transpose(
    ifdcz:void *A_T, ifdcz:void *A, int axis_1, int axis_2);
```

OVERLOADED DATA TYPES

ifdcz **int**, **float**, **double**, **CMSSL_complex_t**, or **CMSSL_dcomplex_t**

A and **A_T** must have the same data type. The **CMSSL_complex_t** and **CMSSL_dcomplex_t** data types are defined in the header file **<cm/cmssl.h>**.

ARGUMENTS

A_T	Pointer to a parallel variable of the same rank as **A**. Upon successful completion, **A_T** contains the parallel variable that results when axes **axis_1** and **axis_2** of **A** are exchanged. Axis **axis_1** of **A_T** must have the same length as axis **axis_2** of **A**; axis **axis_2** of **A_T** must have the same length as axis **axis_1** of **A**. All other axes of **A_T** must have the same lengths as the corresponding axes of **A**. **A** and **A_T** must be distinct variables.
A	Pointer to a parallel variable of rank greater than or equal to 2, whose axes are to be exchanged.
axis_1	Supply a value between 1 and the rank of **A**, exclusive. Identifies one of the two axes of **A** to be exchanged.
axis_2	Supply a value between 1 and the rank of **A**, exclusive. Identifies one of the two axes of **A** to be exchanged.

DESCRIPTION

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

$$[.]...[i]...[.]...[j]...[.]\mathbf{A_T} = [.]...[j]...[.]...[i]...[.]\mathbf{A}$$

Copyright © 1993 Thinking Machines Corporation

NOTES

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

- One of the axes to be exchanged (axis **`axis_1`** or axis **`axis_2`**) must be local; the other must be non-local. An axis is *local* if all elements along the axis reside within one processing element. An axis is *non-local* if the elements along the axis span multiple processing elements.
- The processing element layouts (machine geometries) underlying **`A`** and **`A_T`** must be identical. You must ensure that this requirement is met by using the **`extents`** and **`axis_orderings`** arguments of the **`allocate_detailed_shape`** function, which is described in the *C*Programming Guide*.

Local Axes. To ensure that an axis is local to a processing element, you may need to use the **`allocate_detailed_shape`** function, which is described in the *C* Programming Guide*.

EXAMPLES

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

```
matrix-tranpose/cstar/
```

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

Copyright © 1993 Thinking Machines Corporation

6.3 Sparse Gather Utility

The sparse gather utility is a communication primitive that is used internally by the CMSSL basic linear algebra routines for arbitrary sparse matrices (which are not available in the C* interface in this Beta release). The gather utility is intended for applications that do not do explicit sparse linear algebra operations, but want to make use of some of the primitives commonly used in these operations.

6.3.1 The Gather Utility Routines

Given three parallel variables (a source with rank 1, a destination with any rank, and a gathering mapping), the sparse gather routines gather elements from the source into the destination. To perform a gather operation (or multiple gather operations, sequentially), you must follow these steps:

1. Call the **CMSSL_sparse_util_gather_setup** routine.
2. Call the **CMSSL_sparse_util_gather** routine.

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

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

6.3.2 Definition of the Gather Operation

The gather operation is defined by

Copyright © 1993 Thinking Machines Corporation

```
with(shapeof(y))
where (y_mask)    y = [p]x;
```

where **x** is the source from which elements are being gathered, **y** is the destination, **p** contains a mapping supplied by the user application, and **y_mask** is a mask for the destination. These arguments are described in detail in the man page.

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

6.3.3 Gather Operation Examples

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

Example 1: Elementwise Gather Operation

Given the source parallel variable

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

the destination mask

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

(where T denotes **CMSSL_true** and F denotes **CMSSL_false**), and the mapping

p = [0 4 6 0 2 1 1 3 0 – – – –]

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

y = [d i h d k g g b d – – – –].

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

Example 2: Finite-Element Type Application

Given the source

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

the destination mask

$$\texttt{y_mask} = \begin{bmatrix} T & T & T & T & T & F \\ T & T & T & T & T & F \\ T & T & T & T & T & F \\ T & F & T & F & F & F \end{bmatrix},$$

and the mapping

$$\texttt{p} = \begin{bmatrix} 0 & 1 & 3 & 4 & 4 & - \\ 1 & 2 & 4 & 2 & 6 & - \\ 3 & 4 & 5 & 7 & 7 & - \\ 4 & - & 6 & - & - & - \end{bmatrix},$$

the gather operation results in the destination

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

Example 3: Unmasked Destination

Given the source

x = [d g k b],

the mapping

p = [0 3 2 0 2 1],

and the destination mask

Copyright © 1993 Thinking Machines Corporation

y_mask = [T T T T T T],

the gather operation results in the destination

y = [d b k d k g].

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

Sparse Gather Utility

Given three parallel variables (a source with rank 1, a destination with any rank, and a mapping specifying a gathering pattern), the sparse gather routines gather elements from the source into the destination.

SYNTAX

```
#include <cm/cmssl.h>

int CMSSL_sparse_util_gather_setup(
    int:void *p, CMSSL_logical_t:void *y_mask,
    ifdcz:void *x_template, int *setup);

void CMSSL_sparse_util_gather(
    ifdcz:void *y, ifdcz:void *x, int setup);

void CMSSL_deallocate_gather_setup(
    int setup);
```

OVERLOADED DATA TYPES

ifdcz **int**, **float**, **double**, **CMSSL_complex_t**, or **CMSSL_dcomplex_t**

In a given **CMSSL_sparse_util_gather** call, **x** and **y** must have the same data type (which need not match that of **x_template)**. The **CMSSL_complex_t**, **CMSSL_dcomplex_t**, and **CMSSL_logical_t** data types are defined in the header file **<cm/cmssl.h>**.

ARGUMENTS

p Pointer to a parallel variable with the same shape as **y**. Contains the gathering mapping. If an element of **p** contains the value *n*, then the corresponding element of the destination receives the *n*th element of the source during the gather operation. The contents of **p** remain unchanged by **CMSSL_sparse_util_gather_setup**.

y_mask Pointer to a parallel variable with the same shape as **y**. Set to **CMSSL_true** the elements that correspond to active elements of **y**. The contents of **y_mask** remain unchanged by **CMSSL_sparse_util_gather_setup**.

Copyright © 1993 Thinking Machines Corporation

x_template Pointer to a parallel variable with the same shape as **x**. The setup routine uses only the shape of this variable, ignoring the contents.

y Pointer to a parallel variable with arbitrary shape. Destination to which source elements are gathered. The initial values of **y** are overwritten.

x Pointer to a parallel variable of rank 1. Source from which elements are gathered. The contents of **x** remain unchanged by **CMSSL_sparse_util_gather**.

setup Internal variable (pointer to an internal variable, when you call the setup routine). You must supply **CMSSL_sparse_util_gather** and **CMSSL_deallocate_gather_setup** with the value that **CMSSL_sparse_util_gather_setup** assigns to **setup**.

RETURNED VALUE

CMSSL_sparse_util_gather_setup returns 0 on successful return.

DESCRIPTION

Definition. The gather operation is defined by

```
with(shapeof(y))
where (y_mask)    y = [p]x;
```

where **x** is the source from which elements are being gathered, **y** is the destination, and **p** contains a mapping.

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

1. Call **CMSSL_sparse_util_gather_setup**.
2. Call **CMSSL_sparse_util_gather**.

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

Copyright © 1993 Thinking Machines Corporation

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

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

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

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

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

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

NOTES

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

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

Copyright © 1993 Thinking Machines Corporation

EXAMPLES

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

```
sparse-utilities/cstar/
```

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

Copyright © 1993 Thinking Machines Corporation

6.4 Sparse Scatter Utility

The sparse scatter utility is a communication primitive that is used internally by the CMSSL basic linear algebra routines for arbitrary sparse matrices (which are not available in the C* interface in this Beta release). The scatter utility is intended for applications that do not do explicit sparse linear algebra operations, but want to make use of some of the primitives commonly used in these operations.

6.4.1 The Scatter Utility Routines

Given three parallel variables (a source with any rank, a destination with rank 1, and a scattering mapping), the sparse scatter routines scatter elements from the source to the destination. To perform a scatter operation (or multiple scatter operations, sequentially), you must follow these steps:

1. Call the **CMSSL_sparse_util_scatter_setup** routine.
2. Call the **CMSSL_sparse_util_scatter** routine.

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

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

6.4.2 Definition of the Scatter Operation

The scatter operation is defined by

Copyright © 1993 Thinking Machines Corporation

```
with(shapeof(x))
where (x_mask)     [p]y += x;
```

where **x** is the source from which elements are being scattered, **y** is the destination, **p** contains a mapping supplied by the user application, and **x_mask** is a mask for **x**. These arguments are described in detail in the man page.

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

6.4.3 Scatter Operation Example

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

Given the source

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

with mask

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

(where T denotes **CMSSL_true** and F denotes **CMSSL_false**), and mapping

Copyright © 1993 Thinking Machines Corporation

$$\mathbf{p} = \begin{bmatrix} 0 & 1 & 3 & 4 & 4 \\ 1 & 2 & 4 & 2 & 6 \\ 3 & 4 & 5 & 7 & 7 \\ 4 & - & 6 & - & - \end{bmatrix},$$

and given the initial destination

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

the scatter operation results in the destination

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

Copyright © 1993 Thinking Machines Corporation

Sparse Scatter Utility

Given three parallel variables (a source with any rank, a destination with rank 1, and a mapping specifying a scattering pattern), the sparse scatter routines scatter elements from the source to the destination.

SYNTAX

```
#include <cm/cmssl.h>

int CMSSL_sparse_util_scatter_setup(
    int:void *p, CMSSL_logical_t:void *x_mask,
    ifdcz:void *y_template, int *setup);

void CMSSL_sparse_util_scatter(
    ifdcz:void *y, ifdcz:void *x, CMSSL_logical_t:void *x_mask,
    int setup);

void CMSSL_deallocate_scatter_setup(
    int setup);
```

OVERLOADED DATA TYPES

ifdcz **int**, **float**, **double**, **CMSSL_complex_t**, or **CMSSL_dcomplex_t**

In a given **CMSSL_sparse_util_scatter** call, **x** and **y** must have the same data type (which need not match that of **y_template**). The **CMSSL_complex_t**, **CMSSL_dcomplex_t**, and **CMSSL_logical_t** data types are defined in the header file **<cm/cmssl.h>**.

ARGUMENTS

p Pointer to a parallel variable with the same shape as **x**. Contains the scattering mapping. If an element of **p** (as supplied to **CMSSL_sparse_util_scatter_setup**) contains the value *n*, then the scatter operation adds the corresponding source element to destination element *n*. *If two or more source elements are sent to the same destination element, the colliding destination values are added*. The contents of **p** are modified by a successful call to **CMSSL_sparse_util_scatter_setup**. When you call **CMSSL_sparse_util_scatter**, supply the values assigned to **p** by **CMSSL_sparse_util_scatter_setup**.

Copyright © 1993 Thinking Machines Corporation

x_mask Pointer to a parallel variable with the same shape as **x**. Set to **CMSSL_true** the elements that correspond to active elements of **x**. Only those source elements for which the mask is true are sent to the destination. Elements of **p** corresponding to masked elements of the source are ignored. The contents of **x_mask** remain unchanged by **CMSSL_sparse_util_scatter_setup**.

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

y_template Pointer to a parallel variable with the same shape as **y**. The setup routine uses only the shape of this variable, ignoring the contents.

y Pointer to a parallel variable of rank 1. The **CMSSL_sparse_util_scatter** routine adds the scattered **x** elements to the initial values of **y**.

x Pointer to a parallel variable of arbitrary shape. Source from which elements are scattered. The contents of **x** remain unchanged by **CMSSL_sparse_util_scatter**.

setup Internal variable (pointer to an internal variable, when you call the setup routine). You must supply **CMSSL_sparse_util_scatter** and **CMSSL_deallocate_scatter_setup** with the value assigned to **setup** by **CMSSL_sparse_util_scatter_setup**.

RETURNED VALUE

CMSSL_sparse_util_scatter_setup returns 0 on successful return.

DESCRIPTION

Definition. The scatter operation is defined by

```
with(shapeof(x))
where (x_mask)     [p]y += x;
```

where **x** is the source from which elements are being scattered, **y** is the destination, and **p** contains a mapping.

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

Copyright © 1993 Thinking Machines Corporation

1. Call **CMSSL_sparse_util_scatter_setup**.

2. Call **CMSSL_sparse_util_scatter**.

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

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

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

Setup Phase. The **CMSSL_sparse_util_scatter_setup** routine analyzes the scattering mapping supplied by the application in the **`p`** argument. Using **`p`** and **`x_mask`**, **CMSSL_sparse_util_ scatter_setup** assigns an appropriate value to the internal variable **`setup`**; this value must be supplied in subsequent calls to **CMSSL_sparse_util_scatter**.

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

As long as the arguments supplied to **CMSSL_sparse_util_scatter_setup** remain the same (except that **`x_mask`** may change — see the arguments list), the application can call **CMSSL_sparse_util_scatter** multiple times following one **CMSSL_sparse_util_scatter_setup** call, each time supplying a source, **`x`**, and receiving in return a destination, **`y`**.

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

Copyright © 1993 Thinking Machines Corporation

EXAMPLES

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

```
sparse-utilities/cstar/
```

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

Copyright © 1993 Thinking Machines Corporation

6.5 Sparse Vector Gather Utility

The sparse vector gather utility, **CMSSL_sparse_util_vec_gather**, performs the same operation as the sparse gather utility, **CMSSL_sparse_util_gather**, except that **CMSSL_sparse_util_vec_gather** operates on vectors rather than individual data elements. The vectors that are gathered may lie along axes of your choice in the source and destination parallel variables; each of these vector axes must be local to a processing element.

6.5.1 Definition of the Vector Gather Operation

The vector gather operation is defined by

```
with shapeof(p)
      where (y_mask)
          for (i=0; i<dimof(y,y_vector_axis); i++)
              [.]...[i]...[.]y = [i][p]x;
```

where **x** is the source from which vectors are being gathered, **y** is the destination, **p** contains a mapping supplied by the user application, **y_mask** is a mask for the destination, and the axis denoted by **[i]** in **y** is axis **y_vector_axis**. These arguments are described in detail in the man page.

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

6.5.2 Examples

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

Copyright © 1993 Thinking Machines Corporation

Example 1: Vector Gather Operation with Masked Destination

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

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

If the destination mask is

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

(where T denotes **CMSSL_true** and F denotes **CMSSL_false**), and the mapping is

p = [0 4 6 0 2 1 1 3 0 – – – –]

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

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

Example 2: Vector Gather Operation with Unmasked Destination

Given the source

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

(again with vectors of length 3 lying along the first axis), mapping

p = [0 4 6 0 2 1 1 3 0],

and destination mask

Copyright © 1993 Thinking Machines Corporation

$$\texttt{y_mask} = \begin{bmatrix} T & T & T & T & T & T & T & T & T \\ T & T & T & T & T & T & T & T & T \\ T & T & T & T & T & T & T & T & T \end{bmatrix},$$

the vector gather operation results in the destination

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

Copyright © 1993 Thinking Machines Corporation

Sparse Vector Gather Utility

Given three parallel variables (a source with rank 2, a destination with any rank, and a mapping specifying a gathering pattern), the sparse vector gather routines gather vectors from the source into the destination.

SYNTAX

```
#include <cm/cmssl.h>

int CMSSL_sparse_util_vec_gather_setup(
    int:void *p, CMSSL_logical_t:void *y_mask,
    ifdcz:void *x_template, int *setup);

void CMSSL_sparse_util_vec_gather(
    ifdcz:void *y, ifdcz:void *x, int y_vector_axis,
    int x_vector_axis, int setup);

void CMSSL_deallocate_vec_gather_setup(
    int setup);
```

OVERLOADED DATA TYPES

ifdcz **int**, **float**, **double**, **CMSSL_complex_t**, or **CMSSL_dcomplex_t**

In a given **CMSSL_sparse_util_vec_gather** call, **x** and **y** must have the same data type. (which need not match that of **x_template**). The **CMSSL_complex_t**, **CMSSL_dcomplex_t**, and **CMSSL_logical_t** data types are defined in the header file **<cm/cmssl.h>**.

ARGUMENTS

p Pointer to a parallel variable with the same shape as the portion of **y** formed by omitting axis **y_vector_axis** of **y**. Contains the gathering mapping. If element $[p_1]...[p_k]$**p** contains the value n, then the vector [.][n]**x** or [n][.]**x** is gathered from the source to locations

$[p_1]...[.]...[p_k]$**y**

in the destination during the gather operation (where [.] denotes axis **y_vector_axis** of **y** and axis **x_vector_axis** of **x**). The contents of **p** remain unchanged by **CMSSL_sparse_util_vec_gather_setup.**

Copyright © 1993 Thinking Machines Corporation

y_mask Pointer to a parallel variable with the same shape as the portion of **y** formed by omitting axis **y_vector_axis** of **y**. Set to **CMSSL_true** the elements that correspond to active vectors within **y**. The contents of **y_mask** remain unchanged by **CMSSL_sparse_util_vec_gather_setup**.

x_template Pointer to a parallel variable with the same shape as the portion of **x** formed by omitting axis **x_vector_axis** of **x**. The setup routine uses only the shape of this variable, ignoring the contents.

y Pointer to a parallel variable with arbitrary shape. Destination to which vectors from the source are gathered. The gathered vectors will lie along axis **y_vector_axis** of **y**. The initial values of **y** are overwritten.

x Pointer to a parallel variable of rank 2. Source from which vectors are gathered. The vectors to be gathered must lie along axis **x_vector_axis** of **x**. The contents of **x** remain unchanged by **CMSSL_sparse_util_vec_gather**.

y_vector_axis Specifies the axis of **y** along which the gathered vectors will lie. This axis must be zero-based and local to a processing element, and must have the same length as axis **x_vector_axis** of **x**.

x_vector_axis Specifies the axis of **x** along which the vectors to be gathered must lie. This axis must be zero-based and local to a processing element, and must have the same length as axis **y_vector_axis** of **y**.

setup Internal variable. (Pointer to an internal variable, when you call the setup routine.) You must supply **CMSSL_sparse_util_vec_gather** and **CMSSL_deallocate_vec_gather_setup** with the value that **CMSSL_sparse_util_vec_gather_setup** assigns to **setup**.

If you need to call both **CMSSL_sparse_util_gather** and **CMSSL_sparse_util_vec_gather**, you can save memory by supplying the **setup** generated by **CMSSL_sparse_util_vec_gather_setup** to **CMSSL_sparse_util_gather**.

RETURNED VALUE

CMSSL_sparse_util_vec_gather_setup returns 0 on successful return.

Copyright © 1993 Thinking Machines Corporation

DESCRIPTION

Definition. The vector gather operation is defined by

```
with shapeof(p)
   where (y_mask)
      for (i=0; i<dimof(y,y_vector_axis); i++)
         [.]...[i]...[.]y = [i][p]x;
```

where **x** is the source from which vectors are being gathered, **y** is the destination, **p** contains a mapping, **y_mask** is a mask for the destination, and the axis denoted by **[i]** in **y** is axis **y_vector_axis**.

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

1. Call **CMSSL_sparse_util_vec_gather_setup**.
2. Call **CMSSL_sparse_util_vec_gather**.

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

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

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

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

Copyright © 1993 Thinking Machines Corporation

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

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

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

NOTES

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

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

Local Axes. To ensure that an axis is local to a processing element, you may need to use the **allocate_detailed_shape** function, which is described in the *C* Programming Guide*.

Layout and Performance. For best performance, axes **x_vector_axis** and **y_vector_axis** should be either the left-most or the right-most axes of their respective parallel variables.

EXAMPLES

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

```
sparse-utilities/cstar/
```


Copyright © 1993 Thinking Machines Corporation

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

Copyright © 1993 Thinking Machines Corporation

6.6 Sparse Vector Scatter Utility

The sparse vector scatter utility, **CMSSL_sparse_util_vec_scatter**, performs the same operation as the sparse scatter utility, **CMSSL_sparse_util_scatter**, except that **CMSSL_sparse_util_vec_scatter** operates on vectors rather than individual data elements. The vectors that are scattered may lie along axes of your choice in the source and destination parallel variables; each of these vector axes must be local to a processing element.

6.6.1 Definition of the Vector Scatter Operation

The scatter operation is defined by

```
with(shapeof(p))
      where (x_mask)
          for (i=0; i<dimof(x, x_vector_axis); i++)
              [i][p]y += [.]...[i]...[.]x;
```

where **x** is the source from which vectors are being scattered, **y** is the destination, **p** contains a mapping supplied by the user application, **x_mask** is a mask for the source, and the axis denoted by **[i]** in **x** is axis **x_vector_axis**. These arguments are described in detail in the man page at the end of this section.

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

6.6.2 Example

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

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

Copyright © 1993 Thinking Machines Corporation

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

If **x_mask** is

x_mask = [T T T T F]

(where T denotes **CMSSL_true** and F denotes **CMSSL_false**), the mapping is

p = [0 3 1 1 −],

(where the symbol **–** indicates masked data), and the initial destination is

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

then the vector scatter operation results in destination

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

Copyright © 1993 Thinking Machines Corporation

Sparse Vector Scatter Utility

Given three parallel variables (a source with any rank, a destination with rank 2, and a mapping specifying a scattering pattern), the sparse vector scatter routines scatter elements from the source to the destination.

SYNTAX

```
#include <cm/cmssl.h>

int CMSSL_sparse_util_vec_scatter_setup(
    int:void *p, CMSSL_logical_t:void *x_mask,
    ifdcz:void *y_template, int *setup);

void CMSSL_sparse_util_vec_scatter(
    ifdcz:void *y, ifdcz:void *x, CMSSL_logical_t:void *x_mask,
    int y_vector_axis, int x_vector_axis, int setup);

void CMSSL_deallocate_vec_scatter_setup(
    int setup);
```

OVERLOADED DATA TYPES

ifdcz **int**, **float**, **double**, **CMSSL_complex_t**, or **CMSSL_dcomplex_t**

In a given **CMSSL_sparse_util_vec_scatter** call, **x** and **y** must have the same data type (which need not match that of **y_template**). The **CMSSL_complex_t**, **CMSSL_dcomplex_t**, and **CMSSL_logical_t** data types are defined in the header file **<cm/cmssl.h>**.

ARGUMENTS

p Pointer to a parallel variable with the same shape as the portion of **x** formed by omitting axis **x_vector_axis** of **x**. Contains the scattering mapping. If element $[p_1]...[p_k]$**p** contains the value n, then the scatter operation adds the vector $[p_1]...[.]...[p_k]$**x** to the vector $[.][n]$**y** or $[n][.]$**y** within the destination (where [.] denotes axis **y_vector_axis** of **y** and axis **x_vector_axis** of **x**). *If two or more source elements are sent to the same destination element, the colliding destination values are added*. The contents of **p** remain unchanged by **CMSSL_sparse_util_vec_scatter_setup**.

Copyright © 1993 Thinking Machines Corporation

x_mask Pointer to a parallel variable with the same shape as the portion of **x** formed by omitting axis **x_vector_axis** of **x**. Set to **CMSSL_true** the elements that correspond to active vectors within **x**. Only those vectors for which the mask is true are sent to the destination. Elements of **p** corresponding to masked locations of the source are ignored. The contents of **x_mask** remain unchanged by **CMSSL_sparse_util_vec_scatter_setup**.

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

y_template Pointer to a parallel variable with the same shape as the portion of **y** formed by omitting axis **y_vector_axis** of **y**. The setup routine uses only the shape of this variable, ignoring the contents.

y Pointer to a parallel variable of rank 2. The **CMSSL_sparse_util_vec_scatter** routine adds the scattered vectors from **x** to the initial values of the vectors that lie along axis **y_vector_axis** of **y**.

x Pointer to a parallel variable of arbitrary shape. Source for the scatter operation. The vectors to be scattered must lie along axis **x_vector_axis** of **x**. The contents of **x** remain unchanged by **CMSSL_sparse_util_vec_scatter**.

y_vector_axis Specifies the axis of **y** along which the scattered vectors will lie. This axis must be zero-based and local to a processing element, and must have the same length as axis **x_vector_axis** of **x**.

x_vector_axis Specifies the axis of **x** along which the vectors to be scattered must lie. This axis must be zero-based and local to a processing element, and must have the same length as axis **y_vector_axis** of **y**.

setup Internal variable. (Pointer to an internal variable, when you call the setup routine.) You must supply **CMSSL_sparse_util_vec_scatter** and **CMSSL_deallocate_vec_scatter_setup** with the value assigned to **setup** by **CMSSL_sparse_util_vec_scatter_setup**.

Copyright © 1993 Thinking Machines Corporation

If you need to call both **CMSSL_sparse_util_scatter** and **CMSSL_sparse_util_vec_scatter**, you can save memory by supplying the **setup** generated by **CMSSL_sparse_util_vec_scatter_setup** to **CMSSL_sparse_util_scatter**.

RETURNED VALUE

CMSSL_sparse_util_vec_scatter_setup returns 0 on successful return.

DESCRIPTION

Definition. The scatter operation is defined by

```
with(shapeof(p))
   where (x_mask)
      for (i=0; i<dimof(x, x_vector_axis); i++)
         [i][p]y += [.]...[i]...[.]x;
```

where **x** is the source from which vectors are being scattered, **y** is the destination, **p** contains a mapping, **x_mask** is a mask for the source, and the axis denoted by **[i]** in **x** is axis **x_vector_axis**.

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

1. Call **CMSSL_sparse_util_vec_scatter_setup**.
2. Call **CMSSL_sparse_util_vec_scatter**.

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

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

Copyright © 1993 Thinking Machines Corporation

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

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

As long as the arguments supplied to **CMSSL_sparse_util_vec_scatter_setup** remain the same (except that **x_mask** may change — see the arguments list), the application can call **CMSSL_sparse_util_vec_scatter** multiple times following one **CMSSL_sparse_util_vec_scatter_setup** call, each time supplying a source, **x**, and receiving in return a destination, **y**.

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

NOTES

Local Axes. To ensure that an axis is local to a processing element, you may need to use the **allocate_detailed_shape** function, which is described in the *C* Programming Guide*.

Layout and Performance. For best performance, axes **x_vector_axis** and **y_vector_axis** should be either the left-most or the right-most axes of their respective parallel variables.

EXAMPLES

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

```
sparse-utilities/cstar/
```

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

Copyright © 1993 Thinking Machines Corporation

6.7 Block Gather and Scatter Utilities

The **CMSSL_block_gather** and **CMSSL_block_scatter** routines move a block of data from a source parallel variable into a destination parallel variable. The source and destination must have the same rank (≥ 2), type (**int**, **float**, **double**, **CMSSL_complex_t**, or **CMSSL_dcomplex_t**), and layout, with at least one local axis and at least one non-local axis. The gather or scatter operation occurs along a single, specified local axis. In the simplest case, a block of data elements is moved from a two-dimensional source (with one local dimension and one non-local dimension) to a similar destination. You can add instances by extending the non-local axis or by adding more axes (which may be local or non-local).

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

You can use **CMSSL_block_gather** and **CMSSL_block_scatter** to avoid implicit indirect addressing or communication operations.

The man page at the end of this section provides detailed information about these routines and their arguments. Refer to the argument list when examining the following three figures. Figure 15 shows a block gather operation; Figure 16 shows a block scatter operation. In these figures, the source and destination parallel variables are two-dimensional; axis **vector_axis** is vertical. Only a single instance is pictured.

Copyright © 1993 Thinking Machines Corporation

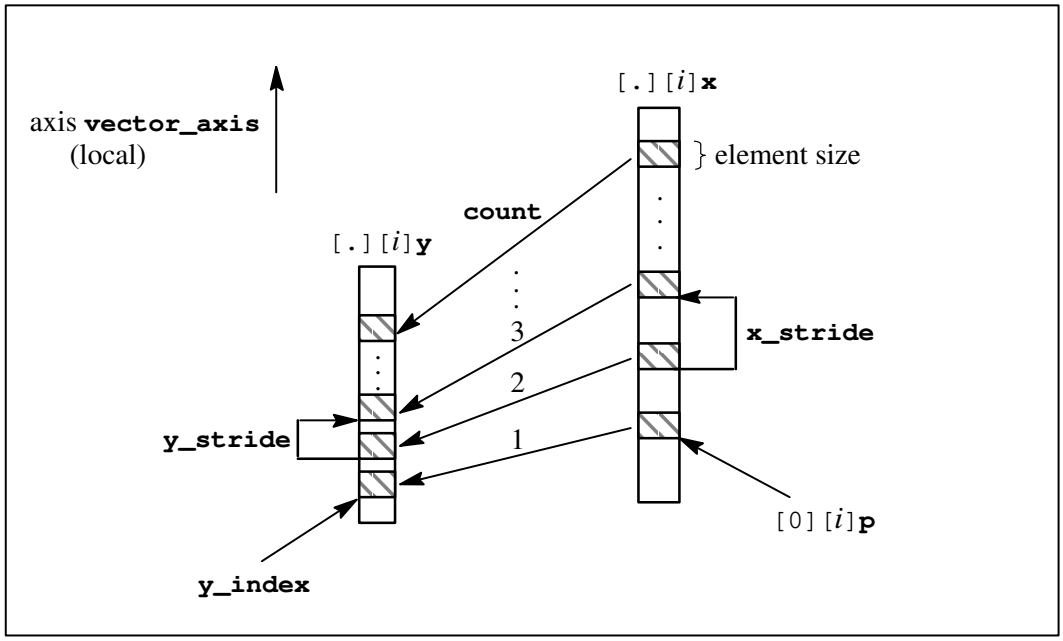

Figure 15. A block gather operation. The source and destination are 2-dimensional. The vertical axis is the selected local axis. Only a single instance is shown.

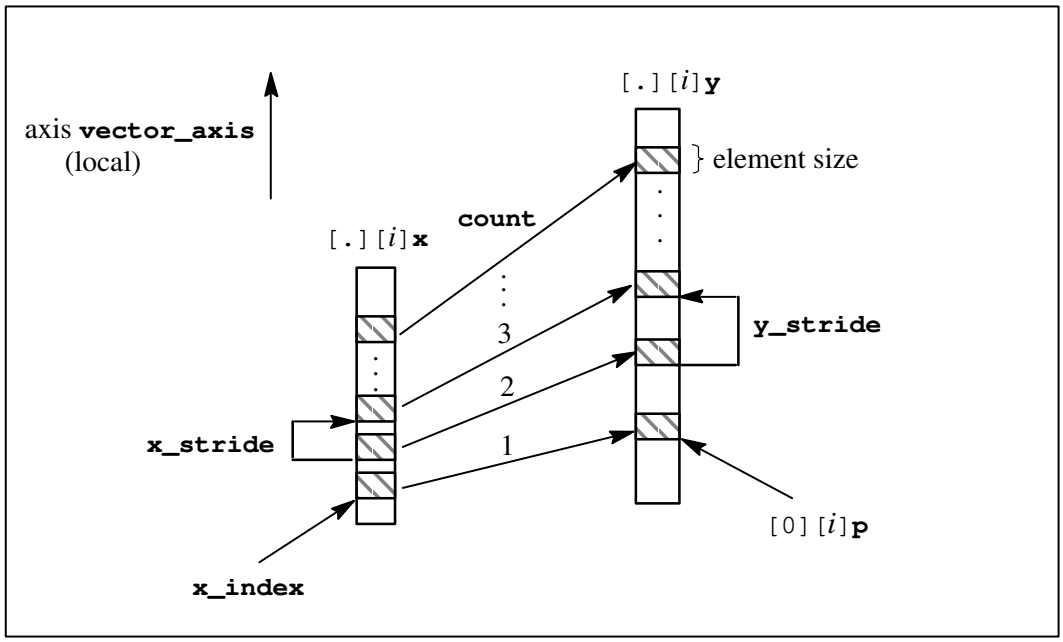

Figure 16. A block scatter operation. The source and destination are 2-dimensional. The vertical axis is the selected local axis. Only a single instance is shown.

Copyright © 1993 Thinking Machines Corporation

Figure 17 shows two instances of the same block scatter operation pictured in Figure 16. Note that the destination starting indices are different for the two instances, whereas the source starting indices, strides, and count are the same for all instances.

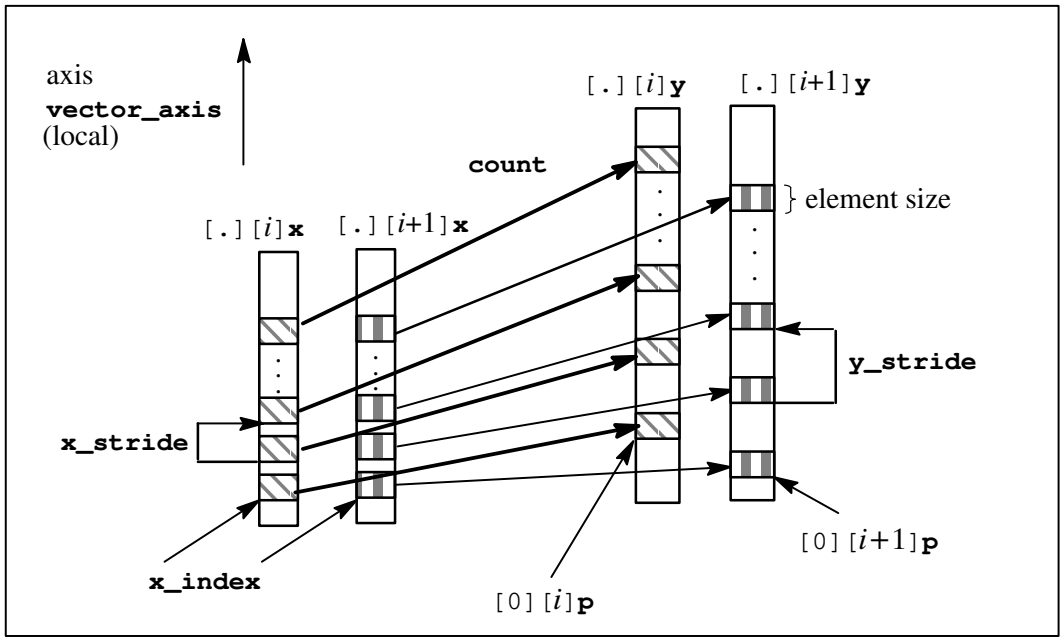

Figure 17. Two instances of the block scatter operation from Figure 16.

Copyright © 1993 Thinking Machines Corporation

Block Gather and Scatter Utilities

The **CMSSL_block_gather** and **CMSSL_block_scatter** routines move a block of data from a source parallel variable into a destination parallel variable. The source and destination must have the same rank (≥ 2), type, and layout, with at least one local axis and at least one non-local axis. In the gather operation, the source starting index can be different for each instance; the destination starting index is the same for all instances. In the scatter operation, the source starting index is the same for all instances; the destination starting index can be different for each instance.

SYNTAX

```
#include <cm/cmssl.h>

void CMSSL_block_gather(
    ifdcz:void *y, int y_stride, int y_index,
    ifdcz:void *x, int x_stride, int:void *p, int count,
    int vector_axis);

void CMSSL_block_scatter(
    ifdcz:void *y, int y_stride, int:void *p,
    ifdcz:void *x, int x_stride, int x_index, int count,
    int vector_axis);
```

OVERLOADED DATA TYPES

ifdcz **int**, **float**, **double**, **CMSSL_complex_t**, or **CMSSL_dcomplex_t**

In a given **CMSSL_block_gather** or **CMSSL_block_scatter** call, **x** and **y** must have the same data type. The **CMSSL_complex_t** and **CMSSL_dcomplex_t** data types are defined in the header file **<cm/cmssl.h>**.

ARGUMENTS

y Pointer to a parallel variable of rank ≥ 2; serves as the destination. Must have at least one local axis and at least one non-local axis.

Copyright © 1993 Thinking Machines Corporation

y_stride Specifies the distance (along axis **vector_axis**) between the **y** locations into which data is to be moved. A gap of (**y_stride**−1) elements will be left between incoming destination data elements. If **y_stride** = 1, destination data elements are placed in locations that are contiguous along axis **vector_axis**.

y_index Zero-based variable. Used only in the **CMSSL_block_gather** call. Specifies the **y** starting index (along axis **vector_axis**) for all instances of the gather operation.

x Pointer to a parallel variable with the same rank and layout as **y**; serves as the source. The axis identified by **vector_axis** may have different lengths in **x** and **y**; however, all other axes must have the same lengths in **x** and **y**.

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

x_index Zero-based variable. Used only in the **CMSSL_block_scatter** call. Specifies the **x** starting index (along axis **vector_axis**) for all instances of the scatter operation.

p Pointer to a parallel variable with the same rank and layout as **x** and **y**. Axis **vector_axis** must have length 1; all other axes must have the same lengths as in **x** and **y**. The meaning of the element

$$[u_1]...[u_{\mathtt{vector_axis}-1}][0][u_{\mathtt{vector_axis}+1}]...[u_n]_{\mathtt{p}}$$

depends on whether you are calling **CMSSL_block_gather** or **CMSSL_block_scatter**:

- When you call **CMSSL_block_gather**, it is the **x** starting index (along axis **vector_axis**) for the gather operation occurring at locations

  $$[u_1]...[u_{\mathtt{vector_axis}-1}][.][u_{\mathtt{vector_axis}+1}]...[u_n]$$

  of **x** and **y**.

Copyright © 1993 Thinking Machines Corporation

- When you call **CMSSL_block_scatter**, it is the **y** starting index (along axis **vector_axis**) for the scatter operation occurring at locations

  $[u_1]...[u_{\texttt{vector_axis}-1}][.][u_{\texttt{vector_axis}+1}]...[u_n]$

  of **x** and **y**.

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

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

DESCRIPTION

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

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

You can use **CMSSL_block_gather** and **CMSSL_block_scatter** to avoid implicit indirect addressing or communication operations.

Given **x** and **y** with rank 2, one local axis, and one non-local axis whose length equals the number of processing elements, the following C* pseudo-code describes the gather operation:

Copyright © 1993 Thinking Machines Corporation

```
unsigned long  ext_y[2] = {32, 64};
unsigned long  ext_x[2] = {128, 64};
unsigned long  ext_p[2] = {1, 64};
unsigned long  ext_q[2] = {count, 64};

CMC_axis_order_t orderings[2] = {CMC_serial_order,
                                 CMC_news_order};

shape sh_y = allocate_detailed_shape(
       &sh_y, 2, ext_y, NULL, orderings,
       NULL, NULL, NULL);

shape sh_x = allocate_detailed_shape(
       &sh_x, 2, ext_x, NULL, orderings,
       NULL, NULL, NULL);

shape sh_p = allocate_detailed_shape(
       &sh_p, 2, ext_p, NULL, orderings,
       NULL, NULL, NULL);

shape sh_q = allocate_detailed_shape(
       &sh_q, 2, ext_q, NULL, orderings,
       NULL, NULL, NULL);

float y:sh_y, x:sh_x;
int p:sh_p;
int q:sh_q;

with (sh_p) {
   [0] [.] q = p;
   }
with (sh_q) {
   q = copy_spread (&q, 0, 0);
   [. * y_stride + y_index] [.] y = [. * x_stride + q] [.] x;
}
```

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

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

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

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

Block gather: ... [*i*] ... **y** = ... [... [0] ... **p**] ... **x**

Block scatter: ... [... [0] ... **p**] ... **y** = ... [i] ... **x**

where

$$\texttt{y_index} \leq i \leq \texttt{y_index} + (\texttt{count} - 1) * \texttt{y_stride}$$
$$\texttt{x_index} \leq i \leq \texttt{x_index} + (\texttt{count} - 1) * \texttt{x_stride}$$

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

NOTES

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

EXAMPLES

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

```
block-scatter-gather/cstar/
```

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

Copyright © 1993 Thinking Machines Corporation

6.8 Partitioning of an Unstructured Mesh and Reordering of Mappings

The routines described in this section allow you to reorder a mapping derived from a mesh so that the communication required by subsequent gather and scatter operations is reduced. The following routines are provided:

CMSSL_generate_dual — Given a parallel variable **ien** containing *element nodes* values that describe an unstructured mesh, this routine computes the corresponding *dual connectivity* values and places them in the parallel variable **idual**.

CMSSL_partition_mesh — Given a parallel variable **idual** containing dual connectivity values describing an unstructured mesh, this routine returns a permutation, **q**, that reorders the mesh elements to form discrete *partitions*. The routine also returns the number of resulting partitions and the number of elements per partition.

CMSSL_reorder_mapping — Given a parallel variable **p** containing a mapping, and given a permutation **q** (for example, the permutation returned by **CMSSL_partition_mesh**), this routine reorders **p** along its last axis using **q**.

CMSSL_renumber_mapping — Given a mapping **p** that has been reordered by **CMSSL_reorder_mapping**, this routine changes the supplied mapping to improve data locality and returns the renumbering mapping.

If you derive the mapping **p** from a mesh, reorder **p** using the permutation **q** returned by **CMSSL_partition_mesh**, and then supply this reordered mapping to the setup routine for the partitioned gather or scatter operation (described in Sections 6.9 and 6.10, respectively), the setup routine takes

Copyright © 1993 Thinking Machines Corporation

advantage of data locality, and the communication required by the gather or scatter is reduced.

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

6.8.1 Definitions

An *unstructured mesh* is a finite collection of elements and nodes. Each element is a bounded region of n-space (where $n = 1, 2,$ or 3) defined by nodes and faces. Each element and each node is assigned a number. Figure 18 shows a two-dimensional unstructured mesh with 11 elements. Node numbers are not shown in this figure.

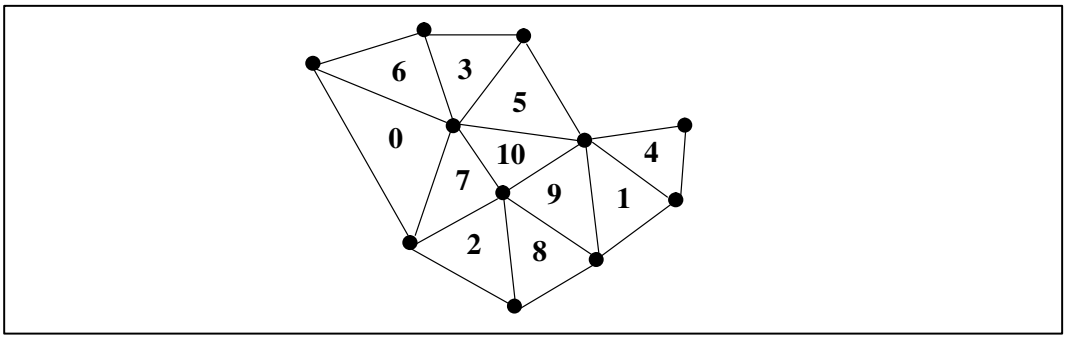

Figure 18. A two-dimensional unstructured mesh with 11 elements.

Two ways of representing an unstructured mesh are as follows:

- Each node in the mesh is associated with one or more elements of the *element nodes* parallel variable, **`ien`**, defined by

  [*m*][*n*]**`ien`** The node number of the *m*th node of the *n*th mesh element.

  This parallel variable has dimensions [*nnode*][*nel*], where *nnode* is the maximum number of nodes per element and *nel* is the number of elements in the mesh. In Figure 19, [0][7]**`ien`** == [1][0]**`ien`** == 0.

Copyright © 1993 Thinking Machines Corporation

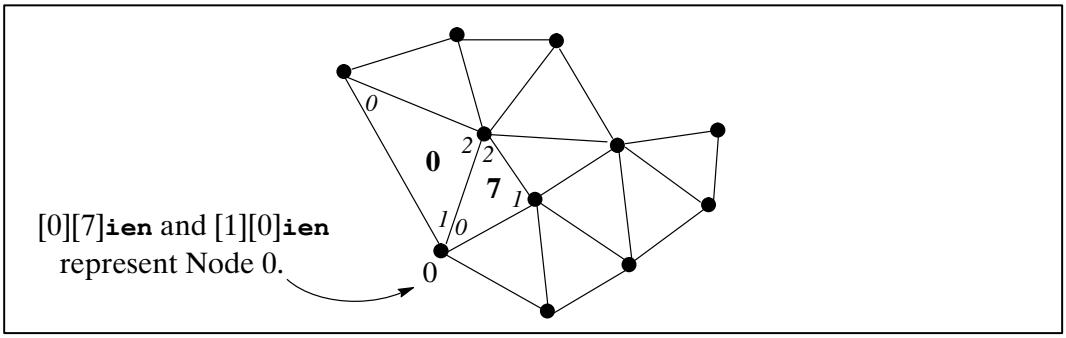

Figure 19. The `ien` parallel variable identifies nodes.

- The connectivity of the mesh elements across faces is represented by the *dual connectivity* parallel variable, **`idual`**, defined by

  [*m*][*n*]**`idual`** The element that shares face *m* with element *n*;
  [*m*][*n*]**`idual`** −1 if face *m* of element *n* is a boundary of the mesh.

  This parallel variable has dimensions [*nface*][*nel*], where *nface* is the maximum number of faces per element and *nel* is the number of elements in the mesh. In Figure 20, [0][0]**`idual`** == 6, [1][0]**`idual`** == 7, and [2, 0]**`idual`** == −1.

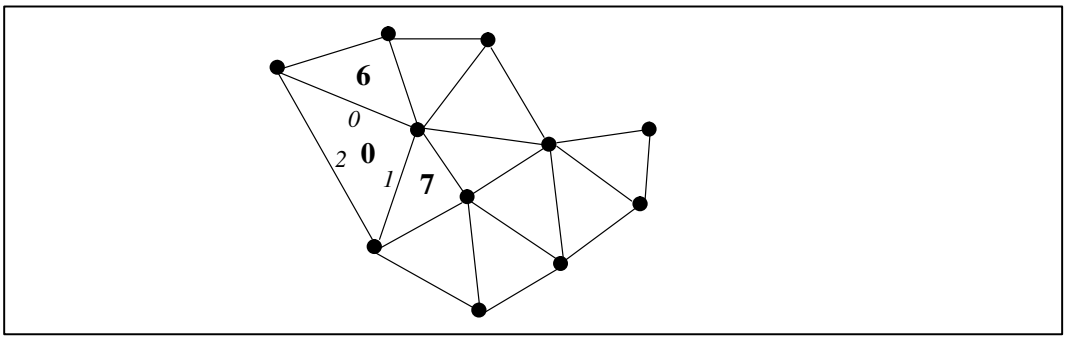

Figure 20. The `idual` parallel variable identifies neighboring elements.

Copyright © 1993 Thinking Machines Corporation

NOTE

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

6.8.2 Finite Element Numbering Scheme

The **CMSSL_generate_dual** routine generates the **idual** values corresponding to the **ien** values and element type you supply. Your **ien** parallel variable must use the standard finite element numbering scheme. Figure 21 shows the standard numbering schemes for the element types supported by **CMSSL_generate_dual**.

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

Copyright © 1993 Thinking Machines Corporation

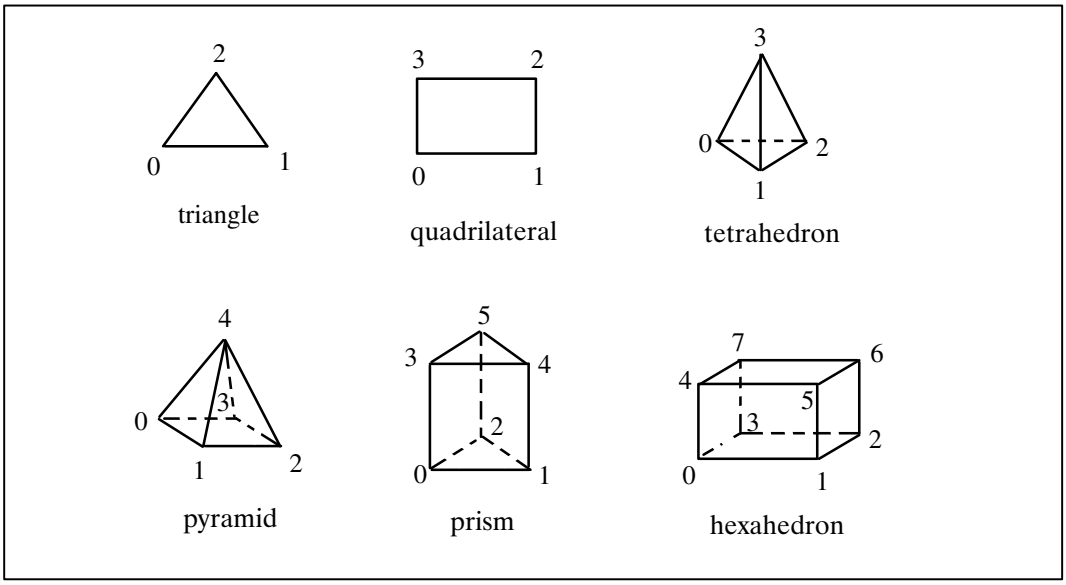

Figure 21. The standard finite element numbering schemes expected by CMSSL_generate_dual.

6.8.3 Partitioning Rules

You must supply the **CMSSL_partition_mesh** routine with **idual** values that describe a mesh. The routine determines the number of partitions based on the number of mesh elements and the value of the input argument **numproc**, as described below. Note that in this discussion and throughout this book, the term *processing element* refers to a vector unit or, on machines without vector units, a (SPARC) processing node.

The elements of the mesh (represented by the parallel variable **idual**) are laid out on the processing elements by the compiler; they span as many processing elements as possible. (On machines with vector units, the layout is subject to the multiple-of-eight rule — that is, the per-processor subgrid size must be a multiple of eight; the compiler pads the subgrid with garbage elements to reach a legal size.) Partitioning is accomplished as follows:

1. The **CMSSL_partition_mesh** routine divides the processing elements into equal subsets (that is, each subset contains the same number of processing elements). The number of subsets is determined by the

Copyright © 1993 Thinking Machines Corporation

input value of **numproc**, which must be either ≤ 0 or a positive power of 2 that is less than or equal to the number of processing elements. These values have the following effects:

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

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

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

Upon completion, **CMSSL_partition_mesh** returns the number of subsets in **numproc**, the number of partitions in **numpar**, and the number of elements per partition (k, in step 2 above) in **nelpar**. (On machines with vector units, **nelpar** is actually the *largest* number of elements each partition can contain.)

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

Copyright © 1993 Thinking Machines Corporation

Example

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

- On a machine *without* vector units (where the multiple-of-eight rule is not in effect), there are two mesh elements in each of 100 processing elements (0 through 99); the one remaining mesh element resides in the processing element 100 (see Figure 22).

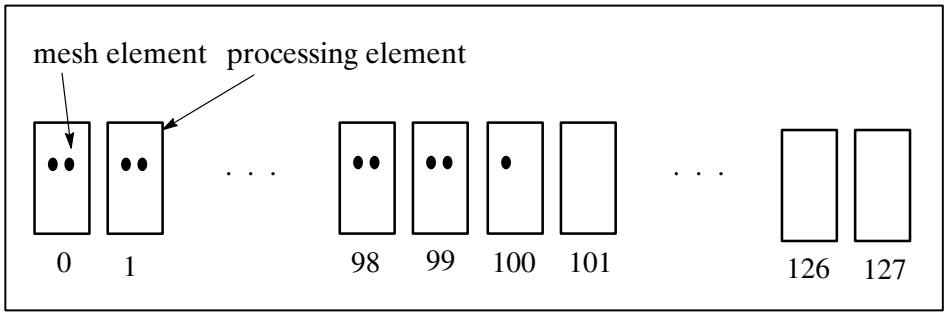

Figure 22. Mesh with 201 elements in machine configuration with 128 processing elements (SPARC nodes) and no vector units.

- On a machine *with* vector units, where the multiple-of-eight rule is in effect, there are eight mesh elements in each of 25 processing elements (0 through 24); the one remaining mesh element resides in the processing element 25 (see Figure 23).

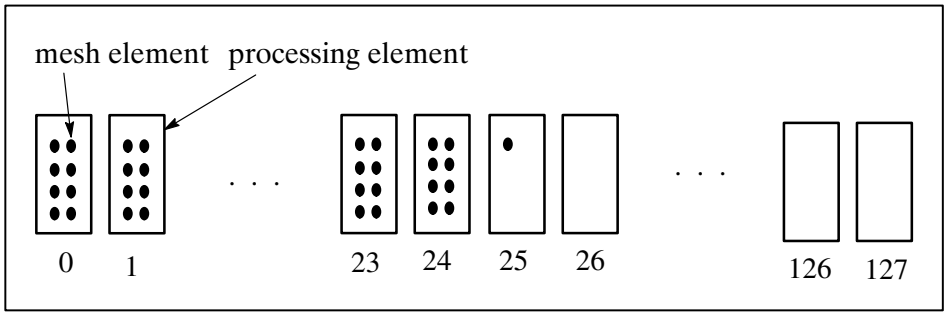

Figure 23. Mesh with 201 elements in machine configuration with 128 processing elements (vector units).

Copyright © 1993 Thinking Machines Corporation

If you set **numproc** $\leq$ 0 on input, then there are 128 subsets, each consisting of a single processing element. In the non-vector-unit case (Figure 22), steps 2 and 3 result in 101 partitions; **CMSSL_partition_mesh** returns **numproc** = 128, **numpar** = 101, and **nelpar** = 2. In the vector-unit case (Figure 23), steps 2 and 3 result in 26 partitions; **CMSSL_partition_mesh** returns **numproc** = 128, **numpar** = 26, and **nelpar** = 8.

On the other hand, if you set **numproc** = 4, **CMSSL_partition_mesh** divides the processing elements into 4 subsets, each with 32 processing elements. Figure 24 shows the subsets and partitions resulting from steps 2 and 3 in the non-vector-unit case. Upon completion, **CMSSL_partition_mesh** returns **numproc** = 4, **numpar** = 4, and **nelpar** = 64.

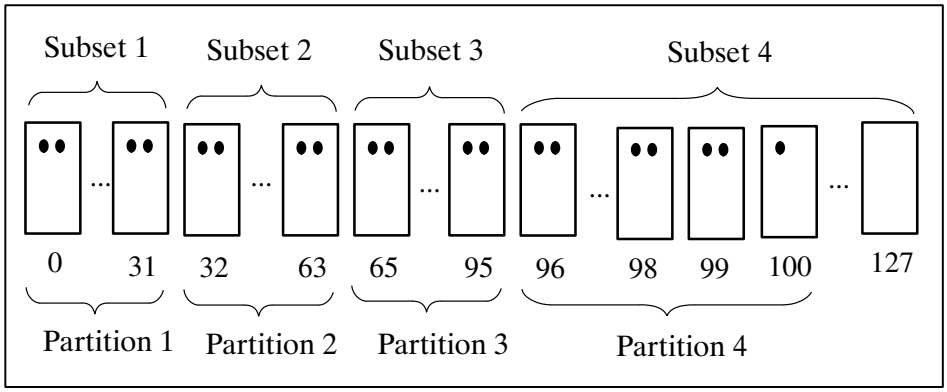

Figure 24. Subsets and partitions with no vector units.

Figure 25 shows the subsets and partitions resulting from steps 2 and 3 in the vector-unit case. Upon completion, **CMSSL_partition_mesh** returns **numproc** = 4, **numpar** = 1, and **nelpar** = 256. (In this case, since the multiple-of-eight rule is in effect, **nelpar** is the largest number of elements each partition can contain, even though the mesh does not have enough elements to fill one partition.)

Copyright © 1993 Thinking Machines Corporation

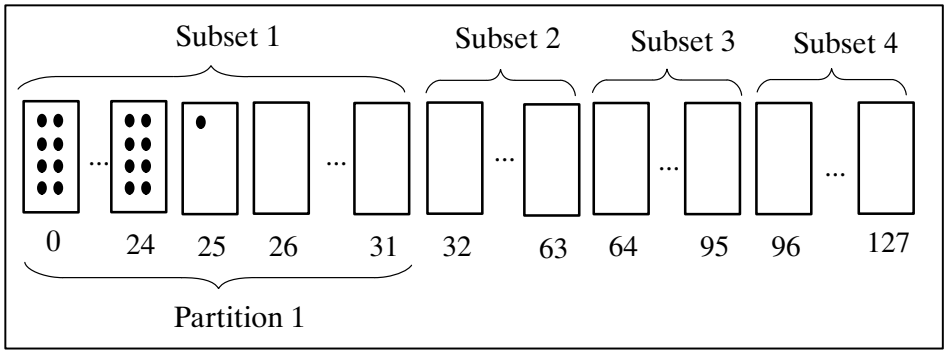

Figure 25. Subsets and (single) partition with vector units.

Implementation and Statistics

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

6.8.4 The Partitioning Permutation

The **CMSSL_partition_mesh** routine generates a permutation of the mesh element numbers, and returns the permutation in the parallel variable `q`. When element m is replaced with element `q`(m), effectively the nodes of the mesh remain unchanged, but the elements are reordered. Reordered elements 0 through `nelpar`−1 comprise the first partition, elements `nelpar` through 2*`nelpar`−1 comprise the second partition, and so on.

Example

Suppose you supply **CMSSL_partition_mesh** with `numproc` = 0 and with `idual` values that represent the 11-element mesh shown in Figure 26; and suppose the current machine configuration has four processing elements. Based on the partitioning principles discussed above, **CMSSL_partition_mesh**

Copyright © 1993 Thinking Machines Corporation

determines that four partitions are required; they will hold elements 0−2, 3−5, 6−8, and 9−10, respectively. The **CMSSL_partition_mesh** routine might return the following values of `q`:

`q`(0) = 0 `q`(6) = 1
`q`(1) = 9 `q`(7) = 3
`q`(2) = 6 `q`(8) = 7
`q`(3) = 2 `q`(9) = 8
`q`(4) = 10 `q`(10) = 4
`q`(5) = 5

This permutation results in the reordered mesh shown in Figure 26. Heavy lines denote partition boundaries.

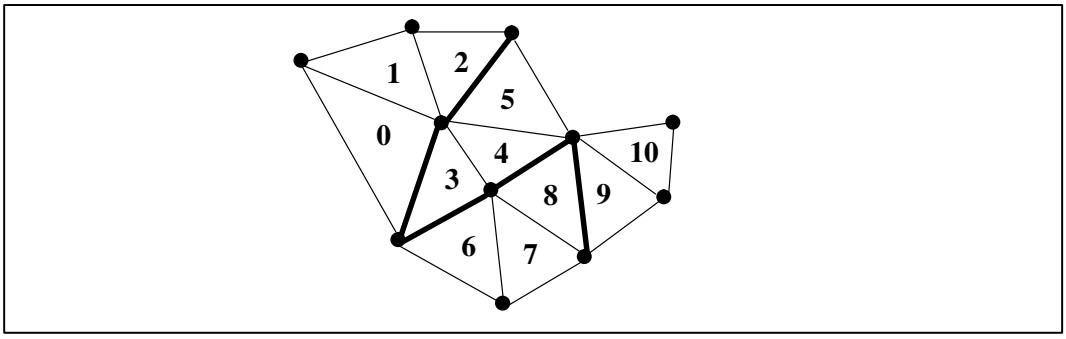

Figure 26. Partitions of a reordered mesh.

6.8.5 Reordering a Mapping

The **CMSSL_reorder_mapping** routine reorders a given parallel variable, `p`, containing a mapping, along its last axis using a permutation `q`. The parallel variable `q` must have rank 1 and length equal to the length of the last axis of `p`, and must be a permutation of `p` along its last axis.

If `p` has rank s, **CMSSL_reorder_mapping** replaces element $[p_1][p_2][p_3] \ldots [p_s]$`p` with element $[p_1][p_2][p_3] \ldots [[p_s]$`q`$]$`p`. The reordered mapping can then be supplied to the setup routines for the partitioned gather or scatter (see Sections 6.9 and 6.10).

Copyright © 1993 Thinking Machines Corporation

6.8.6 Renumbering Mappings

The **CMSSL_renumber_mapping** routine changes the values of a parallel variable, **p**, containing a mapping, to improve data locality. That is, **CMSSL_renumber_mapping** performs the operation

```
with(shapeof(p))
where (mask) p = [p]r;
```

where **p** is a parallel variable containing a mapping that has been reordered by **CMSSL_reorder_mapping**, **r** is an output parallel variable containing the renumbering mapping, and **mask** is a parallel variable used to mask out the elements of **p** whose values are less than 0. (These elements would also be masked when **p** is supplied to the partitioned gather and scatter utilities.)

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

NOTE

You must call **CMSSL_reorder_mapping** before calling **renumber_mapping**.

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

1. Call **CMSSL_renumber_mapping**, which renumbers **p** and returns **r**.
2. Reorder $\mathbf{x_0}$, your initial source for the gather operation, by mapping $\mathbf{x_0}$ to $[\mathbf{r}]\mathbf{x_0}$.
3. Perform partitioned gather and scatter operations.
4. Reorder your final parallel variable, $\mathbf{x_N}$, by mapping $[\mathbf{r}]\mathbf{x_N}$ to $\mathbf{x_N}$.

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

Partitioning of an Unstructured Mesh and Reordering of Mappings

The routines described below allow you to reorder a mapping derived from an unstructured mesh so that the communication required by subsequent gather and scatter operations is reduced.

SYNTAX

```
#include <cm/cmssl.h>

int CMSSL_generate_dual(
    int:void *idual, int:void *ien,
    CMSSL_part_elt_type_t elem_type);

int CMSSL_partition_mesh(
    int:void *idual, int:void *q, int *numproc, int *numpar,
    int *nelpar, CMSSL_part_storage_t storage_option,
    CMSSL_logical_t verbose);

void CMSSL_reorder_mapping(
    int:void *p, int:void *q);

void CMSSL_renumber_mapping(
    int:void *p, int:void *r, CMSSL_logical_t:void *mask);
```

ARGUMENTS

idual Pointer to a zero-based parallel variable that describes an unstructured mesh. Must have rank 2 and dimensions [*nface*][*nel*], where *nface* is the maximum number of faces per element and *nel* is the number of elements in the mesh. These *dual connectivity* values are defined by

idual(m, n) The element that shares face m with element n;
idual(m, n) -1 if face m of element n is a boundary of the mesh.

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

Copyright © 1993 Thinking Machines Corporation

When you call **CMSSL_partition_mesh**, **idual** is an input argument; it contains the dual connectivity values that represent the mesh you want to partition.

ien Input parameter. Pointer to a parallel variable that describes an unstructured mesh. This variable must be at least 0-based, with rank 2 and dimensions [*nnode*][*nel*], where *nnode* is the maximum number of nodes per element and *nel* is the number of elements in the mesh. These *element nodes* values are defined by

ien(*m*, *n*) The node number of the *m*th node of the *n*th mesh element.

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

elem_type Specifies the type of elements in the mesh whose **ien** values you are providing, as follows:

```
typedef enum CMSSL_part_elt_type_e {
  CMSSL_part_triangle = 0,
  CMSSL_part_quadrilateral = 1,
  CMSSL_part_tetrahedron = 2,
  CMSSL_part_pyramid = 3,
  CMSSL_part_prism = 4,
  CMSSL_part_hexahedron = 5,
  CMSSL_part_elt_type_limit = 6
} CMSSL_part_elt_type_t;
```

q When you call **CMSSL_partition_mesh**, **q** is an output parameter and is a pointer to a parallel variable of rank 1 and length equal to the number of elements in the mesh you want to partition. Upon successful completion, **q** contains a permutation that reorders the mesh into partitions. When element *n* is replaced with element [*n*]**q**, effectively the nodes of the mesh remain unchanged, but the elements are reordered. Reordered elements 0 through **nelpar**−1 comprise the first partition, elements **nelpar** through 2*__nelpar__−1 comprise the second partition, and so on.

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

When you call **CMSSL_reorder_mapping**, **q** is an input parameter and points to a parallel variable of rank 1 and length equal to the length of the last axis of **p**; **q** must be a permutation of **p** along its last axis.

p Pointer to a parallel variable of any shape.

On input to **CMSSL_reorder_mapping**, **p** contains a mapping. On successful completion of **CMSSL_reorder_mapping**, **p** contains the reordered mapping. If **p** has rank *s*, **CMSSL_reorder_mapping** replaces element $[p_1][p_2][p_3]$ ... $[p_s]$**p** with element $[p_1][p_2][p_3]$ ...$[[p_s]$**q**]**p**.

On input to **CMSSL_renumber_mapping**, **p** contains a mapping that has already been reordered by **CMSSL_reorder_mapping**. On successful completion of **CMSSL_renumber_mapping**, for each element $[p_1][p_2][p_3]$ ... $[p_s]$**p** for which **mask** is **CMSSL_true**, $[p_1][p_2][p_3]$ ... $[p_s]$**p** is replaced by $[[p_1][p_2][p_3]$ ... $[p_s]$**p**]**r**.

r Pointer to a parallel variable of rank 1 and length equal to the maximum value of **p**. Upon successful completion of **renumber_mapping**, **r** contains the mapping that **CMSSL_renumber_mapping** has used to renumber **p** for improved data locality.

mask Pointer to a parallel variable of the same shape as **p**. Used to mask out the elements of **p** whose values are less than 0. Set to **CMSSL_false** any values you want to mask out; set all other values to **CMSSL_true**. The type **CMSSL_logical_t** is defined in **<cm/cmssl.h>**.

numproc Pointer to a scalar variable. On input, this variable must be either ≤ 0 or a positive power of 2 that is less than or equal to the number of processing elements. When **CMSSL_partition_mesh** divides the processing elements into subsets, as discussed in the Description section, the input value of **numproc** determines the number of subsets, as follows:

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

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

Copyright © 1993 Thinking Machines Corporation

numpar Pointer to a scalar variable that contains, on return, the number of partitions formed by the reordered mesh.

nelpar Pointer to a scalar variable. On return, on machines without vector units, this variable contains the number of elements in each partition (with the possible exception of the last partition, which may contain fewer elements). On machines with vector units, **nelpar** contains the *largest* number of elements each partition can contain.

storage_option The legal values are listed below. The low-storage option is slower than the default option, but uses less working storage. In particular, the default option uses approximately (309 + 3*nface*) doublewords per mesh element, and the low-storage option uses approximately (11 + 3*nface*) doublewords per mesh element. These storage evaluations are not exact, but very closely approximate the actual memory usage.

```
typedef enum CMSSL_part_storage_e {
  CMSSL_part_low_storage = 0,
  CMSSL_part_default = 1
} CMSSL_part_storage_t;
```

verbose If set to **CMSSL_true**, **verbose** causes **CMSSL_partition_mesh** to print statistics to standard output at each step of the partitioning algorithm. For details, see the Description section. Set verbose to **CMSSL_false** to suppress the printing of statistics. The type **CMSSL_logical_t** is defined in **<cm/cmssl.h>**.

RETURNED VALUE

CMSSL_generate_dual returns one of the following codes:

0	Successful return.
−1	Invalid **elem_type**.
−3	**ien** is not 0-based.
−4	The mesh topology defined by **ien** is incorrect (for example, more than two elements share a face).

CMSSL_partition_mesh, returns one of the following codes:

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

0	Successful return.
−1	The second dimension of **idual** is not equal to the length of **q**.
−2	**numproc** is invalid on input (must be either $\leq$ 0 or a positive power of 2 that is less than or equal to the number of processing elements).

DESCRIPTION

The routines described here perform the following operations:

CMSSL_generate_dual

Given a parallel variable **ien** containing *element nodes* values that describe an unstructured mesh, this routine computes the corresponding *dual connectivity* values and places them in the parallel variable **idual**.

CMSSL_partition_mesh

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

CMSSL_reorder_mapping

Given a parallel variable **p** containing a mapping, and a permutation **q** (for example, the permutation returned by **partition_mesh**), this routine reorders **p** along its last axis using **q**.

CMSSL_renumber_mapping

Given a mapping **p** that has been reordered by **CMSSL_reorder_mapping**, this routine changes the supplied mapping values to improve data locality and returns the renumbering mapping.

Mixed Element Types. In the current implementation, the **CMSSL_generate_dual** routine requires that all elements of the mesh have the same type (the same number of nodes). However, you can work with a mesh with different element types when you call **CMSSL_partition_mesh**, by setting the extra faces of the input variable **idual** to −1. For example, to supply **CMSSL_partition_mesh** with a mesh with both triangles (3 faces) and quadrilaterals (4 faces), we set *nface* to 4 (the larger of the two), and set [3][*n*]**idual** to −1 if element *n* is a triangle.

Copyright © 1993 Thinking Machines Corporation

Partitioning Rules. The **CMSSL_partition_mesh** routine determines the number of partitions based on the number of mesh elements and the input value of **numproc**, as described below. Note that the term *processing element* refers to a vector unit or, on machines without vector units, a (SPARC) processing node.

The elements of the mesh (represented by the parallel variable **idual**) are laid out on the processing elements by the compiler; they span as many processing elements as possible. (On machines with vector units, the layout is subject to the multiple-of-eight rule — that is, the per-processor subgrid size must be a multiple of eight; the compiler pads the subgrid with garbage elements to reach a legal size.) Partitioning is accomplished as follows:

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

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

    - If **numproc** $= n > 0$, there are n subsets. If n is strictly less than the number of processing elements, then each subset contains multiple processing elements.

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

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

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

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

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

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

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

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

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

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

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

Statistics:  48  0.24599E-13  0.60988E-03
Eigenvalues:  0.11724E-01  0.10398E+00

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

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

- `Statistics:`

Copyright © 1993 Thinking Machines Corporation

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

- `Eigenvalues`:

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

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

- `Number of graph edges:`

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

- `Number of graph edge cuts:`

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

- `Percentage of cuts:`

  This number is

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

NOTES

Inactive Elements. If **`idual`** is larger than the number of active elements, set the inactive components of **`idual`** to −1.

Use of the Renumbering Routine. It is recommended that you call **renumber_ mapping** before calling the partitioned gather and scatter utilities. (You must call **CMSSL_reorder_mapping** before calling **CMSSL_renumber_mapping**.) If you find that using **renumber_mapping** does not enhance performance, you may omit it.

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

1. Call **CMSSL_renumber_mapping**, which renumbers **`p`** and returns **`r`**.
2. Reorder $\mathbf{x_0}$, your initial source for the gather operation, by mapping $\mathbf{x_0}$ to $[\mathbf{r}]\mathbf{x_0}$.

Copyright © 1993 Thinking Machines Corporation

3. Perform partitioned gather and scatter operations.

4. Reorder your final parallel variable, $\mathbf{x_N}$, by mapping $[\mathbf{r}]\mathbf{x_N}$ to $\mathbf{x_N}$.

EXAMPLES

Sample C* code that uses the **CMSSL_generate_dual**, **CMSSL_partition_mesh**, **CMSSL_reorder_mapping**, and **CMSSL_renumber_mapping** routines can be found on-line in the subdirectory

```
partitioning/cstar
```

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

Copyright © 1993 Thinking Machines Corporation

6.9 Partitioned Gather Utility

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

If your mapping is derived from a mesh, you can produce a partitioned mapping using the **CMSSL_partition_mesh** and **CMSSL_reorder_mapping** routines described in Section 6.8.

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

Copyright © 1993 Thinking Machines Corporation

Partitioned Gather Utility

Given three parallel variables (a source with rank 1 or 2, a destination with any rank, and a mapping specifying a gathering pattern), the routines described below gather elements or vectors from the source into the destination. If the mapping has been reordered to achieve data locality, communication time is reduced.

SYNTAX

```
#include <cm/cmssl.h>

int CMSSL_part_gather_setup(
    int:void *p, CMSSL_logical_t:void *y_mask,
    ifdcz:void *x_template, int *setup);

void CMSSL_part_gather(
    ifdcz:void *y, ifdcz:void *x, CMSSL_logical_t:void *y_mask,
    int setup);

void CMSSL_part_vector_gather(
    ifdcz:void *y, ifdcz:void *x, CMSSL_logical_t:void *y_mask,
    int y_vector_axis, int x_vector_axis, int setup);

void CMSSL_deallocate_part_gather_setup(
    int setup);
```

OVERLOADED DATA TYPES

ifdcz **int**, **float**, **double**, **CMSSL_complex_t**, or **CMSSL_dcomplex_t**

In a given **CMSSL_part_gather** or **CMSSL_part_vector_gather** call, **x** and **y** must have the same data type (which need not match that of **x_template**). The **CMSSL_complex_t**, **CMSSL_dcomplex_t**, and **CMSSL_logical_t** data types are defined in the header file **<cm/cmssl.h>**.

ARGUMENTS

p Pointer to a parallel variable whose dimensions must satisfy the following conditions:

- **p** must have the same shape as the **y** you supply in any subsequent associated call to **CMSSL_part_gather**.

Copyright © 1993 Thinking Machines Corporation

- **p** must have the same shape as the subset, formed by omitting axis **y_vector_axis**, of the **y** you supply in any subsequent associated call to **CMSSL_part_vector_gather**.

The parallel variable **p** contains a mapping that indicates the gathering pattern, as follows. If $[p_1]...[p_k]$**y_mask** = **CMSSL_true** and element $[p_1]...[p_k]$**p** contains the value n, then

- In a subsequent call to **CMSSL_part_gather**, element $[n]$**x** is gathered from the source to location $[p_1]...[p_k]$**y** in the destination during the gather operation.

- In a subsequent call to **CMSSL_part_vector_gather**, the vector [.]$[n]$**x** or $[n]$[.]**x** is gathered from the source to locations $[p_1]...[.]...[p_k]$**y** in the destination during the gather operation (where [.] denotes axis **x_vector_axis** of **x** and axis **y_vector_axis** of **y**).

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

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

y_mask Pointer to a parallel variable with the same shape as

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

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

Set to **CMSSL_true** the elements that correspond to active elements of **y**. The contents of **y_mask** remain unchanged by **CMSSL_part_gather_setup**, **CMSSL_part_gather**, and **CMSSL_part_vector_gather**.

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

x_template Pointer to a parallel variable with the same shape as

Copyright © 1993 Thinking Machines Corporation

- The **x** you supply in any subsequent associated call to **CMSSL_part_gather**, or
- The subset, formed by omitting axis **x_vector_axis**, of the **x** you supply in any subsequent associated call to **CMSSL_part_vector_gather**.

The setup routine uses only the shape of this variable, ignoring the contents.

y Pointer to a parallel variable with arbitrary shape. The destination to which elements or vectors from the source are gathered. The values of active elements of **y** are overwritten.

When you call **CMSSL_part_gather**, **y** must have the same shape as the **p** you supplied in the **CMSSL_part_gather_setup** call.

When you call **CMSSL_part_vector_gather**, you must declare **y** with an extra local axis (axis **y_vector_axis**) with the same length as axis **x_vector_axis** of **x**. Upon return from **CMSSL_part_vector_gather**, the gathered vectors will lie along this axis of **y**. The remaining portion of **y** must have the same shape as the corresponding portion of the **p** you supplied in the **CMSSL_part_gather_setup** call.

If you use these routines in conjunction with the **CMSSL_partition_mesh** routine, you should declare all axes of **y** to be local except the reordered axis, which should be non-local. This strategy achieves maximum locality and yields the best performance.

x Pointer to a parallel variable of rank 1 (in a call to **CMSSL_part_gather**) or 2 (in a call to **CMSSL_part_vector_gather**). The source from which elements or vectors are gathered. In a call to **CMSSL_part_vector_gather**, the vectors to be gathered must lie along axis **x_vector_axis** of **x**. The contents of **x** remain unchanged by **CMSSL_part_gather** and **CMSSL_part_vector_gather**.

y_vector_axis Specifies the axis of **y** along which the gathered vectors will lie. This axis must be zero-based and local to a processing element, and must have the same length as axis **x_vector_axis** of **x**.

Copyright © 1993 Thinking Machines Corporation

x_vector_axis Specifies the axis of **x** along which the vectors to be gathered must lie. This axis must be zero-based and local to a processing element, and must have the same length as axis **y_vector_axis** of **y**.

setup Internal variable (pointer to an internal variable, when you call the setup routine). You must supply **CMSSL_part_gather**, **CMSSL_part_vector_gather**, and **CMSSL_deallocate_part_gather_setup** with the value that **CMSSL_part_gather_setup** assigns to **setup**.

RETURNED VALUE

CMSSL_part_gather_setup returns 0 on successful return.

DESCRIPTION

Definition. The gather operation is defined by

```
with(shapeof(y))
where (y_mask)    y = [p]x;
```

and the vector gather operation, by

```
with shapeof(p)
   where (y_mask)
      for (i=0; i<dimof(y,y_vector_axis); i++)
         [.]...[i]...[.]y = [i][p]x;
```

where **x** is the source from which vectors are being gathered, **y** is the destination, **p** contains a mapping, **y_mask** is a mask for the destination, and the axis denoted by **[i]** in **y** is axis **y_vector_axis**.

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

1. Call **CMSSL_part_gather_setup**.
2. Call **CMSSL_part_gather** or **CMSSL_part_vector_gather**.

   To perform more than one gather operation using the same sparsity (gathering mapping), follow one call to **CMSSL_part_gather_setup** with multiple calls to **CMSSL_part_gather** and/or **CMSSL_part_vector_gather**. (You may intersperse calls to these two routines.) If the sparsity changes, start with Step 1 again.

Copyright © 1993 Thinking Machines Corporation

3. After all **CMSSL_part_gather** and **CMSSL_part_vector_gather** calls associated with the same **CMSSL_part_gather_setup** call, call **CMSSL_deallocate_part_gather_setup** to deallocate the processing element storage space required by the setup routine.

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

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

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

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

As long as the arguments supplied to **CMSSL_part_gather_setup** remain the same (except that **y_mask** may change — see the arguments list), the application can call **CMSSL_part_gather** and/or **CMSSL_part_vector_gather** multiple times following one call to **CMSSL_part_gather_setup**, each time supplying **setup**, a mask, and a source, **x**, and receiving in return a destination, **y**.

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

NOTES

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

Copyright © 1993 Thinking Machines Corporation

Permutation of the Source. Some applications require permutation of the source prior to the gather operation. This permutation is the responsibility of the user application and is not performed by **CMSSL_part_gather_setup**, **CMSSL_part_gather**, or **CMSSL_part_vector_gather**.

Local Axes. To ensure that an axis is local to a processing element, you may need to use the **`allocate_detailed_shape`** function, which is described in the *C* Programming Guide*.

Layout and Performance. For best performance, axes **`x_vector_axis`** and **`y_vector_axis`** should be either the left-most or the right-most axes of their respective parallel variables. If you use these routines in conjunction with the **CMSSL_partition_mesh** routine, you should use **`allocate_detailed_shape`** to ensure that all axes of **`y`** are local except the reordered axis, which should be non-local. This strategy achieves maximum locality and yields the best performance.

EXAMPLES

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

```
partitioning/cstar/
```

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

Copyright © 1993 Thinking Machines Corporation

6.10 Partitioned Scatter Utility

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

If your mapping is derived from a mesh, you can produce a partitioned mapping using the **CMSSL_partition_mesh** and **CMSSL_reorder_mapping** routines described in Section 6.8.

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

Copyright © 1993 Thinking Machines Corporation

Partitioned Scatter Utility

Given three parallel variables (a source with any rank, a destination with rank 1 or 2, and a mapping specifying a scattering pattern), the routines described below scatter elements or vectors from the source to the destination. If the mapping has been reordered to achieve data locality, communication time is reduced.

SYNTAX

```
#include <cm/cmssl.h>

int CMSSL_part_scatter_setup(
    int:void *p, CMSSL_logical_t:void *x_mask,
    ifdcz:void *y_template, int *setup);

void CMSSL_part_scatter(
    ifdcz:void *y, ifdcz:void *x, CMSSL_logical_t:void *x_mask,
    int setup);

void CMSSL_part_vector_scatter(
    ifdcz:void *y, ifdcz:void *x, CMSSL_logical_t:void *x_mask,
    int y_vector_axis, int x_vector_axis, int setup);

void CMSSL_deallocate_part_scatter_setup(
    int setup);
```

OVERLOADED DATA TYPES

ifdcz **int**, **float**, **double**, **CMSSL_complex_t**, or **CMSSL_dcomplex_t**

In a given **CMSSL_part_scatter** or **CMSSL_part_vector_scatter** call, **x** and **y** must have the same data type (which need not match that of **y_template**). The **CMSSL_complex_t**, **CMSSL_dcomplex_t**, and **CMSSL_logical_t** data types are defined in the header file **<cm/cmssl.h>**.

ARGUMENTS

p Pointer to a parallel variable whose dimensions must satisfy the following conditions:

- **p** must have the same shape as the **x** you supply in any subsequent associated call to **CMSSL_part_scatter**.

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

- **p** must have the same shape as the subset, formed by omitting axis **x_vector_axis**, of the **x** you supply in any subsequent associated call to **CMSSL_part_vector_scatter**.

The parallel variable **p** contains a mapping that indicates the scattering pattern, as follows. If $[p_1]...[p_k]$**x_mask** = **CMSSL_true** and element $[p_1]...[p_k]$**p** contains the value n, then

- A subsequent call to **CMSSL_part_scatter** adds element $[p_1]...[p_k]$**x** of the source to destination element $[n]$**y**.
- A subsequent call to **CMSSL_part_vector_scatter** adds the vector $[p_1]...[.]...[p_k]$**x** to the vector $[.][n]$**y** or $[n][.]$**y** within the destination (where [.] denotes axis **x_vector_axis** of **x** and axis **y_vector_axis** of **y**).

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

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

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

x_mask Pointer to a parallel variable with the same shape as

- The **x** you supply in any subsequent associated call to **CMSSL_part_scatter**, or
- The subset, formed by omitting axis **x_vector_axis**, of the **x** you supply in any subsequent associated call to **CMSSL_part_vector_scatter**.

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

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

Copyright © 1993 Thinking Machines Corporation

y_template Pointer to a parallel variable with the same shape as

- The **y** you supply in any subsequent associated call to **CMSSL_part_scatter**, or
- The subset, formed by omitting axis **y_vector_axis**, of the **y** you supply in any subsequent associated call to **CMSSL_part_vector_scatter**.

The setup routine uses only the shape of this variable, ignoring the contents.

y Pointer to a parallel variable of rank 1 (in a call to **CMSSL_part_scatter**) or 2 (in a call to **CMSSL_part_vector_scatter**). The **CMSSL_part_scatter** routine adds the scattered elements from **x** to the initial values of **y**. The **CMSSL_part_vector_scatter** routine adds the scattered vectors from **x** to the initial values of the vectors that lie along axis **y_vector_axis** of **y**. This axis must be local to a processing element, and must have the same length as axis **x_vector_axis** of **x**.

x Pointer to a parallel variable of arbitrary shape. The source from which elements or vectors are scattered. The contents of **x** remain unchanged by **CMSSL_part_scatter** and **CMSSL_part_vector_scatter**.

When you call **CMSSL_part_scatter**, **x** must have the same shape as the **p** you supplied in the **CMSSL_part_scatter_setup** call.

When you call **CMSSL_part_vector_scatter**, you must declare **x** with an extra local axis (axis **x_vector_axis**) with the same length as axis **y_vector_axis** of **y**. The vectors to be scattered must lie along this axis of **x**. The remaining portion of **x** must have the same shape as the corresponding portion of the **p** you supplied in the **CMSSL_part_scatter_setup** call.

If you use these routines in conjunction with the **CMSSL_partition_mesh** routine, you should declare all axes of **x** to be local except the reordered axis, which should be non-local. This strategy achieves maximum locality and yields the best performance.

y_vector_axis Specifies the axis of **y** along which the scattered vectors will lie. This axis must be zero-based and local to a processing element, and must have the same length as axis **x_vector_axis** of **x**.

Copyright © 1993 Thinking Machines Corporation

x_vector_axis Specifies the axis of **x** along which the vectors to be scattered must lie. This axis must be zero-based and local to a processing element, and must have the same length as axis **y_vector_axis** of **y**.

setup Internal variable (pointer to an internal variable, when you call the setup routine). You must supply **CMSSL_part_scatter**, **CMSSL_part_vector_scatter**, and **CMSSL_deallocate_part_scatter_setup** with the value assigned to **setup** by **CMSSL_part_scatter_setup**.

RETURNED VALUE

CMSSL_part_scatter_setup returns 0 on successful return.

DESCRIPTION

Definition. The scatter operation is defined by

```
with(shapeof(x))
where (x_mask)     [p]y += x;
```

and the vector scatter operation, by

```
with(shapeof(p))
   where (x_mask)
      for (i=0; i<dimof(x, x_vector_axis); i++)
         [i][p]y += [.]...[i]...[.]x;
```

where **x** is the source from which vectors are being scattered, **y** is the destination, **p** contains a mapping, **x_mask** is a mask for the source, and the axis denoted by **[i]** in **x** is axis **x_vector_axis**.

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

1. Call **CMSSL_part_scatter_setup**.
2. Call **CMSSL_part_scatter** or **CMSSL_part_vector_scatter**.

   To perform more than one scatter operation using the same sparsity (scattering mapping), follow one call to **CMSSL_part_scatter_setup** with multiple calls to **CMSSL_part_scatter** and/or **CMSSL_part_vector_scatter**. (You may intersperse calls to these two routines.) If the sparsity changes, start with Step 1 again.

Copyright © 1993 Thinking Machines Corporation

3. After all **CMSSL_part_scatter** and **CMSSL_part_vector_scatter** calls associated with the same **CMSSL_part_scatter_setup** call, call the **CMSSL_deallocate_part_scatter_setup** routine to deallocate the processing element storage space required by the setup routine.

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

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

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

As long as the arguments supplied to **CMSSL_part_scatter_setup** remain the same (except that **x_mask** may change — see the arguments list), the application can call **CMSSL_part_scatter** and/or **CMSSL_part_vector_scatter** multiple times following one **CMSSL_part_scatter_setup** call, each time supplying a source, **x**, and a mask, **x_mask**, and receiving in return a destination, **y**.

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

NOTES

Local Axes. To ensure that an axis is local to a processing element, you may need to use the **allocate_detailed_shape** function, which is described in the *C* Programming Guide*.

Copyright © 1993 Thinking Machines Corporation

Layout and Performance. For best performance, axes **`x_vector_axis`** and **`y_vector_axis`** should be either the left-most or the right-most axes of their respective parallel variables. If you use these routines in conjunction with the **CMSSL_partition_mesh** routine, you should use **`allocate_detailed_shape`** to ensure that all axes of **`x`** are local except the reordered axis, which should be non-local. This strategy achieves maximum locality and yields the best performance.

EXAMPLES

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

```
partitioning/cstar/
```

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

Copyright © 1993 Thinking Machines Corporation

6.11 Vector Move (Extract and Deposit)

The **CMSSL_vector_move** routine moves a vector from each subgrid of a source parallel variable to a vector in a corresponding subgrid of a destination parallel variable. The source and destination must have the same rank, type, and processing element layout. The **CMSSL_vector_move_utils** routine returns processing element layout and subgrid shape information for any parallel variable.

Details about these routines are provided in the man page that follows.

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

Vector Move (Extract and Deposit)

The **CMSSL_vector_move** routine moves a vector from each subgrid of a source parallel variable to a vector in a corresponding subgrid of a destination parallel variable. The source and destination must have the same rank, type, and processing element layout. The **CMSSL_vector_move_utils** routine returns processing element layout and subgrid shape information for any parallel variable.

SYNTAX

```
#include <cm/cmssl.h>

int CMSSL_vector_move (
    ifdcz:void *y, ifdcz:void *x, int *y_coords, int y_axis,
    int *x_coords, int x_axis);

int CMSSL_vector_move_utils (
    ifdcz:void *template, int *subgrid_lengths,
    int *num_procs);
```

OVERLOADED DATA TYPES

ifdcz **int**, **float**, **double**, **CMSSL_complex_t**, or **CMSSL_dcomplex_t**

In a given **CMSSL_vector_move** call, **x** and **y** must have the same data type. The **CMSSL_complex_t** and **CMSSL_dcomplex_t** data types are defined in the header file **<cm/cmssl.h>**.

ARGUMENTS

y Pointer to a parallel variable of any supported data type.

x Pointer to a parallel variable of the same rank and type as **y**.

The parallel variables **x** and **y** must have the same processing element layout; otherwise, **CMSSL_vector_move** returns the error code −2. *Processing element* refers to a SPARC processing node on a CM-5 without vector units, or a vector unit on a CM-5 with vector units. *Processing element layout* refers to the number of processing elements along each axis.

Copyright © 1993 Thinking Machines Corporation

The length of axis **x_axis** in **x** must equal the length of axis **y_axis** in **y**.

y_coords Pointer to a scalar array of rank 1 and length equal to the rank of **y**. Defines the destination vectors to which the source vectors will be moved. The destination vectors are formed by holding the local coordinate along each axis *i* of **y** constant at the value **y_coords**(*i*), except the coordinate corresponding to axis **y_axis**. The value of **y_coords**(**y_axis**) is ignored. The array **y_coords** must satisfy $0 \leq$ [*i*]**y_coords** < subgrid length of axis *i* of **y** for each axis of **y**.

y_axis Identifies the axis in **y** along which the destination vectors lie. Must satisfy $0 \leq$ **y_axis** < rank(**y**).

x_coords Pointer to a scalar array of rank 1 and length equal to the rank of **x**. Defines the source vectors to be moved to **y**. The source vectors are formed by holding the local coordinate along each axis *i* of **x** constant at the value **x_coords**(*i*), except the coordinate corresponding to axis **x_axis**. The value of **x_coords**(**x_axis**) is ignored. The array **x_coords** must satisfy $0 \leq$ [*i*]**x_coords** < subgrid length of axis *i* of **x** for each axis of **x**.

x_axis Identifies the axis in **x** along which the source vectors lie. Must satisfy $0 \leq$ **x_axis** < rank(**x**).

template Pointer to a parallel variable whose processing element layout and subgrid shape is required.

subgrid_lengths Pointer to a scalar array of rank 1 and length equal to the rank of **template**. On successful return, **subgrid_lengths**(*i*) contains the subgrid length of **template** along axis *i*.

num_procs Pointer to a scalar array of rank 1 and length equal to the rank of **template**. On successful return, **num_procs**(*i*) contains the number of processing elements along axis *i* of **template**.

RETURNED VALUE

These routines return one of the following codes:

Copyright © 1993 Thinking Machines Corporation

0	Successful return.
−1	**x** and **y** do not have the same rank.
−2	**x** and **y** do not have the same processing element layout.
−4	**y_coords** does not satisfy $0 \leq$ [*i*]**y_coords** < subgrid length of axis *i* of **y** for all axes of **y**; or **x_coords** does not satisfy $0 \leq$ [*i*]**x_coords** < subgrid length of axis *i* of **x** for all axes of **x**.

DESCRIPTION

The **CMSSL_vector_move** routine moves a vector from each subgrid of **x** to a vector in a corresponding subgrid of **y**. Each element in the destination vector is overwritten by the corresponding source value. The operation occurring within each subgrid can be expressed by

```
shape [dimof(y,y_axis)] sh;
with (sh)
([y_coords[0]]...[y_coords[y_axis]] [.] [y_coords[y_axis+1]]...[.]y=
[x_coords[0]]...[x_coords[x_axis]] [.] [x_coords[x_axis+1]]...[.]x)
```

The **CMSSL_vector_move_utils** routine returns processing element layout and subgrid shape information for any parallel variable. The **subgrid_lengths** argument returns the subgrid length of the parallel variable along each axis; the **num_procs** argument returns the number of processing elements along each axis. In this context, *processing element* refers to a (SPARC) processing node on a CM-5 without vector units, or a vector unit on a CM-5 with vector units.

The on-line examples show how to use the **CMSSL_vector_move** and **CMSSL_vector_move_utils** routines to extract and deposit rows and columns of a two-dimensional parallel variable, and how to use **CMSSL_vector_move** to extract a face from a three-dimensional parallel variable.

Consider a 64 x 64 source **x** and a 64 x 16 destination **y**. Further assume a 4 x 8 processing element layout underlying both **x** and **y**. The subgrid of **x** on each processing element is 16 x 8; the subgrid of **y** on each processing element is 16 x 2. Suppose we want to extract the third column of the subgrid of **x** on each processing element and deposit it in the second column of the subgrid of **y**. We use the **CMSSL_vector_move_utils** routine to get the processing element layout; use that information to define a shape; and proceed as indicated below:

Copyright © 1993 Thinking Machines Corporation

```
shape [64][8] sh;
with (sh)
    [.][1 + .*2]y = [.][2 + .*8]x;
```

The same functionality can be achieved by calling **CMSSL_vector_move** as follows:

```
int y_coords[2], x_coords[2];
int y_axis, x_axis;

y_coords[1] = 1;
x_coords[1] = 2;
y_axis = 0;
x_axis = 0;
ier = CMSSL_vector_move(y, x, y_coords, y_axis, x_coords, x_axis);
```

NOTES

Processing Element Layout. The processing element layout must be identical for **x** and **y**. An error is signalled if this is not the case. (*Processing element layout* refers to the number of processing elements along each axis.) Use the **allocate_detailed_shape** function (described in the *C* Programming Guide*) to ensure that the processing element layouts of **x** and **y** are identical. You can also obtain the processing element layout and subgrid shape using **CMSSL_vector_move_utils**.

EXAMPLES

Sample C* code that uses the extract and deposit routines can be found on-line in the subdirectory

vector-move/cstar/

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

Version 3.2 Beta, December 1993
Copyright © 1993 Thinking Machines Corporation

6.12 References

For information about the partitioning algorithm implemented on the CM, see the following references:

5. Pothen, A., H. D. Simon, and K.−P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications* **11** (1990): 430−52.

6. Simon, H. D. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering* **2** (1991): 135−48.

7. 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 © 1993 Thinking Machines Corporation