The
Connection Machine
System

CMSSL for CM Fortran: CM-5 Edition, Volume II

Preliminary Documentation for Version 3.1 Beta 2

January 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First preliminary edition for 3.1 Beta 2, January 1993

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1264
(617) 234-1000

Contents

Volume II

About This Manual

Objectives

This manual describes the CM Fortran programming interface to the Connection Machine Scientific Software Library (CMSSL).

This manual describes CMSSL software for the Connection Machine supercomputer, model CM-5. (Note that throughout this book, statements made about the CM-200 also apply to the CM-2, unless otherwise noted.)

Intended Audience

Anyone writing CM Fortran programs that use the CMSSL software should read this document.

Organization

This manual is divided into two volumes with fourteen chapters:

Volume I

Chapter 1	**Introduction to the CMSSL for CM Fortran** Describes the contents of the CMSSL. Discusses the data types supported and explains how to perform CMSSL operations on multiple independent data sets concurrently.
Chapter 2	**Using the CMSSL CM Fortran Interface** Explains how to include CMSSL routine definitions in CM Fortran code, and how to compile, link, and execute CM Fortran programs that call CMSSL routines.
Chapter 3	**Dense Matrix Operations** Describes the inner product, 2-norm, outer product, matrix vector multiplication, vector matrix multiplication, matrix multiplication, and infinity norm routines.
Chapter 4	**Sparse Matrix Operations** Describes the routines that perform arbitrary elementwise sparse matrix operations, arbitrary block sparse matrix operations, and grid sparse matrix operations.

Chapter 5 Linear Solvers for Dense Systems
Describes the the Gaussian elimination (*LU* decomposition) routines, routines that solve linear systems using Householder transformations (the *QR* routines), matrix inversion, and the Gauss-Jordan system solver.

Chapter 6 Linear Solvers for Banded Systems
Describes the banded system factorization and solver routines, which solve tridiagonal, block tridiagonal, pentadiagonal, and block pentadiagonal systems.

Chapter 7 Iterative Solvers
Describes routines that solve linear systems using Krylov space iterative methods.

Chapter 8 Eigensystem Analysis
Describes routines that perform eigensystem analysis of dense real symmetric tridiagonal systems, dense Hermitian systems, dense real symmetric systems, dense real systems, and sparse systems. Included are routines that use the Jacobi method, a *k*-step Lanczos method, and a *k*-step Arnoldi method. Also included are routines that reduce Hermitian matrices to real symmetric tridiagonal form (and perform the corresponding basis transformation).

Volume II

Chapter 9 Fast Fourier Transforms
Describes the simple and detailed complex-to-complex FFT routines.

Chapter 10 Ordinary Differential Equations
Describes routines that integrate ordinary differential equations (ODEs) explicitly using a fifth-order Runge-Kutta-Fehlberg formula.

Chapter 11 Linear Programming
Describes a routine that solves multi-dimensional minimization problems using the simplex linear programming method.

Chapter 12 Random Number Generators
Describes the Fast and VP random number generators.

Chapter 13 Statistical Analysis
Describes the histogram and range histogram routines.

Revision Information

This is the first edition of this manual.

Acknowledgments

The Center for Research on Parallel Computation at Rice University has contributed to this release of the CMSSL.

Notation Conventions

The table below displays the notation conventions used in this manual.

Convention	Meaning
`bold typewriter`	UNIX and CM System Software commands, command options, and file names.
boldface sans serif	CM Fortran language elements, such as function and subroutine names and constants, when they appear embedded in text or in syntax lines.
italics	Parameter names, when they appear embedded in text or syntax lines.
bold italics	CM arrays, when they appear embedded in text or syntax lines.
`typewriter`	Code examples and code fragments.

`%` **`bold typewriter`**
`typewriter`
	In interactive examples, user input is shown in **`bold typewriter`** and system output is shown in `regular typewriter` font.

Standard Abbreviations for Matrix Operations and Matrix Types

The following standard abbreviations are used in the CMSSL CM Fortran interfaces to identify matrix types. Further abbreviations will be introduced as more matrix types are supported.

CMSSL Matrix Type Abbreviations

dense general	**gen**
dense symmetric	**sym**
arbitrary elementwise sparse	**sparse**
arbitrary block sparse	**block_sparse**
grid sparse	**grid_sparse**
tridiagonal	**gen_tridiag**
pentadiagonal	**gen_pentadiag**
block tridiagonal	**block_tridiag**
block pentadiagonal	**block_pentadiag**

The following standard abbreviations are used in the CMSSL CM Fortran interfaces to identify matrix operations:

CMSSL Matrix Operation Abbreviations

factorization	**factor**
inversion	**invert**
multiplication	**mult**
solver	**solve**
polyshift	**pshift**

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1264
Internet Electronic Mail:	`customer-support@think.com`
uucp Electronic Mail:	`ames!think!customer-support`
Telephone:	(617) 234-4000 (617) 876-1111

Chapter 9

Fast Fourier Transforms

This chapter describes the CM Fortran interface to the CMSSL Fast Fourier Transforms (FFTs). One section is devoted to each of the following topics:

- the CMSSL FFT library calls
- the complex-to-complex FFT (CCFFT)
- references

This chapter assumes a basic understanding of Fourier Transforms.

NOTE

The complex-to-complex Fast Fourier Transform (FFT) included in this Beta release on the CM-5 is pre-release software. The performance of this pre-release FFT is very poor; it is included only to prevent existing code from breaking. A later release containing full support for the complex-to-complex FFT is planned. In addition, in this Beta release of the complex-to-complex FFT, whenever the operand array has a layout directive, an interface block is necessary; see Section 9.2.4 below for details and an example.

9.1 The CMSSL FFT Library Calls

The CMSSL provides two FFT user interfaces:

- The *Simple FFT.* This interface is used to transform a data set in the same direction along all axes. You can use the simple FFT to perform multidimensional FFTs, but not to perform multiple independent FFTs concurrently.
- The *Detailed FFT.* This interface provides (at the cost of added interface complexity) a great deal of flexibility, including support for multiple instances. For each axis of a multidimensional array, the Detailed FFT allows you to choose the following:
  - Whether a transform is performed along the axis, and if so, in which direction (forward or inverse).
  - The address ordering (normal or bit-reversed) used to store input and output data values along the axis.
  - The scaling factor (none, axis length, or square root of axis length) applied to the transform results. At the end of the FFT, each element of the array is divided by the product of the scaling factors. (The axes for which you specified "none" as the scaling factor do not contribute to this product.)

These features are described in the man pages later in this chapter.

To perform an FFT, you must follow these steps:

1. Call the **fft_setup** routine.

   This routine computes internal values (including twiddle factors), allocates an internal FFT setup object, and computes an *FFT setup ID*, which you must then use as an argument in all subsequent Simple FFT, Detailed FFT, and deallocation calls associated with this setup call.

2. Call the **fft** or **fft_detailed** routine.

   These routines execute the Simple and Detailed FFTs, respectively. You may follow one call to the setup routine with multiple calls to these routines. However, the result of a call to the setup routine can only be used for arrays with the same rank, axis extents, layout directives, and data type as the array supplied in the setup routine. If any of these parameters change, you must call **fft_setup** again to create another setup structure.

3. Call the **deallocate_fft_setup** routine.

   This routine deallocates a specified FFT setup strucure and frees the storage space it required.

You may have more than one setup ID active at a time; that is, you may call the setup routine more than once before deallocating any setup IDs. When you call **fft, fft_detailed**, or **deallocate_fft_setup**, you must be sure to provide the correct setup ID.

9.2 Complex-to-Complex FFT

The CMSSL complex-to-complex FFT is a Discrete Fourier Transform in which the same call is used to specify a one-dimensional or multidimensional transform on any legal array of floating-point data. For example, in Fourier Analysis Cyclic Reduction (FACR) for a three-dimensional problem (see references 1, 6, 16, and 18 in Section 9.3), the data array has at least three axes. You may choose to perform Fourier transformation on planes represented by two of the axes, while tridiagonal systems of equations are solved along the third axis.

9.2.1 Butterfly Computations, Twiddle Factors, and the CCFFT Setup Phase

The CMSSL FFT is of the Cooley-Tukey variety (see references 3 and 4). In the radix-2 algorithm of this type, there are $\log_2 N$ stages for a data set of size $N = 2^{\log_2 N}$. In each such data set, "butterfly" computations are performed on pairs of data; $N/2$ butterfly computations are performed in each stage. In a radix-4 algorithm, there are $\log_4 N$ stages, each of which involves butterfly computations on four data points; $N/4$ butterfly computations are performed in each stage. Higher-radix FFTs are defined similarly. The Discrete Fourier Transform relies on a set of coefficients known as *twiddle factors*. The twiddle factors are various roots of unity, and depend only upon the size of the data set and the radix. In the CMSSL CCFFT, the twiddle computations are performed during a setup phase and used during the subsequent evaluation phase. This organization allows you to amortize the expense of twiddle factor computation over several transformations.

The sign of the twiddle factors determines the direction of an FFT. In the CMSSL FFT, the *forward* transform is defined as one that uses twiddle factors with a *negative* exponent, and the *inverse* transform is defined as one that uses twiddle factors with a *positive* exponent.

9.2.2 Bit Ordering and Bit Reversal

The Cooley-Tukey FFT bit-reverses the addresses in which it stores results. For example, if an 8-point FFT is performed, the indices of the output are

0, 4, 2, 6, 1, 5, 3, 7

In binary, this sequence is

000, 100, 010, 110, 001, 101, 011, 111

that is, the normal sequence with bits reversed.

The Detailed FFT allows you to specify whether the address ordering of the input is normal or bit-reversed, and whether the desired address ordering of the result is normal or bit-reversed.

In the CCFFT, if you specify *opposite* address orderings for input and output, the Detailed FFT performs the single bit-reversal that is inherent in the Cooley-Tukey algorithm, and produces output that is bit-reversed relative to the input. If you specify the *same* address ordering for input and output, the Detailed FFT performs an *extra* bit reversal, and produces output with the same ordering as the input. The extra bit reversal exacts a performance cost.

The Simple FFT does not provide bit-reversal options; it always produces output data with the same bit ordering as the input data. It does this by automatically performing a bit-reversal on the transformed data, thus annihilating the bit-reversal inherent in the FFT. This feature causes a Simple FFT to take 20% to 50% more time than an identical Detailed FFT for which the input and output address orderings are opposite.

9.2.3 Multidimensional and Multiple-Instance FFTs

The CMSSL FFT allows you to specify whether you want to perform a forward transform, an inverse transform, or no transform along each axis of a multidimensional CM array. As described later in this chapter, it is also possible to perform bit-reversal on an axis without transforming it. The axes along which you perform a transform or a bit-reversal define the dataset; the axes along which you perform neither a transform nor bit-reversal are the instance axes, representing multiple independent data sets.

The CMSSL FFT routines perform multidimensional and multiple-instance FFTs by combining the following principles:

- If you supply a multidimensional data set, the routines perform a multidimensional FFT by computing a one-dimensional FFT along each axis of the cell in turn.

- If you specify one or more instance axes (possible only with the Detailed FFT, as described below), the routine computes multiple concurrent, independent one-dimensional FFTs along each cell axis in turn.

For example, if your array has rank 3 and extents ($p \times q \times r$), you can perform any of the FFTs indicated in Table 5.

Table 5. Possible FFTs with a three-dimensional input array.

Data Axis Numbers	Instance Axis Numbers	Resulting FFTs
1	2, 3	*qr* concurrent one-dimensional FFTs along the first axis.
2	1, 3	*pr* concurrent one-dimensional FFTs along the second axis.
3	2, 3	*pq* concurrent one-dimensional FFTs along the third axis.
1, 2	3	*r* concurrent two-dimensional FFTs. The dataset for each two-dimensional FFT is defined by the first two axes. The third axis defines the instances.
1, 3	2	*q* concurrent two-dimensional FFTs. The dataset for each two-dimensional FFT is defined by the first and third axes. The second axis defines the instances.
2, 3	1	*p* concurrent two-dimensional FFTs. The dataset for each two-dimensional FFT is defined by the second and third axes. The first axis defines the instances.
1, 2, 3	none	One three-dimensional FFT on the entire array.

9.2.4 Current Restrictions and Planned Extensions

The current FFT algorithm requires that the length of each axis subject to transformation or bit-reversal is equal to some power of 2. All other array axes can have any length.

Future plans include combining the local reordering of multiple axes, with the result that the local reordering time will be essentially independent of the number of axes reordered.

Need for Interface Blocks

In this Beta release of the complex-to-complex FFT, whenever the operand array (called *c* in the example below) has a layout directive, an interface block is necessary; otherwise, the probablity of getting the right answer is low. The example below illustrates the use of the interface block. (For definitions of arguments, see the man page following this section.) In this case, one of the axes has been declared **:serial**. The subdirectory **fft/cmf** of the CMSSL examples directory contains an example that also illustrates the use of an interface block.

```
cmf$layout c(:serial, :news)
      complex c(lda,lda)

      interface
      subroutine fft_detailed(c, fft_type, ops, in_bit_order,
     $     out_bit_order, scale, fftsetup, ier)
cmf$layout c(:news, :news)
      complex c(:, :)
      character*4 fft_type
      integer ops(2)
      integer scale(2)
      integer in_bit_order(2), out_bit_order(2)
      integer fftsetup, ier
      end interface

      call fft_detailed(c, 'CTOC', ops, in_bit_order,
     $        out_bit_order, scale, fftsetup, ier)
```

9.2.5 Summary: Optimizing CCFFT Address Ordering

For an axis that is being transformed in a CCFFT,

- If you specify *opposite* address orderings for input and output, the Detailed FFT performs the single bit-reversal that is inherent in the Cooley-Tukey algorithm, and produces output that is bit-reversed relative to the input.
- If you specify the *same* address ordering for input and output, the Detailed FFT performs an *extra* bit reversal, and produces output with the same ordering as the input. The extra bit reversal exacts a performance cost.

For an axis that is not being transformed,

- If you specify *opposite* address orderings for input and output, the Detailed FFT performs a single bit-reversal and produces output that is bit-reversed relative to the input.
- If you specify the *same* address ordering for input and output, the Detailed FFT performs *no* bit reversal.

Thus, for optimal CCFFT performance, specify

- Opposite address orderings for input and output, for each axis that is being transformed.
- The same address ordering for input and output, for each axis that is not being transformed.

Note that violating these recommendations along any axis is costly, but violating them along several axes is not significantly more costly.

To bit-reverse the address ordering of a dataset without transforming it, call the Detailed FFT specifying no transform for all axes and specifying *opposite* values for the input and output address ordering.

These points are independent of the directions of the transforms.

Complex-to-Complex Fast Fourier Transform

The routines described below use a complex-to-complex Fast Fourier Transform (CCFFT) algorithm to calculate the Discrete Fourier Transform of an *n*-dimensional CM array. The Simple FFT call, **fft**, performs the same operation (either a forward or an inverse FFT) along all axes of the array. With the Detailed FFT call, **fft_detailed**, you can specify different transform operations, bit orderings, and scaling factors for each axis of the array.

SYNTAX

setup_id = **fft_setup** (*A, type, ier*)

fft (*A, type, op, setup_id, ier*)

fft_detailed (*A, type, ops, in_bit_orders, out_bit_orders, scales, setup_id, ier*)

deallocate_fft_setup (*setup_id*)

ARGUMENTS

A	Complex CM array. The *A* you supply in the **fft_setup** call must have the same rank, axis extents, layout directives, and precision as the *A* arguments you supply in all subsequent **fft** or **fft_detailed** calls associated with the setup call. All axes that are to be transformed must have power-of-2 extents.
	When you call **fft_setup**, you may supply arbitrary values in *A*; the setup routine neither examines nor modifies the contents of the array, but rather uses its size, shape, and data type to create the setup object.
	When you call **fft** or **fft_detailed**, *A* is the array to be transformed. Upon completion, this array is overwritten with the results.
	The elements of *A* may be complex or double- precision complex. The floating-point precision of the result data always matches that of the input.
type	Front-end variable of type CHARACTER*4 that specifies the type of FFT to be performed. Specify **'CTOC'** for a complex-to-complex transform.

op	Front-end scalar integer variable indicating the direction in which the *A* axes are to be transformed in a Simple FFT. Must be one of the following symbolic constants (or the equivalent integer value): **CMSSL_f_xform** (1) forward transformation **CMSSL_i_xform** (2) inverse transformation
ops	Front-end integer vector with length equal to the rank of *A*. Each element specifies the direction (if any) in which the corresponding *A* axis is to be transformed in a Detailed FFT. Each element must be one of the following symbolic constants (or the equivalent integer value): **CMSSL_nop** (0) no operation **CMSSL_f_xform** (1) forward transformation **CMSSL_i_xform** (2) inverse transformation
in_bit_orders	Front-end integer vector with length equal to the rank of *A*. Each element indicates the initial address ordering of the corresponding *A* axis in a Detailed FFT. Each element must be one of the following symbolic constants (or the equivalent integer value): **CMSSL_normal** (0) normal address ordering **CMSSL_bit_reversed** (1) bit-reversed address ordering
out_bit_orders	Front-end integer vector with length equal to the rank of *A*. Each element indicates the desired output address ordering of the corresponding *A* axis in a Detailed FFT. Each element must be one of the following symbolic constants (or the equivalent integer value): **CMSSL_normal** (0) normal address ordering **CMSSL_bit_reversed** (1) bit-reversed address ordering
scales	Front-end integer vector with length equal to the rank of *A*. Each element specifies the scaling factor (if any) for the corresponding *A* axis. Each element must be one of the following symbolic constants (or the equivalent integer value): **CMSSL_noscale** (0) no scaling **CMSSL_scale_sqrt** (1) scale by square root of the axis length **CMSSL_scale_n** (2) scale by the axis length

At the end of the **fft_detailed** call, each element of A is divided by the product of the scaling factors. (The axes for which you specified **CMSSL_noscale** do not contribute to this product.)

setup_id	Scalar integer variable. The ID of a setup structure returned by the **fft_setup** routine.
ier	Scalar integer variable. Upon return, contains 0 if the call succeeded.

DESCRIPTION

To perform a CCFFT, you must follow these steps:

1. Call **fft_setup**.

   This routine computes internal values (including the twiddle factors), allocates an internal FFT setup object, and computes an FFT setup ID, which you must supply in all subsequent Simple FFT, Detailed FFT, and deallocation calls associated with this setup call.

2. Call **fft** or **fft_detailed**.

   These routines execute the Simple and Detailed FFTs, respectively. You may follow one call to **fft_setup** with multiple calls to these routines. However, the result of a call to **fft_setup** can only be used for arrays with the same rank, axis extents, layout directives, and precision as the array supplied in the setup routine. If these parameters change, you must call **fft_setup** again to create another setup structure.

3. Call **deallocate_fft_setup**.

   This routine removes an FFT setup object that is no longer needed. Because an FFT setup object occupies both partition manager and processing node memory, you should free this memory by deallocating the setup object after the completion of all FFT invocations that use it.

The time required to compute the contents of an FFT setup structure is substantially longer than the time required to actually perform an FFT. For this reason, and because it is common to perform FFTs on many arrays with the same rank, axis lengths, layout directives, and precision, the setup phase is separated from the transform phase.

You may have more than one setup ID active at a time; that is, you may call the setup routine more than once before deallocating any setup IDs. When you call **fft, fft_detailed,** or **deallocate_fft_setup**, you must be sure to provide the correct setup ID.

Multidimensional and Multiple-Instance FFTs. The **fft** routine computes a multidimensional transform by performing a one-dimensional transform along each axis in turn. The **fft_detailed** routine supports multiple instances, and computes one or more concurrent, independent one-dimensional FFTs along each data axis in turn.

Front-End Scalar Argument for Simple FFT. In the **fft** call, the *op* argument determines whether the routine performs a forward or an inverse FFT transform along each axis. The transform direction is the same for all axes. The simple FFT uses the following conventions:

- Along each axis, input and output data are stored using the same bit ordering.
- For a forward FFT, no scaling factors are used. For an inverse FFT, the axis length is used as the scaling factor for each axis.

Front-End Vector Arguments for Detailed FFT. The **fft_detailed** call requires four front-end vector arguments, *ops, in_bit_orders*, *out_bit_orders*, and *scales*, whose lengths must equal the rank of *A*. Within each of these vectors, each element specifies an option for the corresponding axis of *A*, as follows:

- The *ops* vector specifies the direction in which the corresponding *A* axis is to be transformed, or specifies that the axis is not to be transformed. In a multiple-instance FFT, the axes to be transformed are the data axes, and the axes that are not to be transformed are the instance axes.
- The *in_bit_orders* and *out_bit_orders* vectors specify the order in which the original and the resulting *A* values are stored along each axis. In normal address ordering (**CMSSL_normal**), data values are stored consecutively. Consecutive order is the only compiler-supported data allocation scheme. In bit-reversed ordering, consecutive data values are stored according to sequential addresses whose high- and low-order bits have been swapped; that is, the consecutive allocation scheme is applied to the bit-reversed indices.

  Although **fft_detailed** can handle any combination of bit orderings, performance is best when the axes of an untransformed array and a transformed array have opposite bit-orderings — that is, input normal and output bit-reversed, or input bit-reversed and output normal.

  For some applications, operating with bit-reversed indices is acceptable. However, special caution is required when data motion is involved. For example, NEWS communication is efficient for SHIFT operations, but if you use NEWS communication to perform a SHIFT on data that is bit-reversed, the results will not be those expected. Bit reversing rearranges the order of the data elements along an axis; a subsequent SHIFT simply shifts those rearranged elements. If you want to perform an operation that requires adjacency of consecutive indices, you must either operate on data that has normal bit ordering or take the bit-reversal into account in your computations.

- The *scales* vector specifies a scaling factor (none, square root of axis length, or axis length) for each axis. At the end of the **fft_detailed** call, each element of *A* is divided by the product of the scaling factors. (The axes for which you specified **CMSSL_noscale** do not contribute to this product.)

  For example, suppose *A* has dimensions (*m, n, p*) and you supply

  (CMSSL_scale_n, CMSSL_scale_sqrt, CMSSL_noscale)

  as the *scales* vector. At the end of the **fft_detailed** call, each element of *A* is divided by *m* * sqrt(*n*).

  One artifact of the Fourier Transform algorithm is that if no scaling is performed, then a forward FFT followed immediately by an inverse FFT results in the original array with all values multiplied by the FFT size — that is, the product of the lengths of the transformed axes. Scaling is typically used to correct this effect and prevent arithmetic overflow. Typically, a scaling factor of **CMSSL_noscale** is used for all axes in a forward transform, and a scaling factor of **CMSSL_scale_n** is used for all axes being transformed in an inverse transform. Alternatively, you may specify a scaling factor of **CMSSL_scale_sqrt** for all transformed axes in both the forward and the inverse FFTs. This alternative is less efficient, but can reduce the risk of arithmetic overflow for very large numbers.

  Scaling is independent of the operations performed along the axes. For example, it is possible to specify a scaling factor of **CMSSL_scale_n** for an axis that is not being transformed.

Bit-Reversing the Addressing of a CM Array. You can use the Detailed FFT to bit-reverse the addressing of any *n*-dimensional CM array. To do this, specify **CMSSL_nop** for all elements of the *ops* argument, and opposite values for the input and output orderings; that is, either **CMSSL_normal** for the *in_bit_orders* values and **CMSSL_bit_reversed** for the *out_bit_orders* values, or **CMSSL_bit_reversed** for the *in_bit_orders* values and **CMSSL_normal** for the *out_bit_orders* values.

NOTES

Setup Uses Array Shape, Layout, and Data Type. The setup ID computed by the **fft_setup** call can be used for all arrays with the same shape, layout, and data type (including precision) as the *A* argument supplied in the **fft_setup** call — and *only* for such arrays. If a transform is to be performed on two arrays, *A* and *B*, of the same shape, layout, and data type, then *one* call to the setup routine suffices, even if transforms are performed on different axes of the two arrays. But if *A* and *B* have different shapes, layouts, or data types, then you must make one call for each array. Note that to be the same shape, two arrays must not only have the same number of axes, but also the same axis lengths. Thus, if two arrays with the same number of axes differ in even one axis length or in any layout directives, they require separate setup calls.

Setup is Private. The FFT setup data contains information private to the FFT. The format of this data is not documented and may change between CMSSL releases. For this reason, application code should never access or modify the contents of an FFT setup structure.

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

Bit Reversal and Performance. For an axis that is being transformed,

- If you specify *opposite* address orderings for input and output, the Detailed FFT performs the single bit-reversal that is inherent in the Cooley-Tukey algorithm, and produces output that is bit-reversed relative to the input.
- If you specify the *same* address ordering for input and output, the Detailed FFT performs an *extra* bit reversal, and produces output with the same ordering as the input. The extra bit reversal exacts a performance cost.

For an axis that is not being transformed,

- If you specify *opposite* address orderings for input and output, the Detailed FFT performs a single bit-reversal and produces output that is bit-reversed relative to the input.
- If you specify the *same* address ordering for input and output, the Detailed FFT performs *no* bit reversal.

Thus, for optimal FFT performance, specify

- Opposite address orderings for input and output, for each axis that is being transformed.

- The same address ordering for input and output, for each axis that is not being transformed.

To bit-reverse the address ordering of a dataset without transforming it, call the Detailed FFT specifying no transform for all axes and specifying *opposite* values for the input and output address ordering.

These points are independent of the directions of the transforms.

EXAMPLES

Sample code that uses the complex-to-complex FFT can be found on-line in the subdirectory

`fft/cmf/`

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

Provided below are code skeletons showing the basic usage of the Simple FFT and the Detailed FFT.

Code Skeleton Showing Basic Usage of Simple FFT. The following code demonstrates the syntax of the **fft** routine:

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

      INTEGER setup_id, ier
      COMPLEX my_cm_array (256, 256)

C     initialize our cm array (this makes sure it's on
C       the cm)
      my_cm_array = (1.0, 0.0)

C     create an FFT setup id before taking the FFT
      setup_id = fft_setup (my_cm_array, 'CTOC', ier)

C     perform the forward transform on my_cm_array
      CALL fft(my_cm_array,'CTOC',CMSSL_f_xform,setup_id,ier)

C     use the transformed array here

C     perform the inverse transform on my_cm_array
      CALL fft(my_cm_array,'CTOC',CMSSL_i_xform,setup_id,ier)
```

```
C       finally, scrap the setup
        CALL deallocate_fft_setup( setup_id )

        STOP
        END
```

Code Skeleton Showing Basic Usage of Detailed FFT. The code below demonstrates the syntax of the **fft_detailed** routine. After allocating and initializing the needed variables, this code performs a forward FFT followed by an inverse FFT on the array **my_cm_array**. Notice that the output of the forward FFT call is in bit-reversed order. The transformed array is operated on with data in bit-reversed order, thereby avoiding two applications of the reordering function (one after the forward FFT, one after the inverse FFT).

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

C       declare and initialize variables
        INTEGER forward_ops(2), inverse_ops(2), normal_bits(2),
       & reversed_bits(2), unscaled(2), n_scaled(2), setup, ier
        DATA forward_ops/CMSSL_f_xform, CMSSL_f_xform/,
       &  inverse_ops/CMSSL_i_xform, CMSSL_i_xform/,
       &  normal_bits/CMSSL_normal, CMSSL_normal/,
       &  reversed_bits/CMSSL_bit_reversed,CMSSL_bit_reversed/,
       &  unscaled/CMSSL_noscale, CMSSL_noscale/,
       &  n_scaled/CMSSL_scale_n, CMSSL_scale_n/,
       &  ier/0/
        COMPLEX my_cm_array (256, 256)

C       initialize our cm array (this ensures it's on the cm)
        my_cm_array = (1.0, 0.0)

C       get the setup
        setup_id = fft_setup( my_cm_array. 'CTOC', ier )

C       call a forward detailed FFT
        CALL fft_detailed( my_cm_array, 'CTOC', forward_ops,
       & normal_bits, reversed_bits, unscaled, setup_id, ier )

C      Do something with the transformed array here

C       call an inverse detailed FFT
        CALL fft_detailed( my_cm_array, 'CTOC', inverse_ops,
       & reversed_bits, normal_bits, n_scaled, setup_id, ier )

C       scrap the setup
        CALL deallocate_fft_setup (setup_id)
```

```
      STOP
      END
```

9.3 References

For further information about the CCFFT, see the following references:

1. Buzbee, B. L., G. H. Golub, and C. W. Nielson. On Direct Methods for Solving Poisson's Equations. *SIAM J. Numer. Anal.* 7, no. 4 (1970): 627-56.

2. Chu, C. *The Fast Fourier Transform on Hypercube Parallel Computers.* Ph.D. Thesis, Center for Applied Mathematics, Cornell University, Ithaca NY 14585.

3. Cooley, J. C., P. Lewis, and P. D. Welch. The Fast Fourier Transform Algorithm: Programming Considerations in the Calculation of the Sine, Cosine, and Laplace Transforms. *J. Sound Vibrations* 12, no. 3 (1970): 315-37.

4. Cooley, J. C., and J. W. Tukey. An Algorithm for the Machine Computation of Complex Fourier Series. *Math. Comp.* 19 (1965): 291-301.

5. Edelman, A. *Optimal Matrix Transposition and Bit-Reversal on Hypercubes: Node Address–Memory Address Exchanges.* Thinking Machines Corporation Technical Report, 1989.

6. Hockney, R. W. The Potential Calculation and Some Applications. *Methods Comput. Phys.* 9 (1970): 135-211.

7. Hong, J. W., and H. T. Kung. I/O Complexity: The Red-Blue Pebble Game. In *Proc. of the 13th ACM Symposium on the Theory of Computation.* ACM, 1981. Pp. 326-33.

8. Johnsson, S. L. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *J. Parallel Distributed Comput.* 4, no. 2 (1987): 133-72.

9. Johnsson, S. L. and C-T. Ho. Matrix Transposition on Boolean n-Cube Configured Ensemble Architectures. *SIAM J. Matrix Anal. Appl.* 9, no. 3 (1988): 419-54.

10. Johnsson, S. L. and C-T. Ho. *Optimal Communication Channel Utilitzation for Matrix Transposition and Related Permutations on Boolean Cubes.* Technical Report TR-03-91, Harvard University, Division of Applied Sciences, January 1991.

11. Johnsson, S. L., C-T. Ho, M. Jacquemin, and A. Ruttenberg. Computing Fast Fourier Transforms on Boolean Cubes and Related Networks. In *Advanced Algorithms and Architectures for Signal Processing II,* **826**. Society of Photo-Optical Instrumentation Engineers, 1987. Pp. 223-31.

12. Johnsson, S. L., Jacquemin, M., and C-T. Ho. *High Radix FFT on Boolean Cube Networks.* Harvard University, Division of Applied Sciences, Technical Report TR-25-91. To appear in the *Journal of Computational Physics.*

13. Johnsson, S. L. and R. L. Krawitz. Cooley-Tukey FFT on the Connection Machine. *Parallel Computing* **18**, no. 11 (1992): 1201-21.

14. Johnsson, S. L., M. Jacquemin, and R. L. Krawitz. Communication Efficient Multi-Processor FFT. *J. Comp. Phys* **102**, no. 2 (1992): 381-97.

15. Johnsson, S. L and C.-T. Ho. Boolean Cube Emulation of Butterfly Networks Encoded by Gray Code. Yale University Department of Computer Science, Technical Report YALEU/DCS/RR-764, 1990. Also Thinking Machines Corporation Technical Report BA90-1, TMC-5. To appear in *Journal of Parallel and Distributed Computing.*

16. Swarztrauber, P. N. The Methods of Cyclic Reduction, Fourier Analysis, and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle. *SIAM Review* **19** (1977): 490-501.

17. Swarztrauber, P. N. Multiprocessor FFTs. *Parallel Computing* **5** (1987): 197-210.

18. Temperton, C. On the FACR(1) Algorithm for the Discrete Poisson Equation. *J. of Computational Physics* **34** (1980): 314-29.

19. Van Loan, C. *Computational Frameworks for the Fast Fourier Transform.* SIAM, 1992.

Chapter 10

Ordinary Differential Equations

This chapter describes the CMSSL routine that solves the initial value problem for a system of first-order ordinary differential equations (ODEs) using a Runge-Kutta method. Section 10.2 provides references.

10.1 Explicit Integration of Ordinary Differential Equations Using a Runge-Kutta Method

The initial value problem for a system of N coupled first-order ODEs,

$$dy_i(x)/dx = f_i(x, y_1, \ldots, y_N) \quad i=1, \ldots, N \qquad (1)$$

consists of finding the values $y_i(x_1)$ at some value x_1 of the independent variable x, given the values $y_i(x_0)$ of the dependent variables at x_0. The **ode_rkf** routine solves the initial value problem by integrating explicitly the set of equations (1) using a fifth-order Runge-Kutta-Fehlberg formula. Control of the step size during integration is automatic. The evaluation of the right-hand side and possibly the scaling array for accuracy control are provided by the user through a reverse communication interface.

For detailed information about the **ode_rkf** routine, refer to the man page at the end of this section. The examples below use the syntax and reverse communication interface described in the man page.

10.1.1 Examples

Provided below are schematic examples of how to use the **ode_rkf** reverse communication interface. For actual codes, refer to the on-line examples (the pathname is included in the man page). The following examples assume the variables are stored in a two-dimensional array.

Example 1

In this example, we want to integrate an autonomous system of coupled ODEs using the default error control. The routine `deriv(y,dy)` evaluates the derivatives of *y*. Since the system is autonomous, the derivatives are independent of *x*. We check for possible zero values of *yscal* and set them to a small value, *epsilon*. The reverse communication proceeds as follows:

```
      info=0
      ido=0

 10   continue
      call ode_rkf(ido,y,w,x,xc,xf,dx,rtol,atol,ipntr,info,setup)

      if(ido.eq.1) then

          call deriv(w(ipntr(1),:,:),w(ipntr(2),:,:))

      else if (ido.eq.3)

          w(ipntr(2),:,:)=max(w(ipntr(2),:,:),epsilon)

      else
          stop
      endif
      goto 10
```

Example 2

Suppose we want to integrate a time-dependent system of ODEs and we choose our own error control. The routine `deriv(xc,y,dy)` evaluates the derivatives at *xc* while the routine `scal(dx,y,dy,yscal)` evaluates $yscal = tol(|y| + dx * |dy/dx|)$. We do not check for zero values in *yscal* (*ido* = 3), since we assume this is done in the routine `scal`.

```
      info=1
      ido=0
```

```
10  continue

    call ode_rkf(ido,y,w,x,xc,xf,dx,rtol,atol,ipntr,info,setup)

    if(ido.eq.1) then

        call deriv(xc,w(ipntr(1),:,:),w(ipntr(2),:,:))

    else if (ido.eq.2) then

        call scal(dx,y,w(ipntr(1,:,:)),w(ipntr(2),:,:))

    else
        stop
    endif

    goto 10
```

Explicit Integration of Ordinary Differential Equations Using a Runge-Kutta Method

The routines described below integrate ordinary differential equations (ODEs) explicitly using a fifth-order Runge-Kutta-Fehlberg formula. Control of the step size during integration is automatic.

SYNTAX

ode_rkf_setup (*y, w, setup, ier*)

ode_rkf (*ido, y, w, x, xc, xf, dx, rtol, atol, ipntr, info, setup*)

deallocate_ode_rkf_setup (*setup*)

ARGUMENTS

ido Scalar integer variable. Reverse communication flag. *ido* must be zero on the first call to **ode_rkf**. The **ode_rkf** routine sets *ido* to indicate the type of operation to be performed by the calling program, places the operand in *w*(*ipntr*(1), :, ..., :), and returns control to the calling program. The calling program has the responsibility of carrying out the requested operation and calling **ode_rkf** again, placing the result in *w*(*ipntr*(2), :, ..., :). The values of *ido* have the meanings listed below. All values except 0 are returned to the calling program.

0	The calling program must supply this value on the first call to **ode_rkf**.
1	The calling program must compute the derivatives of the dependent variables $dy/dx=f(x,y)$ at $x=xc$. Array elements *ipntr*(1) and *ipntr*(2) are pointers into *w* for *y* and *dy/dx*, respectively, at *xc*.
2	(Returned only if *info* =1.) The calling program must compute the scaling array for accuracy control. The values of the dependent variables *y*(*x*) are in array *y*, while the pointer into *w* for *dy*/*dx*

at *x* is *ipntr*(1). The pointer into *w* for the scaling array is *ipntr*(2).

3	At least one component of the scaling array stored in *w*(*ipntr*(2), :, ..., :), is zero. Reset the zero values to avoid overflow in the stepsize control (equation (2) in the Description section).
99	The integration is completed.

y	Real CM array of arbitrary dimension and shape. Represents the collection of dependent variables. The values of *y* are always the values of the variables at *x*. Input values are taken as initial values for the integration.
w	Real CM work array of rank one greater than that of *y*. The first axis must have extent at least 6 and must be declared **:serial**. The remaining axes must match the axes of *y* in order of declaration, extents, and layout. This array is used in the basic iteration for reverse communication. Do not modify the values of *w* except as indicated by the returned value of *ido*.
x	Real scalar variable. Independent variable. The value of *x* on input (*ido*=0) corresponds to the initial values of the dependent variables *y*. During the integration, *x* always has the last value of the independent variable at which the variables *y* have been computed successfully.
xc	Real scalar variable. Intermediate value of the independent variable to be used for the evaluation of the derivatives. When the derivatives depend explicitly on *x*, it is the variable *xc*, and not *x*, that must be passed to the user-supplied subroutine that evaluates the derivatives when *ido*=1.
xf	Real scalar variable. Value of the independent variable *x* at which the integration is terminated. Input only.
dx	Real scalar variable. Step size. On input (*ido*=0), *dx* is the estimated first increment of the independent variable to be attempted by the integrator. Since the integrator is adaptive, a step size that is too large (small) will be reduced (increased) automatically. However, the input value of *dx* cannot be 0. The sign of the input value of *dx* is set internally to the sign of *xf*-*x*.

During the integration, *dx* is the value of the step size currently being attempted by the integrator.

rtol	Relative tolerance for the local error control. Must have the same precision (single or double) as *y*. Ignored if *info* = 1. See the Description section for details.
atol	Absolute tolerance for the local error control. Must have the same precision (single or double) as *y*. Ignored if *info* = 1. See the Description section for details.
ipntr	One-dimensional front-end integer array of length at least 2. On return, contains pointers to mark the locations in the work array *w*:

ipntr(1) When *ido* = 1, *ipntr*(1) points to the section of *w* holding the values of the dependent variables at *xc*.

When *ido* = 2, *ipntr*(1) points to the array holding the values of the derivatives at *x*.

ipntr(2) When *ido* = 1, *ipntr*(2) points to the section of *w* that should hold the values of the derivatives at *xc* after a user-supplied routine is called using the reverse communication interface.

When *ido* = 2 and *info* = 1, *ipntr*(2) points to the section of *w* that should hold the values of the scaling array used for accuracy control after a user-supplied routine is called using the reverse communication interface.

info	Scalar integer variable. On input, set *info* to 1 if you want to provide your own scaling array for accuracy control using the reverse communication interface. Any other integer value causes **ode_rfk** to use the default scaling array (see the Description section).
setup	One-dimensional front-end integer array of length 2.
ier	Scalar integer variable. Set to 0 upon successful return. Upon return from **ode_rkf_setup**, may contain the following error codes:

−1	The first dimension of *w* is not declared **:serial.**

-2	The serial dimension of *w* has extent less than 6.
-3	The rank of *w* is not one greater than the rank of *y*.
-4	The sections of *w* containing the vectors and indexed by the first dimension do not have the same shape as *y*.

DESCRIPTION

The initial value problem for a system of N coupled first-order ordinary differential equations (ODEs),

$$dy_i(x)/dx = f_i(x, y_1, \ldots, y_N) \quad i=1, \ldots, N \qquad (1)$$

consists of finding the values $y_i(x_1)$ at some value x_1 of the independent variable x, given the values $y_i(x_0)$ of the dependent variables at x_0. The **ode_rkf** routine solves the initial value problem by integrating explicitly the set of equations (1) using a fifth-order Runge-Kutta-Fehlberg formula. Control of the step size during integration is automatic. The evaluation of the right-hand side and possibly the scaling array for accuracy control are provided by the user through a reverse communication interface.

Setup and Deallocation. To use **ode_rkf**, follow these steps:

1. Call **ode_rkf_setup.**

   This routine generates a setup ID and returns it in the front-end array *setup*. You must supply this *setup* array in all subsequent **ode_rkf** and **deallocate_ode_rkf** calls associated with this setup call.

2. Call **ode_rkf** iteratively, as described under **Reverse Communication Interface**, below.

   You can use the same *setup* array to solve more than one initial value problem sequentially, as long as the array geometries are the same. You can also have more than one setup active at a time.

3. Call **deallocate_ode_rkf.**

   This routine deallocates the memory associated with the setup ID.

Step Size Control and Accuracy. For a trial integration step dx, **ode_rkf** estimates the errors $yerr_i(x + dx)$ of the computed solution at $x + dx$. This allows for automatic control of the step size if one is able to accept or reject the step based on a prescribed accuracy requirement. The test implemented in **ode_rkf** is to accept the step if *errmax* < 1, where

$$errmax = \max_i \frac{|yerr_i(x + dx)|}{yscal_i(x)} \qquad i = 1, \ldots, N \quad (2)$$

and the scaling array *yscal* is defined by

$$yscal_i(x) = rtol * |y_i(x)| + atol \qquad i = 1, \ldots, N$$

The input arguments *rtol* and *atol* are the relative and absolute precision, respectively. This combination covers the range between relative (*atol* = 0) and absolute (*rtol* = 0) accuracy. If any component of *yscal* is zero, the reverse communication interface returns the code *ido* = 3, allowing you to reset the zero components in order to avoid overflow in the evaluation of (2).

You may wish to monitor the accuracy of the integration in a way that is more appropriate to your problem. For instance, it may be desirable to scale the error with the time step *dx* and to include the value of the derivatives in *yscal*, as in Example 2 in Section 10.1.1. This combination enforces accuracy over the whole integration range, whereas the default formula merely ensures local accuracy. It is possible to set your own array *yscal* by seting *info* = 1 and using the reverse communication interface (*ido* = 2).

Reverse Communication Interface. The aim of the reverse communication interface is to isolate from the **ode_rkf** code the evaluation of the derivatives and possibly the construction of the scaling array. Such operations are performed by user-supplied routines, on data structures that are the most natural to the problem at hand. To this end, **ode_rkf** is called iteratively; **ode_rkf** gives control back to the calling routine whenever the evaluation of the derivatives or the scaling array is required. The reverse communication interface also allows you to reset possible zero values of *yscal* in order to avoid overflow in (2).

The reverse communication flag, *ido*, which must be 0 for the first call to **ode_rkf**, dictates which operation is to be performed. When *ido* = 1, the derivatives must be evaluated at the intermediate value *xc*; that is, you must evaluate the right-hand side of (1)

$$f_i(xc, y_1, \ldots, y_N) \qquad i = 1, \ldots, N$$

at *xc*. The source array, which contains the values of the dependent variables at *xc*, is stored in the section of the workspace array *w* indexed by the pointer *ipntr*(1). The user-supplied routine that evaluates the derivatives at *xc* must place the derivatives into the destination array, which is the section of the workspace array *w* indexed by the pointer *ipntr*(2).

When *ido* = 2, you must set up your own scaling array to be used for accuracy control. You are prompted to do so only if the input parameter *info* has been set to 1. When **ode_rkf** returns *ido* = 2, the values of the dependent variables and their derivatives to be used in the construction of the scaling array are in the arrays *y* and *w(ipntr*(1),:...:), respectively. The time step currently attempted by the integrator is *dx*. The user-supplied routine that creates the scaling array must place the scaling array into the section of the workspace array *w* indexed by the pointer *ipntr*(2).

A fifth-order Runge-Kutta step requires six right-hand-side evaluations at different values of the independent variable *xc*. Note that it is the array *w(ipntr*(1), :, ..., :), and not *y*, that contains the values of the dependent variables at *xc*, to be used in the evaluation of the derivatives. The array *y* contains the values of the dependent variables at *x*, and is only updated once the currently attempted step *dx* is successful.

Finally, *ido* = 3 indicates that at least one component of the scaling array is zero. To avoid overflow in the error computation (2), you need only replace the zero components with small values, as in Example 1 in Section 10.1.1.

NOTES

Use of Array *w*. Do not use the CM array *w* as temporary workspace.

Data Layout. The **ode_rkf** routine imposes several constraints on the way the CM arrays *y* and *w* are laid out on the machine. The array *y* contains the dependent variables. The number of dimensions of *y* and its layout on the machine can (and should) be chosen in such a way as to optimize the evaluation of the right-hand side of (1). In particular, *y* can be a multidimensional array. The product of the dimensions must be equal to the size of the problem (the number of dependent variables). Once the layout of *y* is chosen, *w* must adhere to the same layout directives as *y* except for its first extra dimension, which must be local to the processors and of size at least 6. Thus, *w* contains at least 6 arrays identical to *y*, "stacked up" in memory. The axis indexing these arrays in *w* must be the first axis; to ensure that it is local, the calling program must use a layout directive to declare it **:serial**.

For example, in the one-dimensional case, the array declarations should be as follows:

```
      real y(n), w(6,n)
CMF$LAYOUT   w(:serial,),y()
```

In the two-dimensional case, the declarations would be

```
          real y(n1,n2), w(6,n1,n2)
      CMF$LAYOUT    w(:serial,,),y(,)
```

In this second case, there are *n1* * *n2* dependent variables.

Caution Regarding Array Sections. In the reverse communication interface, the array sections *w*(*ipntr*(1),:,...,:) and *w*(*ipntr*(2),:,...,:) designate source and destination vectors, respectively, and are passed to the matrix vector subroutines as shown in the examples in Section 10.1.1. Since the first dimension of *w* is always serial, the sections *w*(*ipntr*(1),:,...,:) and *w*(*ipntr*(2),:,...,:) are passed in place only if all dimensions of *w* other than the first have a canonical layout. If *w* has a non-canonical dimension other than the first, the array sections are not passed in place; this degrades performance and may produce incorrect results. One remedy is to pass the whole array *w* along with the pointer array *ipntr* to the subroutine and extract the relevant array sections in the subroutine itself. Alternatively, you can make the interface explicit by means of an interface block. For information about passing array sections and about interface blocks, refer to the *CM Fortran Programming Guide*. For an example, refer to the on-line sample code.

EXAMPLES

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

```
ode/cmf
```

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

10.2 References

1. Press, W. H. and S. A. Teukolsky. Adaptive stepsize Runge-Kutta integration. *Computers in Physics* **6** (1992): 188–91.

2. Cash, J. R. and A. H. Karp. A variable order Runge-Kutta method for initial value problem with rapidly varying right hand sides. *ACM Trans. Math. Software* **16** (1990): 201–22.

Chapter 11

Linear Programming

This chapter describes the dense simplex routine, **gen_simplex**. Section 11.2 provides references.

11.1 Dense Simplex Routine

The **gen_simplex** routine solves multidimensional minimization problems using the simplex linear programming method. The goal is to find the minimum of a linear function of multiple independent variables. In the standard formulation, the problem is to minimize the inner product $c^T x$ subject to the conditions $Mx = b$, $0 \leq x \leq u$, where M is an $m \times n$ matrix, c is a coefficient vector, and $c^T x$ is referred to as the *cost*. The upper bound vector u may be infinity in one or more components.

11.1.1 Geometrical Description of the Algorithm

Geometrically, the algorithm can be described in terms of the polytope in n-space defined by the plane $Mx = b$ and the bounds $0 \leq x \leq u$. The problem is to find a vertex of the polytope that lies as far as possible in the direction $-c$. If the solution consists of more than one point, **gen_simplex** chooses one vertex within the solution as its result.

The simplex method involves the following steps:

1. Find any vertex of the polytope.
2. Find a neighboring vertex that results in a lower cost.

3. Repeat Step 2 until no neighboring vertex results in a lower cost.

11.1.2 Vertices and Bases

To find a vertex, **gen_simplex** selects $n-m$ variables and fixes them to their upper bounds or to 0. These variables are selected so that the columns of M corresponding to the remaining m variables are linearly independent. Solving the resulting m-dimensional system yields a vertex. The *basis* defined by the m linearly independent columns is said to be *feasible* if the solution satisfies the bounds $0 \leq x \leq u$.

The process of moving from one basis to another is called a *pivot* or *iteration*. The input argument *max_iter* is the upper bound on the number of iterations **gen_simplex** will perform without finding a solution.

If you have any starting information about feasible bases (for example, you know or guess that certain columns belong in a feasible or optimal basis), you can supply this information to **gen_simplex** in the ***bcol*** argument, which is described in the man page following this section. Upon return, ***bcol*** contains information about the last basis **gen_simplex** found.

11.1.3 Input Array Format

When you call **gen_simplex**, you must supply a two-dimensional zero-based CM array, A, whose upper-left-hand corner of dimensions $(m+1) \times (n+1)$ contains the standard input items for simplex problems (see Figure 33):

- an offset, ω, from which $c^T x$ is subtracted (element (0,0)). Typically, $\omega=0$.
- the cost coefficient vector, c (row 0)
- the right-hand-side vector, b (column 0)
- the matrix M (rows 1 through m, columns 1 through n)

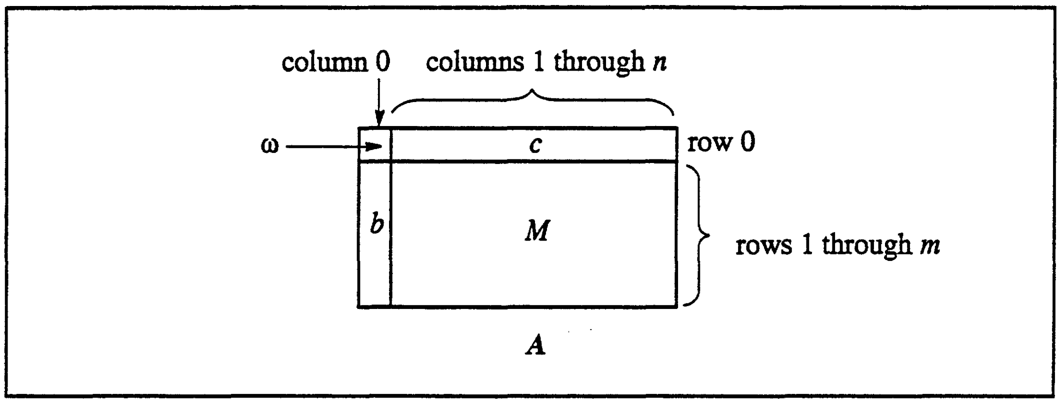

Figure 33. Format of A.

When **gen_simplex** transforms M to find a vertex (basis), row 0 also changes and becomes the *reduced cost row.* This row carries information about how close the current basis is to being a solution to the problem. Theoretically, when every element of the reduced cost row is greater than 0, the basis is optimal (the vertex is the optimal solution). In practice, when every element of the reduced cost row is greater than *-epsilon* (where *epsilon* is an input tolerance argument), the basis is considered to be optimal.

The working version of A during **gen_simplex** processing is referred to as the *tableau.*

NOTE

The **gen_simplex** routine may place zeros in rows of A beyond row m and columns beyond column n.

11.1.4 Reinversion

In some cases, the status code returned by **gen_simplex** indicates that you should *reinvert*: that is, call the routine again, restoring the original values of A (including row and column 0) and supplying the ***bcol*** values returned in the last call. Reinverting allows **gen_simplex** to clear any numerical errors that have accumulated and start again from a more numerically accurate version of the last basis obtained in the prior call. The input argument *reinvert_freq* is an upper bound on the number of iterations **gen_simplex** will perform before exiting with a status that requests reinversion.

11.1.5 Degeneracy

A problem is *degenerate* if more than one basis represents the same vertex. If **gen_simplex** encounters degeneracy, it may perform multiple iterations without much improvement in cost. The input arguments *degen_iter* and *degen_tol* determine how long **gen_simplex** will continue in a degenerate case: *degen_iter* is an upper bound on the number of iterations **gen_simplex** will perform with cost improvement of less than *degen_tol*. We use a slight modification of the EXPAND anti-cycling procedure (see reference 2 in Section 11.2).

11.1.6 Implementation

The **gen_simplex** routine takes the tableau *A* and creates or equivalences it with a four-dimensional tableau in the CM, distributed in such a way as to minimize communications overhead. Using special temporary arrays for pivot selection and the rank 1 update, **gen_simplex** proceeds either to solve the linear problem until the reduced cost row contains elements greater than *-epsilon* and the solution is feasible, or to exit with an appropriate *status* code.

11.1.7 Example

The following example assumes a knowledge of linear programming. Consider the linear programming problem

$$\begin{array}{lllll}
\text{Minimize} & -3x_1 & -5x_2 & & \\
\\
\text{subject to} & x_1 & +x_2 & = & 11 \\
 & x_1 & -x_2 & \geq & -4 \\
 & -2x_1 & +x_2 & \leq & 2 \\
 & x_1 & +x_2 & \geq & 2
\end{array}$$

with $x_1 \geq 1, x_2 \geq 0$. The problem may be transformed to

$$\begin{array}{llllllll} \text{Minimize} & -3x_1 & -5x_2 & & & & & -3 \\ \text{subject to} & x_1 & +x_2 & & & & = & 10 \\ & x_1 & -x_2 & -x_3 & & & = & -5 \\ & -2x_1 & +x_2 & & +x_4 & & = & 4 \\ & x_1 & +x_2 & & & -x_5 & = & 1 \\ & x_1, & x_2, & x_3, & x_4, & x_5 & \geq & 0 \end{array}$$

by adding the slack variable x_4 and the surplus variables x_3 and x_5, and transforming the variable x_1 to be bounded below by zero. To compensate for the change in the definition of x_1, we now include a cost offset (-3). We supply **gen_ simplex** with partial knowledge about a possible starting basis by setting ***bcol***(1)=0 (since we have no knowledge about a basic variable in row 1) and setting ***bcol***(2)=3, ***bcol***(3)=4, and ***bcol***(4)=5 to incorporate knowledge about slack and surplus variables. Since the variables have no upper bounds, we set ***bounds***(*i*) = infinity for all *i*. (To do this, use the built-in infinity function **d_infinity( )**; see the on-line example.) We set the input tableau argument *A* as follows:

$$A = \begin{array}{ccccccccc} 3 & -3 & -5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 10 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -5 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 4 & -2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

In this example, *A* contains some unused rows and columns; its active dimensions are m+1=5 and n+1=6. Elements of ***bcol*** corresponding to the unused rows are set to 0; elements of ***bounds*** corresponding to the unused columns are set to infinity. The array ***flip*** is set to **.false.**.

Dense Simplex

The **gen_simplex** routine solves multidimensional minimization problems using the simplex linear programming method. In the standard formulation, the problem is to minimize the inner product $c^T x$ subject to the conditions $Mx = b$, $0 \leq x \leq u$, where M is an $m \times n$ matrix, c is a coefficient vector, and $c^T x$ is referred to as the *cost*. The upper bound vector u may be infinity in one or more components.

SYNTAX

gen_simplex (*A, x, **flip, bounds, bcol**, statarray, constraint_axis, variable_axis, x_flip_bounds_axis, bcol_axis, m, n, iter_count, max_iter, reinvert_freq, epsilon, degen_iter, degen_tol, ier*)

ARGUMENTS

A Zero-based real CM array (single- or double-precision) with rank 2 and NEWS-ordered axes. Must have dimensions at least $(m+1) \times (n+1)$. On input, contains

- an offset ω from which $c^T x$ is subtracted (element (0,0)).
- the cost coefficient vector, c (row 0).
- the right-hand-side vector, b (column 0).
- the matrix M (rows 1 through m, columns 1 through n).

On exit, A contains the optimal tableau; that is, the basic columns of M are the optimal basis. Element $A(0,0)$ contains the optimal value $\omega - c^T x$, where ω is the original offset passed in and $c^T x$ is the optimal cost.

x One-based real CM array with rank 1, the same precision as A, and length n. Must be NEWS-ordered. When **gen_simplex** exits with *status* = 0 (or *status* = −1, if *reinv_freq* = 0), x contains the solution to the linear problem.

flip One-based logical CM array with rank 1 and length n. Must be NEWS-ordered.The first time you call **gen_simplex,** set

flip(i) = **.false.** for all i. On return, element ***flip***(i) is set to **.true.** if the ith variable has been set to its upper bound. When you reinvert, supply the ***flip*** values returned by the last call.

bounds
One-based real CM array with rank 1, the same precision as A, and length n. Must be NEWS-ordered. The element ***bounds***(i) contains the upper bound of the ith variable of the problem. Each bound must be ≥ 0, but may be infinite.

bcol
One-based integer CM array with rank 1 and length m. Must be NEWS-ordered.

On input to the first call to **gen_simplex**, if you have no information about a starting basis, supply ***bcol***(i) = 0 for all i. Alternatively, you may supply information about a possible starting basis, as follows:

- Set ***bcol***(i) to k to suggest that column k be the basic column for row i. The routine attempts to pivot on element (i, k).
- Set ***bcol***(i) to 0 if you have no information about the basic column for row i.
- Set ***bcol***(i) to a negative number if row i is redundant.

The information you supply is used as a suggestion; if it does not work, **gen_simplex** proceeds to find a full-rank feasible basis.

On exit, ***bcol***(i) contains the number of the basic column corresponding to row i of the problem. A negative entry indicates that row i appears to be numerically redundant.

When you reinvert (restore the values of A and call **gen_simplex** again), do not change the values of ***bcol***; supply the values returned by the last call. The routine uses these values to determine the new starting basis.

statarray
Ignored in the current release. Supply the scalar integer constant 0.

constraint_axis
Reserved for future releases. Supply the scalar integer constant 1.

variable_axis	Reserved for future releases. Supply the scalar integer constant 2.
x_flip_bounds_axis	Reserved for future releases. Supply the scalar integer constant 1.
bcol_axis	Reserved for future releases. Supply the scalar integer constant 1.
m	Scalar integer variable. The number of rows in the matrix M. (The array A must have at least $m + 1$ rows, since it also includes the cost row as row 0.)
n	Scalar integer variable. The number of columns in the matrix M. (The array A must have at least $n + 1$ columns, since it also includes the right-hand-side vector, b, as column 0.)
iter_count	Scalar integer variable. On return, contains the total number of iterations performed. The **gen_simplex** routine does not reset this counter; if you want to reset it, you must do so explicitly. Set *iter_count* to 0 before the first call to **gen_simplex**.
max_iter	Scalar integer variable. An upper bound on the number of iterations **gen_simplex** can perform without reaching an optimal solution. Suggested value: 40n.
reinvert_freq	Scalar integer variable. If you supply a value $p > 0$, **gen_simplex** will perform at most p iterations before exiting with a status that requests reinversion. (Suggested value: 10000.) If you set *reinvert_freq* to 0, **gen_simplex** proceeds until one of the following conditions occurs: **gen_simplex** finds a solution; exceeds the value of *max_iter*; determines that the problem is infeasible, unbounded, or degenerate; or encounters another fatal error. Numerical errors do not cause the routine to exit. However, if numerical errors have accumulated when **gen_simplex** has finished, it returns a status of -1. In this case, if you reinvert, **gen_simplex** clears the numerical errors and re-checks the solution.
epsilon	Double-precision real scalar variable. A tolerance for the tableau elements, introduced for numerical stability. A reduced cost or pivot element whose absolute value is less than *epsilon* is ignored in the pivot selection. These elements

are not necessarily zeroed, however, and may enter the calculations at a future time. When the argument *A* is double-precision, suggested values are between 10^{-10} and 10^{-6}, typically 10^{-8}.

degen_iter	Scalar integer variable. A bound on the number of iterations **gen_simplex** will perform with cost improvement less than *degen_tol*. Suggested value: 2000.
degen_tol	Double-precision real scalar variable. A lower bound on the cost improvement after *degen_iter* iterations. Suggested value: 1e-6.
ier	Scalar integer variable. On return, contains one of the following status codes:

1 (Optimal)	Successful termination. An optimal solution has been found.
2 (Reinvert)	Numerical errors have accumulated, or (if *reinvert_freq* is > 0) the number of iterations has exceeded the value of *reinvert_freq*. Restore the original values of *A* (including row and column 0) and call **gen_simplex** again without modifying *bcol* or *flip*. (This process is known as *reinversion.*)
4 (Infeasible)	The problem appears to be infeasible. However, the apparent infeasibility may be due to numerical ill-conditioning, in which case increasing the value of *epsilon* and calling **gen_simplex** again may clear the error.
8 (Unbounded)	Either the cost minimizing direction extends to infinity, or bad scaling or numerical errors cause the problem to appear unbounded. If the problem is known to be bounded, restore the original values of *A* (including row

	and column 0), increase the value of *epsilon*, and call **gen_simplex** again.
16 (Degenerate)	The number of iterations has exceeded the value of *degen_iter* and the cost has improved by less then *degen_tol*. Restore the original values of *A* (including row and column 0), increase *reinvert_freq* and *epsilon*, and call **gen_simplex** again.
32 (Suboptimal)	An optimal solution has not yet been found, but the number of iterations has exceeded *max_iter*.
-1 (Bad data type)	*A* has a data type other than real (single- or double-precision), or *A*, *x*, and ***bounds*** do not match in data type and precision.
-2 (Axis specification error)	You supplied an invalid value for one of the axis arguments.
-128 (Internal/Other)	An internal error has occurred.

DESCRIPTION

The simplex method involves the following steps:

1 Find any vertex of the polytope defined by the plane $Mx = b$ and the bounds $0 \leq x \leq u$.

2 Find a neighboring vertex that results in a lower cost.

3 Repeat Step 2 until no neighboring vertex results in a lower cost.

To find a vertex, **gen_simplex** selects $n-m$ variables and fixes them to their upper bounds or to 0. These variables are selected so that the columns of M corresponding to the remaining m variables are linearly independent. Solving the resulting m-dimensional system yields a vertex. The *basis* defined by the m linearly independent columns is said to be *feasible* if the solution satisfies the bounds $0 \leq x \leq u$. The process of moving from one basis to another is called a *pivot* or *iteration.*

When **gen_simplex** transforms *M* to find a vertex (basis), row 0 of *A* also changes and becomes the *reduced cost row.* This row carries information about how close the current basis is to being a solution to the problem. Theoretically, when every element of the reduced cost row is greater than 0, the basis is optimal (the vertex is the optimal solution). In practice, when every element of the reduced cost row is greater than -*epsilon*, the basis is considered to be optimal. The solution vertex is returned in *x*, and the basic columns are returned in *A*.

NOTES

Inactive Elements May be Zeroed. The **gen_simplex** routine may place zeros in rows of *A* beyond row *m* and columns beyond column *n*.

Performance. For best performance, *A* must satisfy the following conditions:

- The product of the *physical axis extents* must equal the total number of processors in the machine (or CM-5 partition). In this context, the "number of processors" is the number returned by **CMF_number_of_processors( )**; that is, the number of Vector Units (on a CM-5 with Vector Units), parallel processing nodes (on a CM-5 without Vector Units), or processing elements (on a CM-200 in the slicewise execution model). The physical extent of an axis is the number of processors along the axis.
- The subgrid size *in each dimension* must be a multiple of the vector length. In CM Fortran releases prior to Version 2.1, the vector length is
  - 4 for a CM-2 or CM-200 (slicewise execution model)
  - 1 for a CM-5 without Vector Units
  - 8 for a CM-5 with Vector Units

  Beginning with CM Fortran Version 2.1, the vector length will be 1 for all machines, and this requirement will disappear.

If *A* does not satisfy these conditions, **gen_simplex** must make a copy of *A*, with a resulting cost in both time and memory.

If *A* does satisfy the above conditions (so that a copy is not required), then for best performance, make the subgrid size approximately the same in each dimension. (If **gen_simplex** makes a copy of *A*, this guideline does not apply.)

Precision. For best results, double precision is recommended.

EXAMPLES

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

```
simplex/cmf
```

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

11.2 References

For more information about the simplex method, see the following references:

1. Luenberger, D. G. *Linear and Nonlinear Programming.* Reading, MA: Addison-Wesley, 1984 (or any other introductory text on linear programming).

2. Gill, P. E., W. Murray, M. A. Saunders, and M. H. Wright. A practical anti-cycling procedure for linearly constrained problems. *Mathematical Programming* **45** (1989): 437–74.

3. Eckstein, J., R. Qi, V. I. Ragulin, and S. A. Zenios. *Data-Parallel Implementation of Dense Linear Programming Algorithms.* Thinking Machines Corporation Technical Report TMC-230, 1992.

Chapter 12

Random Number Generators

This chapter describes the CM Fortran interface for the CMSSL random number generators (RNGs). One section is devoted to each of the following topics:

- introduction
- state tables
- safety checkpointing
- alternate stream checkpointing
- references

Man pages for the RNG routines follow these sections.

12.1 Introduction

The CMSSL includes two RNGs:

- the Fast RNG
- the VP RNG

These operations use a lagged-Fibonacci algorithm. They supplement the CM Fortran random number utility, **CMF_RANDOM**, and use a cellular automaton algorithm. (For a description of **CMF_RANDOM**, refer to the CM Fortran documentation set.)

12.1.1 The Fast RNG and the VP RNG Compared

To construct pseudo-random values, the CMSSL random number generators use *state tables* loaded from **CMF_RANDOM**. The difference between the Fast RNG and the VP RNG lies in the allocation of their state table arrays, as follows:

- The Fast RNG operation stores one state table per parallel processing node.
- The VP RNG operation stores one state table per destination array element.

The Fast RNG (so named because it is much faster than the Paris RNG originally used on the CM-2) thus consumes substantially less processing node memory than the VP RNG. The VP RNG can produce identical results on CM partitions of different sizes. Also, the VP RNG is slightly faster than the Fast RNG, albeit at potentially high cost in memory.

The VP RNG, which mimics the Fast RNG, is generally used if access to a CM partition of a specific size is not guaranteed. For instance, if you are using a 16-node partition while developing an application that will ultimately run on a 64-node partition, use the VP RNG to produce the same results that the Fast RNG will produce when the application is finally run on the larger partition. The VP RNG produces the same result for each array element, regardless of partition size or array shape — provided that the same *total* number of array elements is maintained and that the same seed is used.

12.1.2 The RNG Routines

Most applications that use the CMSSL RNG simply require you to call the Fast or VP RNG and then deallocate the state table. The following operations provide this functionality:

initialize_fast_rng
fast_rng
deallocate_fast_rng

initialize_vp_rng
vp_rng
deallocate_vp_rng

Note that explicitly initializing these RNGs is only necessary if the default table parameters are not suitable for your application. Section 12.2 discusses the state table parameters.

To guard against the effects of forced interruption, or to use more than one stream of random values, some applications require *checkpointing*. Checkpointing is the process of recording state information at a specific point in the random number stream generation. Later, the stream generation can be continued from the checkpoint.

In s*afety checkpointing*, you save the RNG state to a file in case of forced interruption — for instance, during long periods of application execution. If an interruption occurs, you can restore the state and continue processing. Safety checkpointing is accomplished with the following routines:

save_fast_rng_temps
restore_fast_rng_temps

save_vp_rng_temps
restore_vp_rng_temps

In *alternate stream checkpointing*, you save the RNG states associated with two distinct streams of random numbers in order to switch back and forth between the streams. Alternate stream checkpointing is accomplished with the following operations:

fast_rng_state_field
fast_rng_residue
reinitialize_fast_rng

vp_rng_state_field
vp_rng_residue
reinitialize_vp_rng

See Sections 12.3 and 12.4 for detailed descriptions of safety checkpointing and alternate stream checkpointing, respectively.

12.1.3 Implementation

The lagged-Fibonacci algorithm used by both CMSSL random number generators is widely used to produce a uniform distribution of random values. This implementation has been subjected to a battery of statistical tests, both on the stream of values within each processing node and for cross-node correlation. The only test that the CMSSL RNGs fail is the Birthday Spacings Test, as predicted by Marsaglia in the paper referenced in Section 12.5. Despite this failure, these lagged-Fibonacci RNGs are recommended for the most rigorous applications such as Monte Carlo simulations of lattice gases.

12.2 State Tables

If you do not initialize the Fast RNG or VP RNG explicitly, the CM Fortran interface uses default parameters to initialize the RNG automatically the first time you call it. If you want to use non-default state table parameters, you must initialize the VP or Fast RNG explicitly.

12.2.1 Fast RNG State Tables

The **initialize_fast_rng** routine allocates the state tables for the Fast RNG and initializes them with values generated by **CMF_RANDOM**. The application can provide a seed for **CMF_RANDOM** by calling **CMF_RANDOMIZE**. Figure 34 shows a Fast RNG state table. In the Fast RNG, one state table is associated with each processing node; from this table, random values are constructed for each subgrid element. (The *subgrid* associated with a processing node is the set of array elements residing in that node's memory.) Thus, each time you call **fast_rng**, the number of pseudo-random values produced by each state table is equal to the subgrid size.

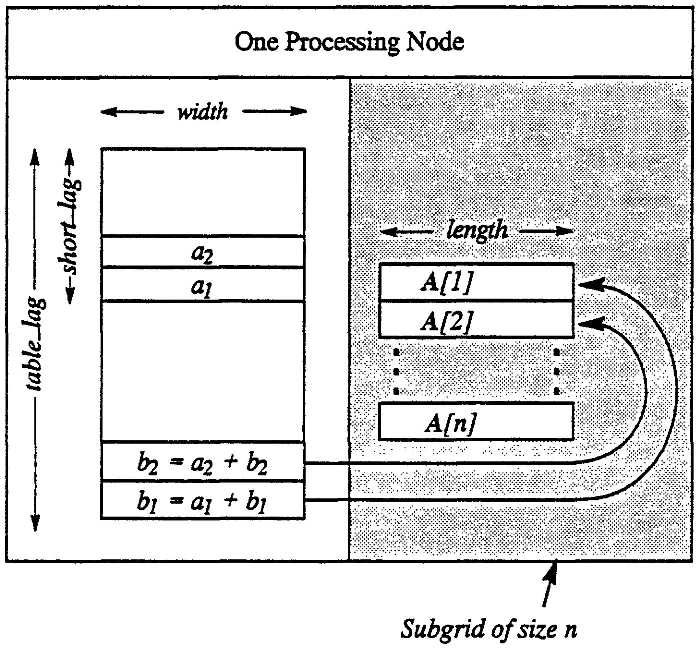

Figure 34. Fast RNG state table.

12.2.2 VP RNG State Tables

The **initialize_vp_rng** routine allocates the state tables for the VP RNG and initializes them with values generated by **CMF_RANDOM**. The application provides a seed for **CMF_ RANDOM** in the **initialize_vp_rng** *seed* argument. Figure 35, below, shows one VP RNG state table. In contrast to the Fast RNG state table, each subgrid element has its own state table. Each time you call **vp_rng**, each state table produces only one value.

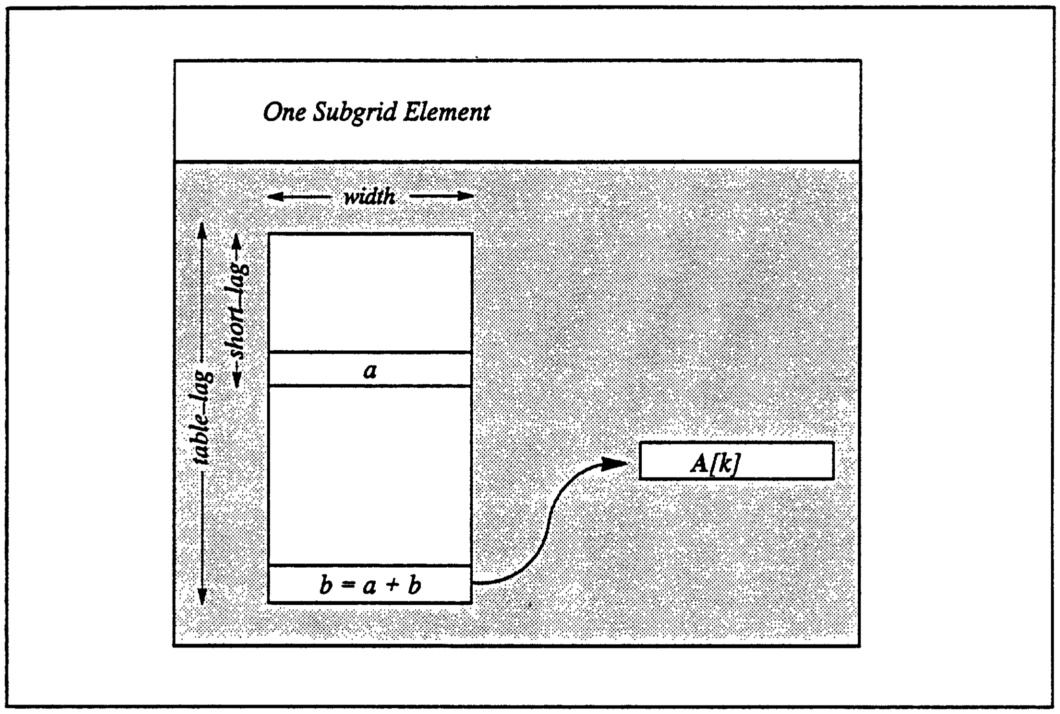

Figure 35. VP RNG state table.

If you do not call **initialize_vp_rng** explicitly, **vp_rng** selects a dynamic seed value based on the system time. (This is the conventional default initialization procedure in most random number generators.) As a result, with default initialization, different results are obtained in different runs. Explicit initialization is required to obtain identical results.

12.2.3 State Table Parameters

For either CMSSL RNG, each state table has *table_lag* elements and width *width*. The state table field is treated as a circular buffer. Internal variables are initialized so that they point into the table at offsets of *short_lag* and *table_lag* elements. The operation of generating a pseudo-random value (termed *stepping* the RNG) consists of adding the element at the *short_lag* pointer to the element at the *table_lag* pointer. This sum is shown as $b = a + b$ in the figures above, where the element shown as b is overwritten by the sum. All or a portion of the bits in element b are used to construct a pseudo-random value, which is then copied to the destination field. The pointers are then either decremented by one element or cycled around the buffer.

The state table parameters are critical to proper CMSSL RNG operation. The defaults used for automatic initialization are (17, 5) for *table_lag* and *short_lag*. For information about the defaults for *width*, refer to the man page for Fast RNG or VP RNG. Also, refer to the man pages for specific requirements and recommendations regarding *table_lag*, *short_lag*, and *width*.

The *period* of a random number generator is the number of discrete pseudo-random values it can generate before repeating the original stream. For applications that call a CMSSL RNG many times, it is important to make sure the period is long enough. The period should be greater than the total number of values produced by any one state table during all calls to the RNG. Otherwise, it may be possible to detect correlations within the stream produced by one processing node or between streams in different nodes. See the man page for Fast RNG or VP RNG, later in this chapter, for information on calculating an RNG period from different table parameters.

12.2.4 Need for Deallocation

Both the Fast RNG and the VP RNG allocate state tables on the CM heap; until explicitly deallocated, the tables occupy CM memory. The operations **deallocate_fast_rng** and **deallocate_vp_rng** deallocate the state table arrays. It is important that you use the deallocation routines, especially for the VP RNG, in which the state tables can occupy a significant amount of processing node storage space. (In the Fast RNG, the state table size is only on the order of *table_lag*.)

12.2.5 Parameters Saved During Checkpointing

Figure 36 shows how a CMSSL RNG state table is stepped. Cycling through the circular buffer requires *table_lag* steps. During one cycle, each element is used once as the a value and once as the b value in the computation $b = a + b$; thus, all element values are modified. Then the cycle repeats using the new values. The *residue* is the number of steps taken so far in a cycle. At any point, if you save the residue and the values in the state table, you can resume stream generation with the next step.

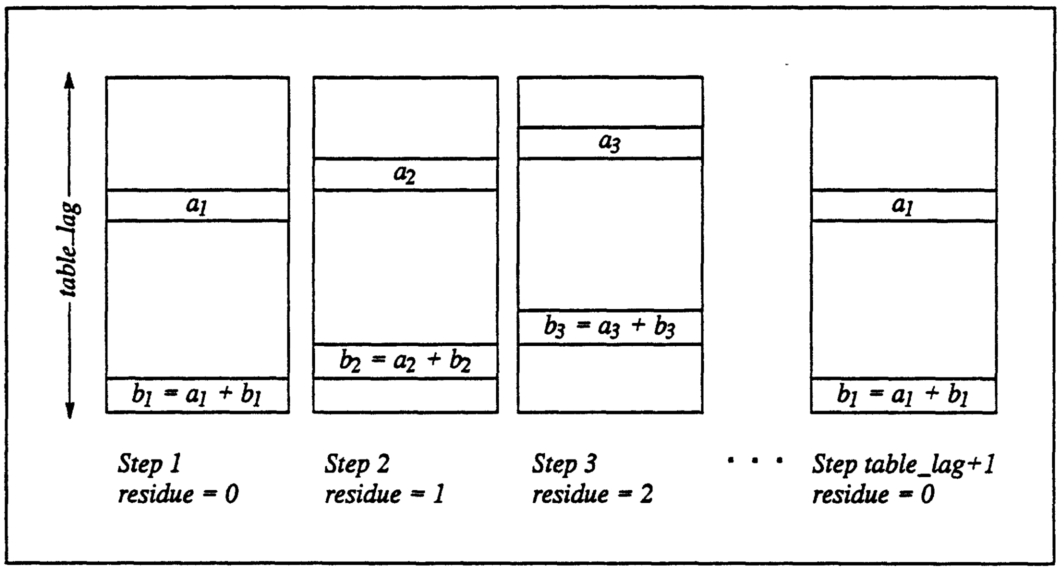

Figure 36. Stepping the state table.

The following routines save or return information for checkpointing:

- The **save_fast_rng_temps** and **save_vp_rng_temps** routines save the contents of the state table array, as well as all state table parameters (including the residue), to a file.
- The **fast_rng_state_field** and **vp_rng_state_field** routines return the state table array descriptor.
- The **fast_rng_residue** and **vp_rng_residue** routines return the residue value.

As explained in Sections 12.3 and 12.4, for safety checkpointing you must copy both the residue value and the contents of the state table array to a file; whereas for alternate-stream checkpointing, you save the residue value and the state table array descriptor (but not the actual contents of the array), and external storage is not required.

12.3 Safety Checkpointing

If you are running a long application that makes repeated use of one of the CMSSL RNGs, it is a good idea to safety checkpoint. Should a forced interruption occur, the application can be resumed at a checkpoint rather than having to be restarted from the beginning.

To perform safety checkpointing, you must use the save and restore routines,

save_fast_rng_temps	**save_vp_rng_temps**
restore_fast_rng_temps	**restore_vp_rng_temps**

To use this method, insert calls to the save routine periodically among your RNG calls. If a forced interruption occurs, call the restore routine, which restores the state table to exactly the values it had when the state was saved. No reinitialization is required; you can call the RNG again immediately after restoring the state. Figure 37 is a flowchart for safety checkpointing.

The Fast RNG generates different random number sequences for destination arrays on different CM partition sizes. Therefore, if you are using the Fast RNG, the partition size must be the same when you restore the Fast RNG state as when you saved it.

In contrast, the VP RNG state table reflects the underlying geometry of the output array originally used to initialize the table. Therefore, the dimensions of the output array must be the same when you restore the VP RNG state as when you saved it.

The save and restore routines use the CM file system (CMFS) to store files, and expect CMFS pathnames. Specific restrictions are included in the man pages, later in this chapter.

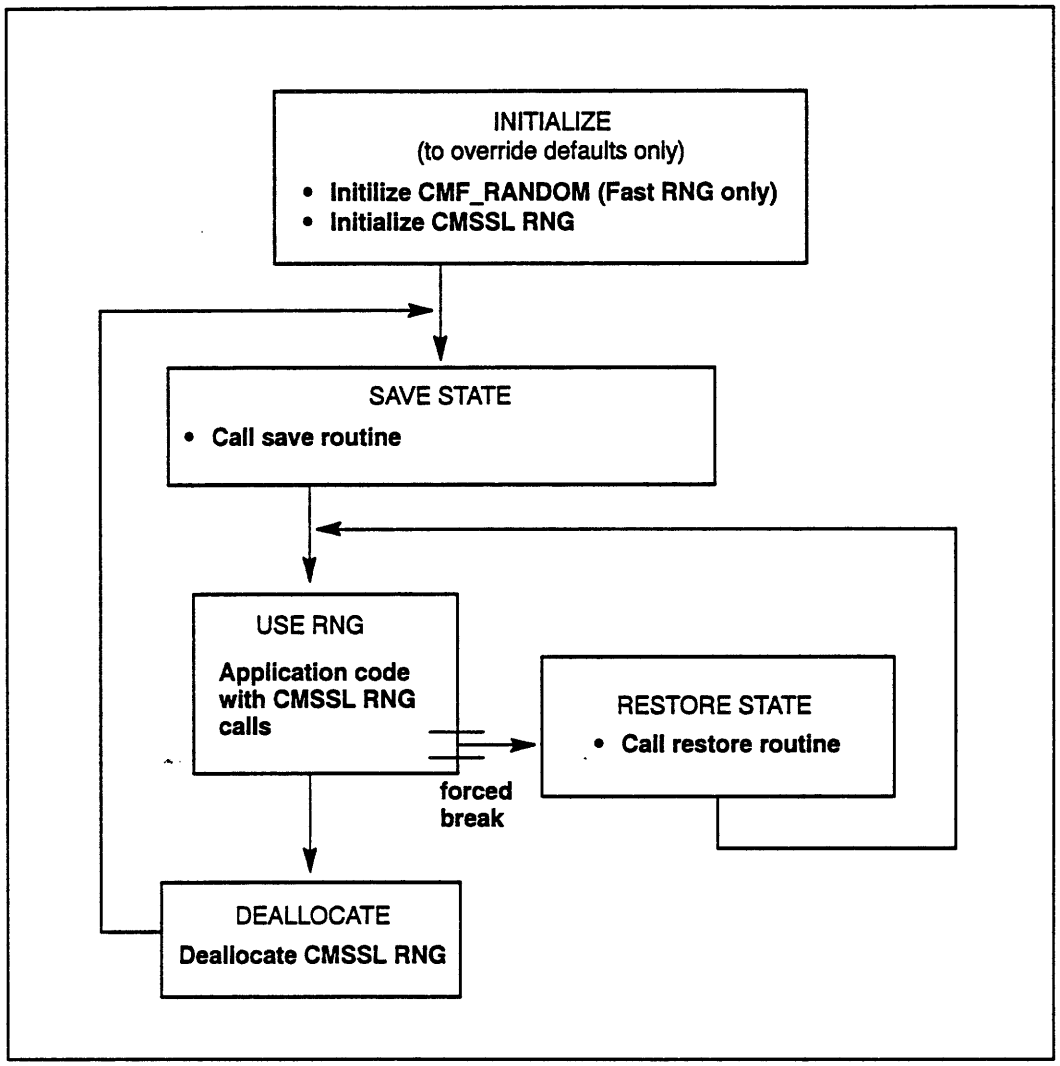

Figure 37. Safety checkpointing.

12.4 Alternate-Stream Checkpointing

It is possible to switch back and forth between two or more Fast or VP RNG streams by saving the state and reinitializing one RNG after the other. For instance, you can use this strategy to make separate random picks from two databases that need to be merged or compared in some way. You can also use it to increase efficiency when one stream of random numbers requires a wide state table and another can use a narrower state table.

If two or more VP RNG calls use destination arrays with different shapes, axis orderings, or axis weights, you *must* use alternate-stream checkpointing.

Figure 38 is a flowchart for alternate-stream checkpointing using two random number streams. The first RNG is initialized and its state is saved. Then the second RNG is initialized and its state is saved. To alternate between streams, first one and then the other RNG is looped through the reinitialization, use, and state-saving phases.

Unlike safety checkpointing, the state-saving phase of alternate-stream checkpointing does not require saving a copy of the state table field contents. Two state table fields are allocated during initialization; their array descriptors are used repeatedly during reinitialization, and finally both fields are deallocated.

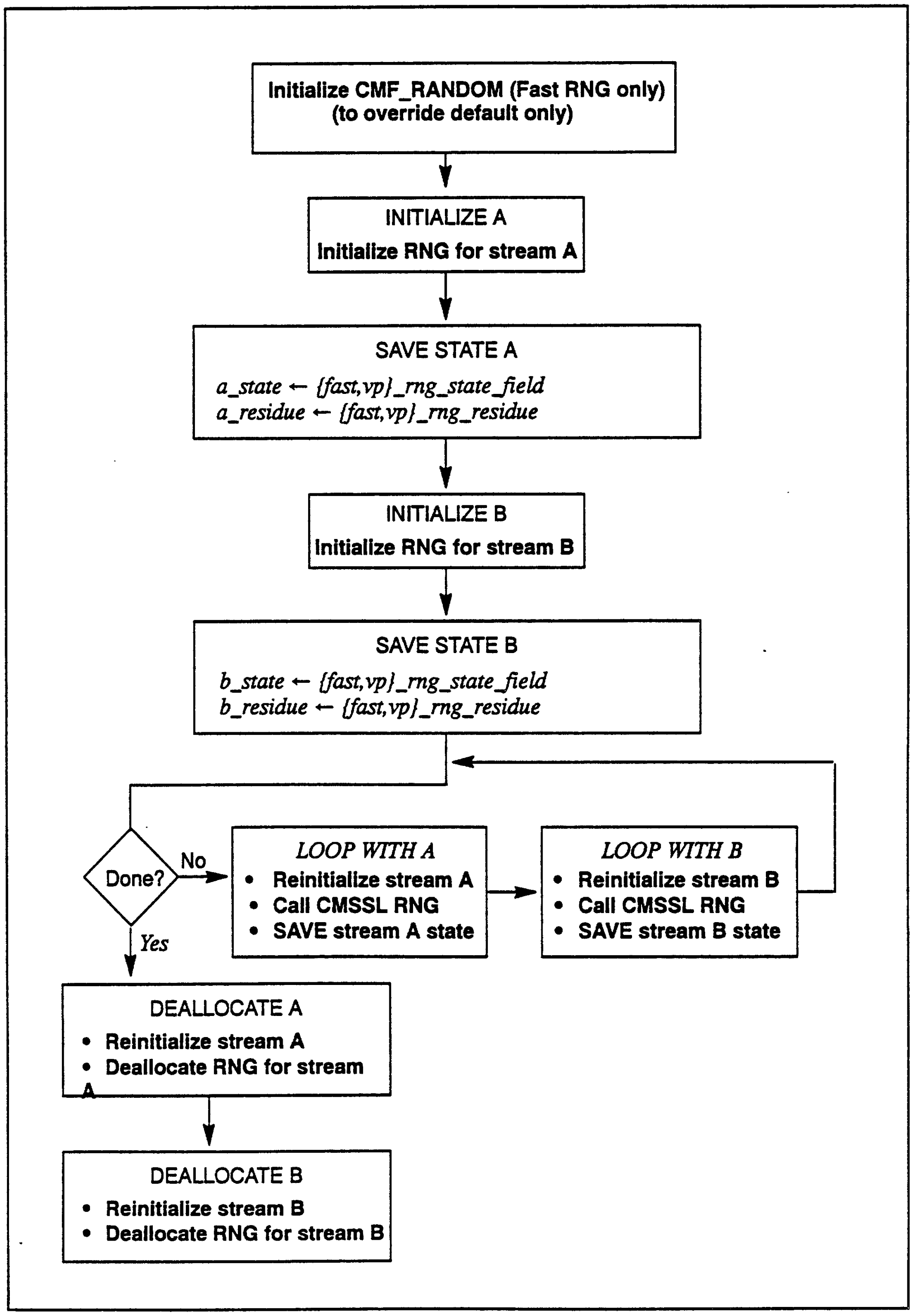
Initialize CMF_RANDOM (Fast RNG only) (to override default only)
INITIALIZE A
Initialize RNG for stream A
SAVE STATE A
a_state ← {fast,vp}_rng_state_field
a_residue ← {fast,vp}_rng_residue
INITIALIZE B
Initialize RNG for stream B
SAVE STATE B
b_state ← {fast,vp}_rng_state_field
b_residue ← {fast,vp}_rng_residue
Done?
No
Yes
LOOP WITH A
• Reinitialize stream A
• Call CMSSL RNG
• SAVE stream A state
LOOP WITH B
• Reinitialize stream B
• Call CMSSL RNG
• SAVE stream B state
DEALLOCATE A
• Reinitialize stream A
• Deallocate RNG for stream A
DEALLOCATE B
• Reinitialize stream B
• Deallocate RNG for stream B

Figure 38. Alternate-stream checkpointing.

12.5 References

For an analysis of the lagged-Fibonacci algorithm and a discussion of optimal lag parameter choices, see:

1. Knuth, D. *The Art of Computer Programming*, Vol. 2: *Seminumerical Algorithms*. Reading, Mass.: Addison-Wesley, 1973. Pp. 26–28.

For a discussion of the vulnerability of lagged-Fibonacci generators to the Birthday Spacings Test, see:

2. Marsaglia, G. A Current View of Random Number Generators. In *Computer Science and Statistics*, 16th Symposium on the Interface, Atlanta, March 1985.

Fast RNG

The CMSSL Fast RNG routines use a lagged-Fibonacci algorithm to generate pseudo-random numbers and store them in a destination array. Results may be integer values subject to a limit, or real values between 0.0 and 1.0.

SYNTAX

initialize_fast_rng (*table_lag, short_lag, width, ier*)

fast_rng (*A, limit, ier*)

save_fast_rng_temps (*file, ier*)

restore_fast_rng_temps (*file, ier*)

state_table = **fast_rng_state_field** (*ier*)

residue = **fast_rng_residue** (*ier*)

reinitialize_fast_rng (*table_lag, short_lag, width, state_table, residue, ier*)

deallocate_fast_rng (*ier*)

ARGUMENTS

table_lag	Scalar integer specifying the length of the state table. The default value for automatic initialization is 17. When you call **reinitialize_fast_rng,** supply the same value that was used in the original **initialize_fast_rng** call.
short_lag	Scalar integer used as an offset into the state table. Must be less than *table_lag*. The default value for automatic initialization is 5. When you call **reinitialize_fast_rng,** supply the same value that was used in the original **initialize_fast_rng** call.
width	Scalar integer specifying the width of the state table. Regardless of the value you supply, the RNG is *always* initialized with the following *width* values:

- 64, if the destination array is declared as double-precision real.

- 32, if the destination array is declared as single-precision real (unless you explicitly specify a *width* of 64, in which case the RNG is initialized with a *width* of 64).
- 32, if the destination array is declared as integer and *limit* is 0.
- 64, if the destination array is declared as integer and *limit* is not 0.

A	CM array of type real or integer. Upon successful completion, this array is overwritten with the RNG results.
limit	Scalar integer. Ignored for the real case. For the integer case, the exclusive, positive, upper bound on the pseudo-random values generated. A limit value of 0 is interpreted as allowing any 32-bit pattern, so that the full range of positive and negative integers is permitted.
file	Literal string or string variable declared, for example, character*(*). The name of the CMFS file in which to save the RNG state (when you are calling **save_fast_rng_temps**), or in which the state was already saved (when you are calling **restore_ fast_rng_temps**). If you do not supply a full pathname, the **DVHOSTNAME** and **DVWD** environment variables supply the defaults for the hostname and current directory name, respectively. (Refer to the *Connection Machine I/O System Programming Guide* for information about these variables.)
state_table	Array descriptor of a processing node heap field that contains the restored checkpointed values of a Fast RNG state table array. When you call **reinitialize_fast_rng,** supply the value returned by a previous call to **fast_rng_state_field.**
residue	Scalar integer returned by a previous call to the **fast_rng_residue** routine. Contains the checkpointed value of the Fast RNG state table residue that was current at the same execution point as the values identified by *state_table.*
ier	Scalar integer variable. Error code. Upon successful return from **initialize_fast_rng,** contains -1 if this initialization overwrote a previous initialization, or 0 if it did not. Upon successful return from **fast_rng,** contains -2 if default initialization was used, or 0 if default initialization was not used.

Upon return from **save_fast_rng_temps** or **restore_fast_rng_temps**, contains 0 if the routine was successful. If the code is non-zero, the upper 16 bits describe the operation that failed (see below) and the lower bits contain **CMFS_errno**. For **CMFS_errno** codes, see the man page for the corresponding CMFS library call in the *Connection Machine I/O System Programming Guide*.

Upper 16 bits	Operation	CMFS Library Call
1	open	**CMFS–open**
2	lseek	**CMFS–lseek**
4 or 8	write	**CMFS–write–file** (for **save_fast_rng_temps**)
4 or 8	read	**CMFS–read–file** (for **restore_fast_rng_temps**)
16	close	**CMFS–close**

Upon successful return from **fast_rng_state_field**, contains -2 if default initialization was used, or 0 if default initialization was not used.

Upon return from **fast_rng_residue**, contains 0 if the routine succeeded.

Upon successful return from **reinitialize_fast_rng**, contains -1 if this initialization overwrote a previous initialization, or 0 if it did not.

Upon return from **deallocate_fast_rng**, contains 0 if the routine succeeded or -1 if there is no previous state to deallocate.

RETURNED VALUE

state_table	Array descriptor of the current Fast RNG state table.
residue	Scalar integer indicating how many times the Fast RNG state has been stepped, modulo the *table_lag*.

DESCRIPTION

The Fast RNG is much faster than **CMF_RANDOM** and uses far less processing node memory than the VP RNG.

Usage. Follow these steps to use the Fast RNG:

1. Call **initialize_fast_rng** (optional). This step is required only if the default initialization parameters are not suitable for your application.
2. Call **fast_rng** to generate the pseudo-random numbers. You may repeat this step as many times as you wish, but the state table parameters with which the RNG was initialized (explicitly or by default) must be appropriate for each call. If a **fast_rng** call requires different state table parameters, you must initiate a new state table by calling **initialize_fast_rng** with the new parameter values.
3. After all **fast_rng** calls associated with one set of state table parameters have finished, call **deallocate_fast_rng** to deallocate the state table.

To perform safety checkpointing, use the **save_fast_rng_temps** and **restore_fast_rng_temps** routines. To perform alternate stream checkpointing, use the **fast_rng_state_field**, **fast_rng_ residue**, and **reinitialize_fast_rng** routines.

Initialization. The **initialize_fast_rng** routine allocates one Fast RNG state table as heap memory in the processing nodes. The state table is initialized with values generated by **CMF_RANDOM**. The initialization routine also initializes internal state, including the state table array descriptor referenced by **fast_rng_state_field** and the residue referenced by **fast_rng_residue**.

Separate state tables are allocated in each processing node. The length of the state table in bits per node is the product of the *table_lag* and *width* parameters. For example, given a *width* of 32 and the recommended *table_lag* of 17, the state table occupies 544 bits per node.

If your application requires a state table array configured differently from the default, you must call **CMF_RANDOMIZE** to initialize **CMF_RANDOM** with a seed, and then call **initialize_fast_rng**, before using **fast_rng**. This is important because **CMF_RANDOM** is used to fill the initial state table for the Fast RNG.

If you do not explicitly initialize **CMF_RANDOM** and the Fast RNG, initialization occurs automatically when you first call **fast_rng**. The Fast RNG selects a dynamic seed value based on the system time for the **CMF_RANDOM** seed. The other defaults used for automatic initialization are 17 for *table_lag* and 5 for *short_lag*; for *width* defaults, see the argument list above.

For reproducible results, use the same **CMF_RANDOM** seed and the same Fast RNG parameters each time. If you need reproducible results on different partition sizes, use the VP RNG.

The state table parameters have an enormous effect on the results obtained by calls to **fast_rng**. Use the following guidelines for proper state table initialization:

- The *period* of a random number generator is the number of random values it produces before repeating the original stream. To avoid correlation, the period should be greater than the total number of random values produced by any processing node; that is,

  $$\textit{desired-period} > \text{subgrid size} \times \textit{invocations}$$

  where *invocations* is the number of times the program calls **fast_rng**.

- The period is very sensitive to the choice of *table_lag* and *short_lag* values. The default pair of values, (17, 5), has been carefully chosen to produce the maximum period for the minimum storage. Other suggested values are (55, 24) and (71, 35). When the lag pairs are properly chosen, the period of the Fast RNG depends exponentially on the state table length (*table_lag*) and on the state table width (*width*), such that:

  $$\textit{period} = (2^{\textit{table\_lag}} - 1) \times 2^{\textit{width}}$$

  For a discussion on choosing proper lag value pairs, see the paper by Knuth referenced in Section 12.5.

- The RNG is *always* initialized with the following *width* values:

  - 64, if the destination array is declared as double-precision real.
  - 32, if the destination array is declared as single-precision real (unless you explicitly specify a *width* of 64, in which case the RNG is initialized with a *width* of 64).
  - 32, if the destination array is declared as integer and *limit* is 0.
  - 64, if the destination array is declared as integer and *limit* is not 0.

Random Number Generation. The **fast_rng** routine copies a pseudo-random value, chosen from a uniform distribution, into each element of *A*. The distribution range in the floating-point case is from 0.0 (inclusive) to 1.0 (exclusive). The distribution range in the integer case with a positive limit is from 0 (inclusive) to the specified *limit* (exclusive). The distribution range in the integer case with a 0 *limit* is all integer values from -2^{31} to $2^{31} - 1$ (in other words, any 32-bit pattern).

The Save and Restore Routines. The **save_fast_rng_temps** and **restore_fast_rng_temps** routines provide a mechanism for safety checkpointing, allowing an application to resume processing from a checkpoint after a forced interruption.

To use the save and restore routines, insert calls to the save routine periodically among your RNG calls. If a forced interruption occurs, call the restore routine, which restores the state table to exactly the values it had when the state was saved. No reinitialization is required; you can call the RNG again immediately after restoring the state. The partition size must be the same when you restore the fast RNG state as when you saved it.

The State Field, Residue, and Reinitialization Routines. The **fast_rng_state_field**, **fast_rng_residue**, and **reinitialize_fast_rng** routines provide a method for alternate-stream checkpointing, in which the application switches back and forth between two or more Fast RNG streams by saving the state and reinitializing one RNG after the other.

To perform checkpointing using these routines, follow these steps:

1. After the Fast RNG has been initialized either implicitly or explicitly by **initialize_fast_rng**, and after **fast_rng** has been called zero or more times, call **fast_rng_state_field** and **fast_rng_residue.**

2. Save the current state table array descriptor as a different array descriptor and save the current state field residue.

3. To restart a previously checkpointed Fast RNG number stream, call **reinitialize_fast_rng** with the array descriptor of the checkpointed state table field and residue.

The **fast_rng_state_field** routine returns the array descriptor of the Fast RNG state table. The array descriptor will have been created previously by an implicit or explicit call to **initialize_fast_rng**. The state table field resides in the processing nodes. Its length is the product of the *table_lag* and *width* parameters used in the **initialize_fast_rng** call.

The **fast_rng_residue** routine returns the residue: a count of the number of times that the Fast RNG state has been stepped, modulo the *table_lag*. The residue is the product of the subgrid size and the number of calls to **fast_rng** that have occurred since the last call to **initialize_fast_rng** or **reinitialize_fast_rng.**

The **reinitialize_fast_rng** routine reinitializes the Fast RNG from a previously checkpointed state so that an interrupted computation can be resumed.

Deallocation. The **deallocate_fast_rng** routine deallocates the heap field that has been used to store the state table for the Fast RNG. Call this routine when you are finished with the Fast RNG.

NOTES

Numerical Performance. The lagged-Fibonacci algorithm implemented by this RNG is widely used to produce a uniform distribution of random values.

For a table *width* of 32 and using the default *table_lag* and *short_lag* values, (17, 5), the period of the fast RNG is $(2^{17}-1)2^{32} \approx 5.6e15$ bits. By comparison, the period for **CMF_RANDOM** is estimated to be 6.8e10 bits, with greater danger of cross-node correlation.

Running time for the Fast RNG increases with the state table width and the number of bits used. For best results, reduce the table width to the number of bits required and use a *limit* value of 0.

Applications. The Fast RNG should be used in applications such as Monte Carlo simulations where speed is a priority and there is enough room for the state table.

Include the CMSSL Header File. The **fast_rng_residue** and **fast_rng_state_field** calls are functions; they return the residue and the array descriptor of the Fast RNG state table, respectively. Therefore, you must include the line

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

in program units that contain calls to these routines. This file declares the types of the CMSSL functions and symbolic constants.

Reproducible Results. To obtain reproducible results from the Fast RNG, initialize **CMF_RANDOM** with the same seed each time, and call **initialize_fast_rng** explicitly.

In contrast, checkpointing and reinitializing an RNG is used to *continue* random value stream generation from a previous or alternate state.

No Error Checking on Reinitialization. The **reinitialize_fast_rng** routine does not perform error checking on the input parameters. Unpredictible results or halted execution are likely under the following conditions:

- The length of *state_table* is less than (*table_lag* × *width*).
- *residue* is negative or *residue* ≥ *table_lag*.

EXAMPLES

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

```
random/cmf/
```

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

VP RNG

The CMSSL VP RNG routines use a lagged-Fibonacci algorithm to generate pseudo-random numbers and store them in a destination array. Results may be integer values subject to a limit, or real values between 0.0 and 1.0.

SYNTAX

initialize_vp_rng (*array, table_lag, short_lag, width, seed, ier*)

vp_rng (*A, limit, ier*)

save_vp_rng_temps (*file, ier*)

restore_vp_rng_temps (*file, ier*)

state_table = **vp_rng_state_field** (*array, ier*)

residue = **vp_rng_residue** (*array, ier*)

reinitialize_vp_rng (*table_lag, short_lag, width, state_table, residue, ier*)

deallocate_vp_rng (*array, ier*)

ARGUMENTS

array	Array descriptor for a CM array of the same shape and size as the *A* argument supplied to any **vp_rng** call for this RNG stream. Information about geometry and layout is taken from this argument.
table_lag	Scalar integer specifying the length of the state table. The default value for automatic initialization is 17. When you call **reinitialize_vp_rng**, supply the same value that was used in the original **initialize_vp_rng** call.
short_lag	Scalar integer used as an offset into the state table. Must be less than *table_lag*. The default value for automatic initialization is 5. When you call **reinitialize_vp_rng**, supply the same value that was used in the original **initialize_vp_rng** call.

width	Scalar integer specifying the width of the state table. Regardless of the value you supply, the RNG is *always* initialized with the following *width* values:

- 64, if the destination array is declared as double-precision real.
- 32, if the destination array is declared as single-precision real (unless you explicitly specify a *width* of 64, in which case the RNG is initialized with a *width* of 64).
- 32, if the destination array is declared as integer and *limit* is 0.
- 64, if the destination array is declared as integer and *limit* is not 0.

seed	Scalar integer used to initialize **CMF_RANDOM** for reproducible results. If you do not explicitly initialize the VP RNG, the VP RNG selects a dynamic seed value based on the system time.
A	Array descriptor for CM array of type real (single- or double-precision) or integer. Upon successful completion, this array is overwritten with the RNG results.
limit	Scalar integer. Ignored for the real case. For the integer case, the exclusive, positive, upper bound on the pseudo-random values generated. A limit value of 0 is interpreted as allowing any 32-bit pattern, so that the full range of positive and negative integers is permitted.
file	Literal string or string variable declared, for example, character*(*). The name of the CMFS file in which to save the RNG state (when you are calling **save_vp_rng_temps**), or in which the state was already saved (when you are calling **restore_vp_rng_temps**). If you do not supply a full pathname, the **DVHOSTNAME** and **DVWD** environment variables supply the defaults for the hostname and current directory name, respectively. (Refer to the *Connection Machine I/O System Programming Guide* for information about these variables.)
state_table	Array descriptor of a processing node heap field that contains the restored checkpointed values of a VP RNG state table. When you

call **reinitialize_vp_rng**, supply the value returned by a previous call to **vp_rng_state_field**.

residue Scalar integer returned by the **vp_rng_residue** routine. Contains the checkpointed value of the VP RNG state table residue that was current at the same execution point as the values identified by *state_table*.

ier Scalar integer variable. Error code. Upon successful return from **initialize_vp_rng**, contains -1 if this initialization overwrote a previous initialization, or 0 if it did not.

Upon successful return from **vp_rng**, contains -2 if default initialization was used, or 0 if default initialization was not used.

Upon return from **save_vp_rng_temps** or **restore_vp_rng_temps**, contains 0 if the routine was successful. If the code is non-zero, the upper 16 bits describe the operation that failed (see below) and the lower bits contain **CMFS_errno**. For **CMFS_errno** codes, see the man page for the corresponding CMFS library call in the *Connection Machine I/O System Programming Guide*.

Upper 16 bits	Operation	CMFS Library Call
1	open	**CMFS–open**
2	lseek	**CMFS–lseek**
4 or 8	write	**CMFS–write–file** (for **save_vp_rng_temps**)
4 or 8	read	**CMFS–read–file** (for **restore_vp_rng_temps**)
16	close	**CMFS–close**

Upon successful return from **vp_rng_state_field**, contains -2 if default initialization was used, or 0 if default initialization was not used.

Upon return from **vp_rng_residue**, contains 0 if the routine succeeded.

Upon successful return from **reinitialize_vp_rng**, contains -1 if this initialization overwrote a previous initialization, or 0 if it did not.

Upon return from **deallocate_vp_rng**, contains 0 if the routine succeeded or -1 if there is no previous state to deallocate.

RETURNED VALUE

state_table	Array descriptor of the current VP RNG state table.
residue	Scalar integer indicating how many times the VP RNG state has been stepped, modulo the *table_lag*.

DESCRIPTION

The VP RNG is useful for producing identical streams of random numbers on partitions of different sizes, as long as the number of array elements is constant. It is much faster than **CMF_RANDOM** and slightly faster than the Fast RNG; however, it uses far more processing node memory than either.

Usage. Follow these steps to use the VP RNG:

1. Call **initialize_vp_rng** (optional). This step is required only if the default initialization parameters are not suitable for your application.

2. Call **vp_rng** to generate the pseudo-random numbers. You may repeat this step as many times as you wish, but the state table parameters with which the RNG was initialized (explicitly or by default) must be appropriate for each call. If a **vp_rng** call requires different state table parameters, you must initiate a new state table by calling **initialize_vp_rng** with the new parameter values.

3. After all **vp_rng** calls associated with one set of state table parameters have finished, call **deallocate_vp_rng** to deallocate the state table.

To perform safety checkpointing, use the **save_vp_rng_temps** and **restore_vp_rng_temps** routines. To perform alternate stream checkpointing, use the **vp_rng_state_field**, **vp_rng_residue**, and **reinitialize_vp_rng** routines.

Initialization. The **initialize_vp_rng** routine allocates a VP RNG state table array as heap memory in the processing nodes. The state table is initialized with values generated by **CMF_RANDOM**. The initialization routine also initializes internal state, including the state table array descriptor referenced by **vp_rng_state_field** and the residue referenced by **vp_rng_residue**.

The VP RNG is designed to produce identical results on partitions of different sizes. To get identical results, be sure the following conditions are true:

- The size of *array* does not change between runs; this ensures that the same total number of array elements are used each time.

- The same *seed* is used each time. If you do not call **initialize_vp_rng** explicitly, **vp_rng** selects a dynamic seed value based on the system time. (This is the conventional default initialization procesdure in most random number generators.) As a result, with default initialization, different results are obtained in different runs. Explicit initialization is required to obtain identical results.

The *array* parameter is used to determine the array geometry in which random values are used. A separate state table is allocated for each destination array element. The length of the state table in bits per destination array element is the product of the *table_lag* and *width* parameters. For example, given a *width* of 32 and the recommended *table_lag* of 17, the state table occupies 544 bits per array element.

If your application requires a state table array configured differently from the default, you must call **initialize_vp_rng** (supplying a seed for **CMF_RANDOM** in the *seed* argument) before using **vp_rng**.

To produce the same results as the Fast RNG would produce, you must ensure that the total number of array elements used by the VP RNG equals the total number of processing nodes used by the Fast RNG. Also, the *seed* you use to initialize the VP RNG state table must be the same as the seed you use when you call **CMF_RANDOMIZE** before calling **initialize_fast_rng**.

If you do not explicitly initialize the VP RNG, initialization occurs automatically when you first call **vp_rng**. The VP RNG selects a dynamic *seed* value based on the system time. The other defaults used for automatic initialization are 17 for *table_lag* and 5 for *short_lag*; for *width* defaults, see the argument list above. The *A* supplied in the first invocation of **vp_rng** is used as the default *array*.

The state table parameters have an enormous effect on the results obtained by calls to **vp_rng**. Use the following guidelines for proper state table initialization:

- The *period* of a random number generator is the number of random values it produces before repeating the original stream. To avoid correlation, the period should be greater than the total number of random values produced for any subgrid element; that is,

  desired-period > *invocations*

  where *invocations* is the number of times the program calls **vp_rng**.

- The period is very sensitive to the choice of *table_lag* and *short_lag* values. The default pair of values, (17, 5), has been carefully chosen to produce the maximum period for the minimum storage. Other suggested values are (55, 24) and (71, 35). When the lag pairs are properly chosen, the period of the

VP RNG depends exponentially on the state table length (*table_lag*) and on the state table width (*width*), such that:

$$period = (2^{table\_lag} - 1) \times 2^{width}$$

For a discussion on choosing proper lag value pairs, see the paper by Knuth referenced in Section 12.5.

- The RNG is *always* initialized with the following *width* values:
  - 64, if the destination array is declared as double-precision real.
  - 32, if the destination array is declared as single-precision real (unless you explicitly specify a *width* of 64, in which case the RNG is initialized with a *width* of 64).
  - 32, if the destination array is declared as integer and *limit* is 0.
  - 64, if the destination array is declared as integer and *limit* is not 0.

Random Number Generation. The **vp_rng** routine copies a pseudo-random value, chosen from a uniform distribution, into each element of *A*. The distribution range in the floating-point case is from 0.0 (inclusive) to 1.0 (exclusive). The distribution range in the integer case with a positive limit is from 0 (inclusive) to the specified *limit* (exclusive). The distribution range in the integer case with a 0 *limit* is all integer values from -2^{31} to $2^{31}-1$ (in other words, any 32-bit pattern).

Each time the VP RNG is called, it generates one random value per subgrid element, and places the random values in *A*. For any one stream of random values, all *A* arguments must have the same shape, size, and axis orderings. To generate random values for destination arrays that differ in these attributes, use alternate-stream checkpointing (described below).

The Save and Restore Routines. The **save_vp_rng_temps** and **restore_vp_rng_temps** routines provide a mechanism for safety checkpointing, allowing an application to resume processing from a checkpoint after a forced interruption.

To use the save and restore routines, insert calls to the save routine periodically among your RNG calls. If a forced interruption occurs, call the restore routine, which restores the state table to exactly the values it had when the state was saved. No reinitialization is required; you can call the RNG again immediately after restoring the state. The dimensions of the destination array must be the same when you restore the VP RNG state as when you saved it.

The State Field, Residue, and Reinitialization Routines. The **vp_rng_state_field**, **vp_rng_residue**, and **reinitialize_vp_rng** routines provide a method for alternate-stream checkpointing, in which the application switches back and forth between two or more VP RNG streams by saving the state and reinitializing one RNG after the other.

To perform checkpointing using these routines, follow these steps:

1. After the VP RNG has been initialized either implicitly or explicitly by **initialize_vp_rng**, and after **vp_rng** has been called zero or more times, call **vp_rng_state_field** and **vp_rng_residue**.

2. Save the current state table array descriptor as a different array descriptor and save the current state field residue.

3. To restart a previously checkpointed VP RNG number stream, call **reinitialize_vp_rng** with the array descriptor of the checkpointed state table field and residue.

The **vp_rng_state_field** routine returns the array descriptor of the VP RNG state table. The array descriptor will have been created previously by an implicit or explicit call to **initialize_vp_rng**. The state table is a heap field that resides in the processing nodes. The length of the state table is the product of the *table_lag* and *width* parameters supplied to **initialize_vp_rng**.

The **vp_rng_residue** routine returns the residue: a count of the number of times that the VP RNG state has been stepped, modulo the *table_lag*. The residue is the product of the subgrid size and the number of calls to **vp_rng** that have occurred since the last call to **initialize_ vp_rng** (implicit or explicit) or to **reinitialize_vp_rng**.

The **reinitialize_vp_rng** routine reinitializes the VP RNG from a previously checkpointed state so that an interrupted computation can be resumed.

Deallocation. The **deallocate_vp_rng** routine deallocates the heap field that has been used to store the state table for the VP RNG. Call this routine when you are finished with the VP RNG. Using **deallocate_vp_rng** is important because the VP RNG state table field can use a significant amount of processing node memory.

NOTES

Numerical Performance. The lagged-Fibonacci algorithm implemented by this RNG is widely used to produce a uniform distribution of random values.

For a table *width* of 32 and using the default *table_lag* and *short_lag* values, (17, 5), the period of the VP RNG is $(2^{17}-1)2^{32} \approx 5.6\text{e}15$ bits. By comparison, the period for **CMF_RANDOM** is estimated to be 6.8e10 bits, with greater danger of cross-node correlation.

Running time for the VP RNG increases with the state table width and the number of bits used. For best results, reduce the table width to the number of bits required and use a *limit* value of 0.

Applications. The VP RNG mimics the Fast RNG. It should be used in applications such as Monte Carlo simulations where reproducible results across different partition sizes must be verified.

Include the CMSSL Header File. The **vp_rng_residue** and **vp_rng_state_field** calls are functions; they return the residue and the array descriptor of the VP RNG state table, respectively. Therefore, you must include the line

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

in program units that include calls to these routines. This file declares the types of the CMSSL functions and symbolic constants.

Reproducible Results. To obtain reproducible results from the VP RNG, call **initialize_vp_rng** using the same *seed* value each time.

In contrast, checkpointing and reinitializing an RNG is used to *continue* random value stream generation from a previous or alternate state.

No Error Checking on Reinitialization. The **reinitialize_vp_rng** routine does not perform error checking on the input parameters. Unpredictible results or halted execution are likely under the following conditions:

- The length of *state_table* is less than (*table_lag* × *width*).
- *residue* is negative or *residue* ≥ *table_lag*.

Use of CMF_RANDOM to Initialize the State Tables. The state table in each subgrid element is initialized as though it were a Fast RNG state table in a partition with size equal to the subgrid size times the number of processing nodes.

EXAMPLES

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

```
random/cmf/
```

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

Chapter 13

Statistical Analysis

This chapter describes the CMSSL statistical analysis routines. Currently included are two histogramming operations that summarize a specified CM array by the number of occurrences of each value or range of values. The following routines are provided:

histogram	Histograms all values in a data set.
histogram_range	Histograms designated ranges of values within a data set.

Histograms provide a statistical mechanism for simplifying data. They are generally used in applications that need to display or extract summary information. For particularly large data sets, **histogram_range** facilitates breaking data down into subranges, perhaps as a preliminary step before doing more detailed analysis of interesting areas.

In particular, histograms have many applications in image analysis and computer vision. For example, a technique known as histogram equalization computes a histogram of pixel intensity values in an image and uses it to rescale the original picture.

The CMSSL histogram operations treat the elements of a front-end array as a series of *bins*. In each bin a tally of CM field values or value ranges is stored. The number of histogram bins varies widely with the application, from a dozen tallies on a large process or a few dozen markers on a probability distribution to a few hundred intensity values in an image or a few thousand instruction codes in a performance analysis.

13.1 How to Histogram

Decide whether to use a simple or a ranged histogram: Consider the number of bits needed to represent the source values that are to be analyzed. For a simple histogram, one front-end array element (or *bin*) is required for each possible source value. Since 2^{length} possible values can be represented in *length* bits, a simple histogram requires 2^{length} front-end bins. From this, we see that 8-bit values can be histogrammed into 2^8 or 256 bins—a manageable number. However, a simple histogram of 16-bit values would require 2^{16} or 65,536 bins—which is probably too many for useful analysis.

A ranged histogram uses one bin for each range of source values. Use a ranged histogram to analyze source arrays that include large values. Determining the number of bins to use and the range to assign each bin is easily done using exponent arithmetic. Suppose for example we have 16-bit source values. Since $2^{16} = 2^8 \times 2^8$, we could use $2^8 = 256$ bins and tally a range of 2^8 values in each. When making these calculations, don't forget to account also for two tail bins—one for either end of the value range. (See the **histogram_ range** man page for more on tail bins.)

As a concrete illustration of histogramming techniques, consider summarizing the pixel information of a framebuffer image so that each bin reflects the number of pixels with a given intensity.

Suppose the intensity is encoded in a CM array of 8-bit elements—as for black and white images. The range of possible values is small, so a simple histogram can be used. The front-end array should have one element (or bin) for each possible source value, or $2^8 = 256$ elements. When we invoke the histogram routine

```
CALL histogram (fe_array, source_array, sbit_len, ier)
```

each bin tallies the number of pixels for one discrete intensity level. (Note: CM Fortran currently follows standard Fortran by supporting only integer values of 32 bits. Nonetheless a histogram call on 8-bit elements may be simulated by using a *length* argument of 8 to histogram the low-order 8 bits of each 32-bit element.)

If, however, the pixel intensity is encoded in a CM array of 24-bit elements—as it is for RGB color images—then a front-end array of $2^{24} = 16{,}777{,}216$ elements would be too large. (Also, since there are 4 bytes in a word, an array with 2^{24} words would occupy over 67 megabytes.) Fortunately, there are two alternatives:

1. Call a simple histogram on each 8-bit color subfield separately, as with the black and white case described above.

2. Use a range histogram as a microscope to close in on the interesting parts of the data, as described below.

To use a range histogram for this problem, try to factor the total 2^{24} range into manageable chunks. In particular, since $2^{24} = 2^{16} \times 2^{8}$, we could put 2^{16} values in each of 2^{8} middle range bins and leave the tail bins empty. Following this course, we can obtain an overview of all the data by invoking the range histogram routine

```
CALL historam_range (fe_array, source_array, 258, 0, 65536 )
```

In this call, we specify the number of bins as 258 (256 + 2 tail bins = 258), the range minimum as 0, and the range within each bin as 2^{16} = 65536.

Histogram

Increments each element of the specified front-end array by the number of times its index equals the value of an element in the specified source array.

For particularly large ranges of source data, consider using the **range_histogram** routine.

SYNTAX

histogram (*s*, *A*, *length*, *ier*)

ARGUMENTS

s	Front-end array of integers for storing the histogram. The number of elements in this array should be at least 2^{length}.
A	Source CM array of type integer.
length	Length in bits of the *A* values, or the number of bits of each *A* value that are to be considered by the histogram routine. The specified *length* should be no greater than *n* such that an *s* of 2^n elements can fit in partition manager memory.
ier	Error code. Scalar integer. Set to 0 if the routine succeeds.

DESCRIPTION

A tally is made of the number of times any particular integer *i* occurs as an *s* value. This tally is added to the value of the corresponding histogram bin, *s*[*i*]. For example, if *s* has been initialized to zero, and if *s* contains five elements that have the value 3, then *s*(3) receives the value 5.

The histogram thus records the distribution of values within one or more source arrays (one per call). Note that histogram bins (that is, elements in *s*) are not precleared automatically. This allows the successive collection of data from many arrays into one histogram.

The *length* value is the number of bits of each *A* value that are to be considered by the histogram routine. For example, if *A* contains 32-bit integers, a *length* value of 8 causes the values represented by the low 8 bits of each source value to be tallied. High-order bits are ignored.

NOTES

No Error Checking. No special error checking is done. Thus, there may be unpredictable damage if there are not enough destination bins to hold all values up through the maximum value of the source data.

Front-End Array Size. To avoid error, the correct number of front-end array elements is 2^{length}, where *length* is the number of bits used to represent each A value.

Performance. With the current implementation of this histogram routine, optimum performance is obtained using large source arrays and a small number of bins.

Zero-Based Array Indexing. Zero values are tallied in the first bin. For this reason, CM Fortran users might want to declare *s* to be zero-based.

EXAMPLES

Sample code that uses the histogram routines can be found on-line in the subdirectory

```
histogram/cmf/
```

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

Range Histogram

Increments each element of a front-end array by the number of times a source value falls into the subrange associated with that element.

This is particularly useful for obtaining a statistical summary of large data sets, for which unranged histogram information is too unwieldy to analyze. To histogram small data sets, consider using the **histogram** routine.

SYNTAX

histogram_range (*s, A, n, min, range, ier*)

ARGUMENTS

s	Front-end array of integers for storing the histogram. The number of elements in this array must be ≥ *n*.
A	Source CM array containing either real or integer values.
n	Scalar integer specifying the number of destination bins, including 2 bins for tail values.
min	Scalar integer specifying the minimum value for counting into the second bin.
range	Scalar integer specifying the size of the subrange associated with all but the first and last bins.
ier	Error code. Scalar integer. Normally set to 0.

DESCRIPTION

This subroutine treats each element of *s* as a bin associated with a range of *A* values. To each bin it assigns an integer value equal to the bin's initial value plus the number of values within the *A* array that fall into the bin's range.

There must be at least *n* number of elements in *s*. The range histogram uses the first bin, *s*[1], and the last bin, *s*[*n*], to tally *A* values below and above the range under inspection. The middle elements are used to tally *A* values within subranges that span successive increments of *range*. The *min* and *range* parameters determine the range of values contributing to each element of the histogram, as follows:

- The first bin, *s*[1], is incremented by the tally of all *A* values *j* for which $j < min$.
- Each middle bin, *s*[*i*], is incremented by the tally of *A* values *j* for which
  $j \geq (min + (i - 1) \times range)$
  and
  $j < (min + i \times range)$.
- The last bin, *s*[*n*], is incremented by the tally of *A* values *j* for which $j \geq (min + (n - 2) \times range)$.

The histogram thus records the distribution of values within one or more source arrays (one per call). Note that histogram bins are not precleared automatically. This allows the successive collection of data from many CM array fields into one histogram.

NOTES

Performance. With the current implementation of this histogram routine, optimum performance is obtained using large source arrays and a small number of bins.

EXAMPLES

Sample code that uses the histogram routines can be found on-line in the subdirectory

```
histogram/cmf/
```

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

Chapter 14

Communication Primitives

This chapter describes the following CMSSL communication primitives:

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

Sections 14.1 through 14.15 introduce these operations. Each section is followed by a man page that describes the routine(s) in detail and provides sample code. Section 14.16 provides references.

14.1 Polyshift

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

14.1.1 The Polyshift Routines

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

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

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

2. Call the **pshift** routine.

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

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

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

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

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

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

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

NOTE

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

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

14.1.2 Optimization Recommendations

Follow these recommendations to optimize your code:

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

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

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

  to increase the x-axis weight.

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

Polyshift

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

SYNTAX

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

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

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

deallocate_pshift_setup (*setup_id*)

ARGUMENTS

n Input. Scalar integer variable, parameter, or constant. The number of distinct shifts, of the shift types, dimensions, and distances that follow, to do in a single call to **pshift**. *n* must be greater than or equal to 0 and less than or equal to 14. The value you supply to

pshift must be the same as the value you supplied in the associated **pshift_setup** or **pshift_setup_looped** call. The contents of this argument are not modified.

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

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

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

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

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

0	Successful return.
2	*n* does not match the value supplied in the **pshift_setup** or **pshift_setup_looped** call.
3	Some *dim_i* is less than 1 or greater than 7.
4	The array geometry of some *A_i* does not match the geometry of the array supplied in the **pshift_setup** or **pshift_setup_looped** call.
5	The array geometry of some *B_i* does not match the geometry of the array supplied in the **pshift_setup** or **pshift_setup_looped** call.
6	Some *type_i* does not match the corresponding

shift type supplied in the **pshift_setup** or **pshift_setup_looped** call.

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

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

type_i

Input. One of the following predefined CMSSL shift types:

CMSSL_CSHIFT
CMSSL_EOSHIFT_0
CMSSL_EOSHIFT_1
CMSSL_EOSHIFT_SCALAR
CMSSL_EOSHIFT_ARRAY

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

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

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

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

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

A_i

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

B_i

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

dim_i	Input. Scalar integer variable, parameter, or constant. The dimension along which to perform the *i*th shift. The contents of this argument are not modified.
dist_i	Input. Scalar integer variable, parameter, or constant. The distance (number of elements) to shift for the *i*th shift. Positive or negative numbers, as defined in the **CSHIFT** and **EOSHIFT** entries of the *CM Fortran Reference Manual*. The contents of this argument are not modified.
bdry_i	Input. Front-end scalar or CM array. The boundary value shifted in if the *i*th shift type is **CMSSL_EOSHIFT_SCALAR** or **CMSSL_EOSHIFT_ARRAY**. The contents of this argument are not modified. If *type_i* is **CMSSL_EOSHIFT_SCALAR** or **CMSSL_EOSHIFT_ARRAY**, you must supply a *bdry_i* argument. If *type_i* is **CMSSL_CSHIFT**, **CMSSL_EOSHIFT_0**, or **CMSSL_EOSHIFT_1**, do not supply a *bdry_i* argument. (CSHIFT does not use a boundary value; **EOSHIFT_0** and **EOSHIFT_1** use boundary values 0 and 1, respectively.)

RETURNED VALUE

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

DESCRIPTION

The polyshift routines perform the following operation:

$$A_1(,,,,i,,,) = B_1(,,,,i+dist\_1,,,);$$
$$A_2(,,j,,,,,) = B_2(,,j+dist\_2,,,,,);$$
$$\vdots \qquad \vdots$$
$$A_n(,,,,,,k,) = B_n(,,,,,,k+dist\_n,)$$

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

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

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

2. Call the **pshift** routine.

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

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

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

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

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

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

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

NOTES

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

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

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

EXAMPLES

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

```
pshift/cmf
```

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

14.2 All-to-All Broadcast

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

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

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

As with all CMSSL functions, the all-to-all broadcast function operates on data arrays as they are allocated by the compiler. Consider a fixed data set M structured as a two-dimensional array $A(M/P, P)$ distributed over N processors, with the first axis of length M/P local to a processor. Assume that an all-to-all broadcast is performed along the second axis, which has local extent P/N. No local data motion is performed when $P = N$ since the second axis has no local component. When $P > N$, the broadcast operation is factored into a local broadcast operation of P/N steps for each exchange of $(M/P)(P/N)$ elements between processors.

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

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

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

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

Code 1

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

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

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

Code 2

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

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

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

14.2.1 The All-to-All Broadcast Routines

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

1. Call the **all_to_all_setup** routine.

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

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

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

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

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

```
call all_to_all_setup (..., step_count, ...)
              .
              .
do i = 1, step_count
   call all_to_all (...)
   where (...)
```

```
                  .
                  .
        elementwise computation
                  .
                  .
      end where
   end do
                  .
                  .
   call deallocate_all_to_all_setup (...)
```

All-to-All Broadcast

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

SYNTAX

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

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

deallocate_all_to_all_setup (*setup_id*)

ARGUMENTS

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

value of **.false.** indicates that for the particular partition size, array shape, and array layout directives you are using, you need not use ***valid_mask*** to contextualize the computation. For more information, refer to the description and notes below.

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

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

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

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

DESCRIPTION

The all-to-all broadcast routines perform the operation

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

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

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

1. Call the **all_to_all_setup** routine.

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

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

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

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

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

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

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

Broadcast Phase. Given a CM array that has been processed by a previous call to **all_to_all_setup**, and given the value assigned to *setup_id* by **all_to_all_setup**, the **all_to_all** routine permutes all elements of the array along the axis selected in the **all_ to_all_setup** call. The setup object identified by *setup_id* determines the permutation pattern.

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

Deallocation Phase. The **deallocate_all_to_all_setup** routine frees the partition manager and parallel processing node storage space that was allocated by the

all_to_all_setup routine for the all-to-all broadcast identified by *setup_id*. Call **deallocate_all_to_all_setup** when the all-to-all broadcast loop has finished. Supply the *setup_id* value returned by **all_to_all_setup**.

NOTES

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

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

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

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

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

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

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

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

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

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

you may optimize your code by using a sequence such as

```
if (use_valid) then
  where (valid_mask)
          .
          .
```

```
      elementwise computation
            .
            .
     end where
   else
            .
            .
      elementwise computation
            .
            .
   end if
```

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

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

EXAMPLES

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

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

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

The sample programs illustrate the following important distinction:

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

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

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

14.3 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. 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.

14.3.1 The Gather Utility Routines

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

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

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

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

14.3.2 Definition of the Gather Operation

The gather operation is defined by

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

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

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

14.3.3 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 vector or the p array.

Example 1: Elementwise Gather Operation

Given the source vector

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

the destination mask

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

and the pointers array

```
p = [1 5 7 1 3 2 2 4 1 - - - -]
```

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

```
y = [d i h d k g g b d - - - -].
```

Example 2: Finite-Element Type Application

Given the source vector

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

the destination mask

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

and the pointers array

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

the gather operation results in the destination array

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

Example 3: Unmasked Destination Array

Given the source vector

x = [d g k b]

and the pointers array

p = [1 4 3 1 3 2],

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

y = [d b k d k g].

Sparse Gather Utility

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

SYNTAX

sparse_util_gather_setup (*p, y_mask, x_template, trace, trace_mask, ier*)

sparse_util_gather (*y, x, trace, trace_mask*)

deallocate_gather_setup (*trace, trace_mask*)

ARGUMENTS

p	Integer CM array with the same axis extents and layout directives as *y*. *Must be one-based.* Indicates the gathering pattern. If an element of the array *p* contains the value *n*, then the corresponding element of the destination array receives the *n*th element of the source vector during the gather operation. The contents of *p* remain unchanged by **sparse_util_gather_setup.**
y_mask	If you need to mask elements of *y*, declare *y_mask* as a logical CM array with the same axis extents and layout directives as *y*; set to **.true.** the elements that correspond to active elements of *y*. The contents of *y_mask* remain unchanged by **sparse_util_gather_setup.** If you do not need to mask elements of *y*, you can conserve processing node memory by supplying the scalar logical value **.true.** for *y_mask.*
x_template	CM array with the same axis extent, layout directives, and data type as *x*. The setup routine uses only the shape and layout of this routine, ignoring the contents.
y	CM array of any type with arbitrary shape. Destination array to which source vector elements are gathered. The initial values of *y* are overwritten.

x	CM array of rank 1 and of the same type and precision as *y*. Source vector from which elements are gathered. The contents of *x* remain unchanged by **sparse_util_gather.**
trace	Scalar integer variable. Internal variable. The initial value you supply to **sparse_util_gather_setup** is ignored. You must supply **sparse_util_gather** and **deallocate_gather_setup** with the value that **sparse_util_gather_setup** assigns to *trace*.
trace_mask	Scalar integer variable. Internal variable. The initial value you supply to **sparse_util_gather_setup** is ignored. You must supply **sparse_util_gather** and **deallocate_gather_setup** with the value that **sparse_util_gather_setup** assigns to *trace_mask*.
ier	Scalar integer variable. Upon return from **sparse_util_gather_setup**, contains one of the following codes:

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

DESCRIPTION

Definition. The gather operation is defined by

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

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

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

1. Call **sparse_util_gather_setup.**

2. Call **sparse_util_gather.**

   To perform more than one gather operation using the same sparsity (gathering pattern), follow one call to **sparse_util_gather_setup** with multiple calls to **sparse_util_gather.** If the sparsity changes, start with Step 1 again.

3. After all **sparse_util_gather** calls associated with the same **sparse_util_gather_setup** call, call **deallocate_gather_setup** to deallocate the processing node storage space required by the setup routine.

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

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

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

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

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

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

NOTES

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

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

EXAMPLES

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

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

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

14.4 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. 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.

14.4.1 The Scatter Utility Routines

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

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

   To perform more than one scatter operation using the same sparsity (scattering pattern), follow one call to **sparse_util_scatter_setup** with multiple calls to **sparse_util_scatter**. If the sparsity changes, you must start with Step 1 again.
3. After all **sparse_util_scatter** calls associated with the same **sparse_util_scatter_setup** call, call the **deallocate_scatter_setup** routine to deallocate the processing node 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 node memory. It is therefore strongly recommended that you call **deallocate_scatter_setup** as soon as you have finished the associated scatter operations.

14.4.2 Definition of the Scatter Operation

The scatter operation is defined by

where $(\boldsymbol{x\_mask})$ $\quad y(p|+) = x$

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

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

14.4.3 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 array

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

with mask

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

and p array

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

and given the initial destination vector

y = [v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8]

the scatter operation results in destination vector

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]

Sparse Scatter Utility

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

SYNTAX

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

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

deallocate_scatter_setup (*setup*)

ARGUMENTS

p	Integer CM array with the same axis extents and layout directives as *x*. *Must be one-based.* Indicates the scattering pattern. If an element of the array *p* (as supplied to **sparse_util_scatter_setup**) contains the value *n*, then the scatter operation adds the corresponding source array element to destination vector element *n*. *If two or more source array elements are sent to the same destination vector element, the colliding destination values are added.* The contents of *p* are modified by a successful call to **sparse_util_scatter_setup**. When you call **sparse_util_scatter**, supply the values assigned to *p* by **sparse_util_scatter_setup.**
y_template	CM array with the same axis extent, layout directives, and data type as *y*. The setup routine uses only the shape and layout of this routine, ignoring the contents.
x_mask	If you need to mask elements of *x*, declare *x_mask* as a logical CM array with the same axis extents and layout directives as *x*; set to **.true.** the elements that correspond to active elements of *x*. Only those source array elements for which the mask is true are sent to the destination vector. Elements of *p* corresponding to masked elements of the source array are ignored. The contents of *x_mask* remain unchanged by **sparse_util_scatter_setup.** You may set any component of *x_mask* to **.false.** during the course of the computation without calling the setup routine again. However, a component of *x_mask* that has been set to **.false.**

before the **sparse_util_scatter_setup** call cannot be set to **.true.** during the computation.

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

y	CM array of any type and rank 1. The **sparse_util_scatter** routine adds the scattered x elements to the initial values of y.
x	CM array of arbitrary shape and of the same type and precision as y. Source array from which elements are scattered. The contents of x remain unchanged by **sparse_util_scatter.**
setup	Scalar integer variable. Internal variable. The initial value supplied to **sparse_util_scatter_setup** is ignored. You must supply **sparse_util_scatter** and **deallocate_scatter_setup** with the value assigned to *setup* by **sparse_util_scatter_setup.**

DESCRIPTION

Definition. The scatter operation is defined by

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

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

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

1. Call **sparse_util_scatter_setup.**

2. Call **sparse_util_scatter.**

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

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

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

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

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

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

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

EXAMPLES

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

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

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

14.5 Sparse Vector Gather Utility

The sparse vector gather utility, **sparse_util_vec_gather**, performs the same operation as the sparse gather utility, **sparse_util_gather**, except that **sparse_util_vec_gather** operates on vectors rather than individual data elements. The vectors that are gathered must lie along the left-most axis (which must be declared **:serial**) in both the source array and the destination array.

14.5.1 Definition of the Vector Gather Operation

The vector gather operation is defined by

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

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

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

14.5.2 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 array or the pointers array.

Example 1: Vector Gather Operation with Masked Destination Array

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

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

If the destination mask is

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

and the pointers array is

p = [1 5 7 1 3 2 2 4 1 - - - -]

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

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

Example 2: Vector Gather Operation with Unmasked Destination Array

Given the source array

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

and the pointers array

p = [1 5 7 1 3 2 2 4 1],

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

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

Sparse Vector Gather Utility

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

SYNTAX

sparse_util_vec_gather_setup (*p, y_mask, x_template, trace, trace_mask, ier*)

sparse_util_vec_gather (*y, x, trace, trace_mask*)

deallocate_vec_gather_setup (*trace, trace_mask*)

ARGUMENTS

p	Integer CM array with the same axis extents and layout directives as the subarray of *y* formed by omitting the left-most axis of *y*. *Must be one-based.* Indicates the gathering pattern. If element (p_1, ..., p_k) of the array ***p*** contains the value *n*, then the vector ***x***(:, *n*) is gathered from the source array to locations $y(:, p_1, ..., p_k)$ in the destination array during the gather operation. The contents of ***p*** remain unchanged by **sparse_util_vec_gather_setup.**
y_mask	If you need to mask elements of *y*, declare ***y_mask*** as a logical CM array with the same axis extents and layout directives as the subarray of *y* formed by omitting the left-most axis of *y*; set to **.true.** the elements that correspond to active vectors within *y*. The contents of ***y_mask*** remain unchanged by **sparse_util_vec_gather_ setup.** If you do not need to mask elements of *y*, you can conserve processing node memory by supplying the scalar logical value **.true.** for ***y_mask.***
x_template	CM array with the same axis extent, layout directives, and data type as the subarray of *x* formed by omitting the left-most axis of *x*. The setup routine uses only the shape and layout of this array, ignoring the contents.
y	CM array of any type with arbitrary shape. Destination array to which vectors from the source array are gathered. The gathered

vectors will lie along the left-most axis of *y*. This axis must be declared **:serial**, and must have the same extent as the left-most axis of *x*. The initial values of *y* are overwritten.

x CM array of rank 2 and of the same type and precision as *y*. Source array from which vectors are gathered. The vectors to be gathered must lie along the left-most axis of *x*. This axis must be declared **:serial**, and must have the same extent as the left-most axis of *y*. The contents of *x* remain unchanged by **sparse_util_vec_gather**.

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

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

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

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

DESCRIPTION

Definition. The vector gather operation is defined by

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

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

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

1. Call **sparse_util_vec_gather_setup.**

2. Call **sparse_util_vec_gather.**

To perform more than one vector gather operation using the same sparsity (gathering pattern), follow one call to **sparse_util_vec_gather_setup** with multiple calls to **sparse_util_vec_gather.** If the sparsity changes, start with Step 1 again.

3. After all **sparse_util_vec_gather** calls associated with the same **sparse_util_vec_gather_setup** call, call **deallocate_vec_gather_setup** to deallocate the processing node storage space required by the setup routine.

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

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

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

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

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

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

NOTES

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

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

EXAMPLES

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

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

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

14.6 Sparse Vector Scatter Utility

The sparse vector scatter utility, **sparse_util_vec_scatter**, performs the same operation as the sparse scatter utility, **sparse_util_scatter**, except that **sparse_util_vec_scatter** operates on vectors rather than individual data elements. The vectors that are scattered must lie along the left-most axis (which must be declared **:serial**) in both the source array and the destination array.

14.6.1 Definition of the Vector Scatter Operation

The scatter operation is defined by

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

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

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

14.6.2 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 array x has dimensions (3, 6). The vectors (of length 3) lie along the first axis, which is **:serial**:

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

If ***x_mask*** is

x_mask = [T T T T F],

the pointers array is

$p = [1 \;\; 4 \;\; 2 \;\; 2 \;\; -]$,

and the initial destination array is

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

then the vector scatter operation results in destination array

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

Sparse Vector Scatter Utility

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

SYNTAX

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

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

deallocate_vec_scatter_setup (*setup*)

ARGUMENTS

p	Integer CM array with the same axis extents and layout directives as the subarray of *x* formed by omitting the left-most axis of *x*. *Must be one-based.* Indicates the scattering pattern. If element (p_1, ..., p_k) of the array *p* (as supplied to **sparse_util_vec_scatter_setup**) contains the value *n*, then the scatter operation adds the vector *x*(:, p_1, ..., p_k) to the vector *y*(:, *n*) within the destination array. *If two or more source array elements are sent to the same destination array element, the colliding destination values are added.* The contents of *p* remain unchanged by **sparse_util_vec_scatter_setup.**
y_template	CM array with the same axis extent, layout directives, and data type as the subarray of *y* formed by omitting the left-most axis of *y*. The setup routine uses only the shape and layout of this array, ignoring the contents.
x_mask	If you need to mask elements of *x*, declare *x_mask* as a logical CM array with the same axis extents and layout directives as the subarray of *x* formed by omitting the left-most axis of *x*; set to **.true.** the elements that correspond to active vectors within *x*. Only those vectors for which the mask is true are sent to the destination array. Elements of *p* corresponding to masked locations of the source array are ignored. The contents of *x_mask* remain unchanged by **sparse_util_vec_scatter_setup.**

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

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

y CM array of any type and rank 2. The **sparse_util_vec_scatter** routine adds the scattered vectors from *x* to the initial values of the vectors that lie along the left-most axis of *y*. This axis must be declared **:serial**, and must have the same extent as the left-most axis of *x*.

x CM array of arbitrary shape and of the same type and precision as *y*. Source array. The vectors to be scattered must lie along the left-most axis of *x*. This axis must be declared **:serial**, and must have the same extent as the left-most axis of *y*. The contents of *x* remain unchanged by **sparse_util_vec_scatter.**

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

DESCRIPTION

Definition. The scatter operation is defined by

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

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

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

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

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

3. After all **sparse_util_vec_scatter** calls associated with the same **sparse_util_vec_scatter_setup** call, call the **deallocate_vec_scatter_setup** routine to deallocate the processing node storage space required by the other two routines.

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

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

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

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

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

EXAMPLES

Sample CM Fortran code that uses the sparse vector gather and scatter utilities can be found on-line in the subdirectory `sparse-utilities/cmf/` of a CMSSL examples directory whose location is site-specific.

14.7 Vector Move (Extract and Deposit)

The **vector_move** routine moves a vector from a source array to a destination array of the same rank, data type, and node layout. The **vector_move_utils** routine returns node layout and subgrid shape information for any CM array.

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

Vector Move (Extract and Deposit)

The **vector_move** routine moves a vector from a source array to a destination array of the same rank, data type, and node layout. The **vector_move_utils** routine returns node layout and subgrid shape information for any CM array.

SYNTAX

vector_move (*y, x, y_coords, y_axis, x_coords, x_axis, ier*)

vector_move_utils (*array, subgrid_extents, node_layout, ier*)

ARGUMENTS

y	One-based CM array of any data type.
x	One-based CM array of the same rank and data type as *y*. The *x* and *y* arrays must have the same node layout; otherwise, the code -1 is returned in *ier*. In this context, *node* refers to a processing element (PE) on a CM-200 or CM-2; a processing node (PN) on a CM-5 without vector units; or a vector unit on a CM-5 with vector units. *Node layout* refers to the number of nodes along each array axis. The extent of axis *x_axis* in *x* must equal the extent of axis *y_axis* in *y*.
y_coords	One-based integer front-end array of rank 1 and extent equal to the rank of *y*. Defines the destination vector to which the source vector will be moved. This destination vector is formed by holding the 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.
y_axis	Scalar integer variable. Identifies the axis in *y* along which the destination vector lies. Must satisfy $1 \leq$ *y_axis* $\leq$ rank(*y*).
x_coords	One-based integer front-end array of rank 1 and extent equal to the rank of *x*. Defines the source vector to be moved to the *y* array. This source vector is formed by holding the 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.
x_axis	Scalar integer variable. Identifies the axis in *x* along which the source vector lies. Must satisfy $1 \leq$ *x_axis* $\leq$ rank(*x*).
array	CM array whose node layout and subgrid shape is required.
subgrid_extents	Integer front-end array of rank 1 and extent equal to the rank of ***array***. On successful return, *subgrid_extents*(*i*) contains the subgrid extent of ***array*** along axis *i*.
node_layout	Integer front-end array of rank 1 and extent equal to the rank of ***array***. On successful return, *node_layout*(*i*) contains the number of nodes along axis *i* of ***array***.
ier	Scalar integer variable. Error code. Set to 0 on successful return, or to -1 otherwise.

DESCRIPTION

The **vector_move** routine moves a source vector from *x* to a destination vector in *y*. Each element in the destination vector is overwritten by the corresponding source value.

The **vector_move_utils** routine returns node layout and subgrid shape information for any CM array. The *subgrid_extents* argument returns the subgrid extent of the array along each axis; the *node_layout* argument returns the number of nodes along each axis. In this context, *node* refers to a processing element (PE) on a CM-200 or CM-2; a processing node (PN) on a CM-5 without vector units; or a vector unit on a CM-5 with vector units.

The on-line examples show how to use the **vector_move** and **vector_move_utils** routines to extract and deposit rows and columns of a two-dimensional array, and how to use **vector_move** to extract a face from a three-dimensional array.

Consider a 64 x 64 source array *A* and a 64 x 16 destination array *C*. Further assume a 4 x 8 node layout underlying both *A* and *C*. The subgrid of *A* on each node is 16 x 8; the subgrid of *C* on each node is 16 x 2. The following CM Fortran expression extracts the third column of the subgrid of *A* on each node and deposits it in the second column of the subgrid of *C*:

```
      C(:,2:16:2) = A(:,3:64:8)
```

The same functionality can be achieved by calling **vector_move** with the following arguments:

```
integer y_coords(2), x_coords(2)
integer y_axis, x_axis

y_coords(2) = 2
x_coords(2) = 3
y_axis = 1
x_axis = 1
vector_move(C, A, y_coords, y_axis, x_coords, x_axis, ier)
```

NOTES

One-Based Arrays. The arrays *x*, *y*, *x_coords*, and *y_coords* must be one-based.

Node Layout. The node layout must be identical for *x* and *y*. An error is signalled if this is not the case. (*Node layout* refers to the number of nodes along each array axis.) Use the detailed layout available in CM Fortran Version 2.0 or 2.1 to ensure that the node layouts of *x* and *y* are identical. The on-line examples illustrate the use of the detailed layout directives in this context. You can also obtain the node layout and subgrid shape using **vector_move_utils.**

EXAMPLES

Sample CM Fortran program that uses the extract and deposit routines can be found on-line in the subdirectory

```
vector-move/cmf/
```

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

14.8 Block Gather and Scatter Utilities

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

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

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

The man page at the end of this section provides detailed information about these routines and their arguments. Refer to the argument list when examining the following three figures. Figure 39 shows a block gather operation; Figure 40 shows a block scatter operation. In these figures, the source and destination arrays are two-dimensional; axis *vector_axis* is vertical. Only a single instance is pictured.

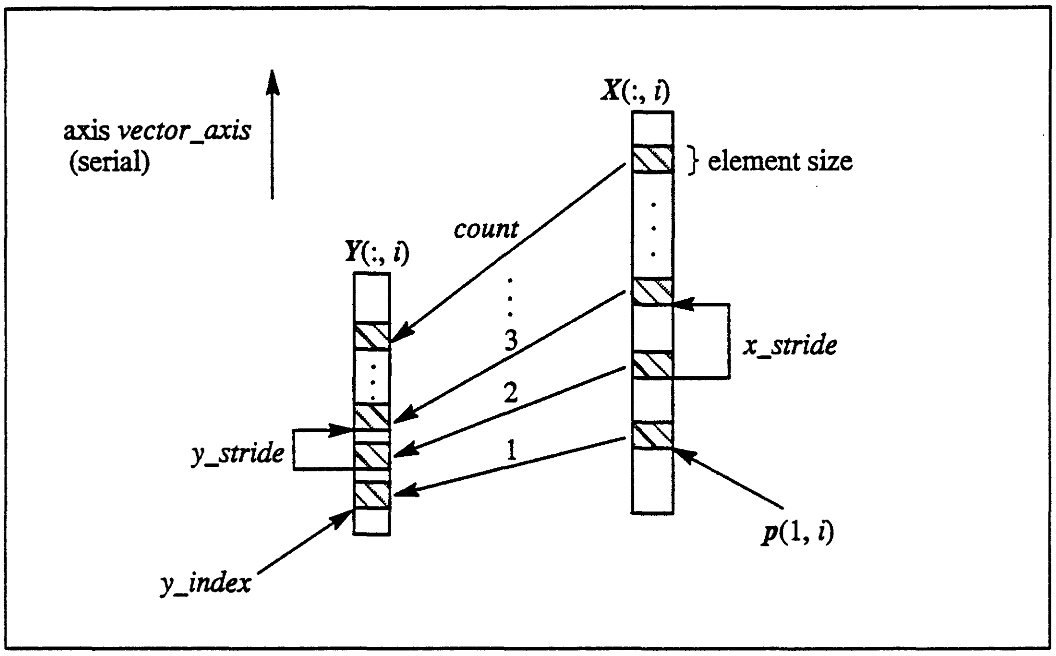

Figure 39. A block gather operation. The source and destination arrays are 2-dimensional. The vertical axis is the selected serial axis. Only a single instance is shown.

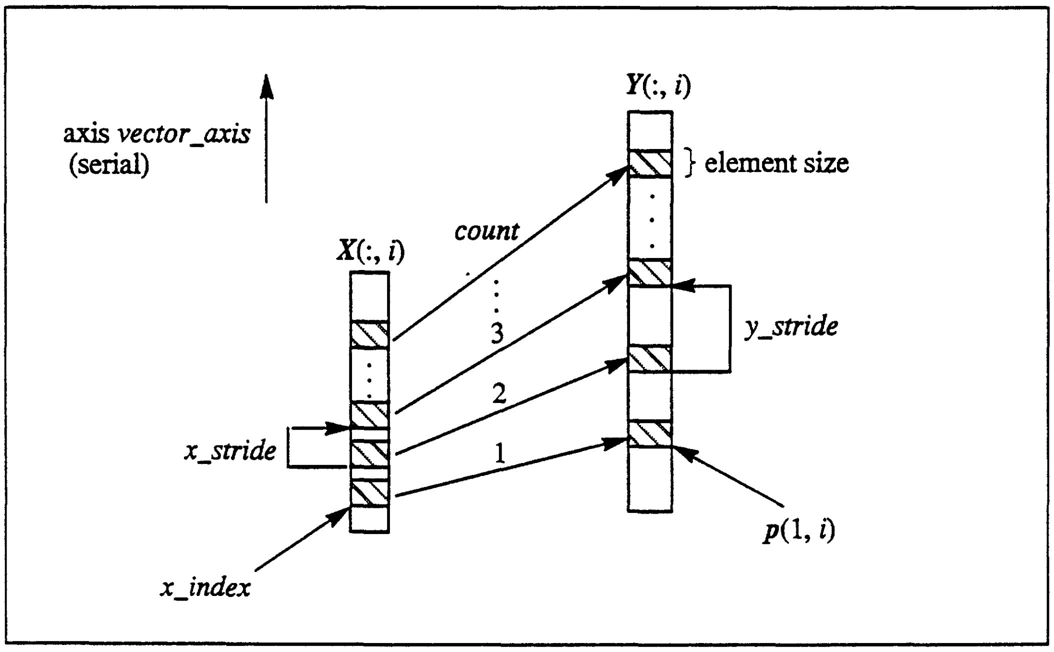

Figure 40. A block scatter operation. The source and destination arrays are 2-dimensional. The vertical axis is the selected serial axis. Only a single instance is shown.

Figure 41 shows two instances of the same block scatter operation pictured in Figure 40. 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.

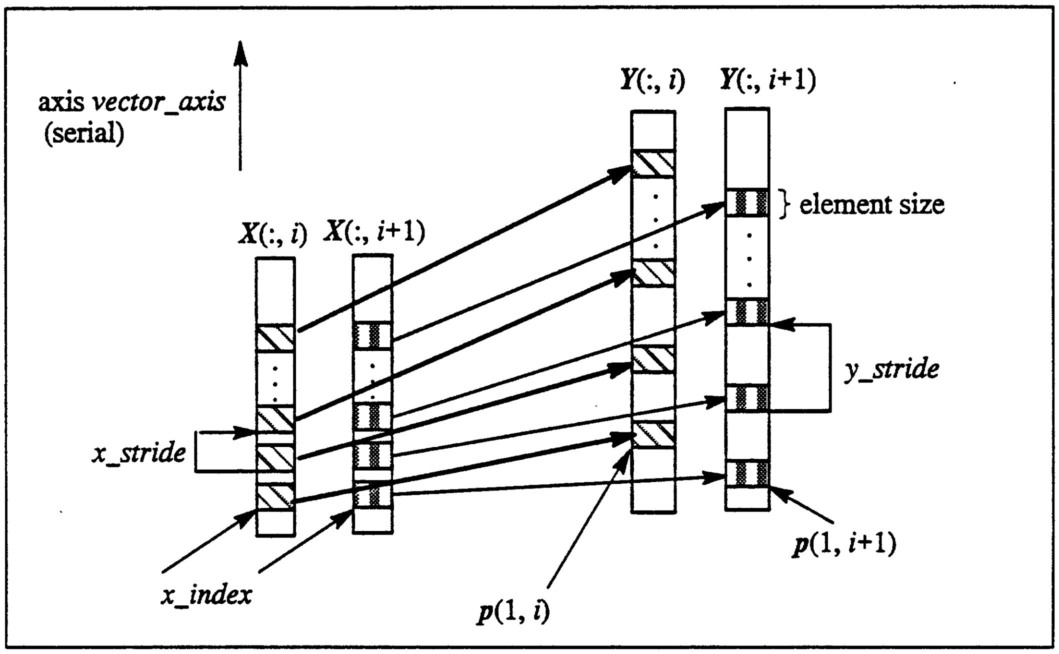

Figure 41. Two instances of the block scatter operation from Figure 40.

Block Gather and Scatter Utilities

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

SYNTAX

block_gather (*Y, y_stride, y_index, X, x_stride, p, count, vector_axis*)
block_scatter (*Y, y_stride, p, X, x_stride, x_index, count, vector_axis*)

ARGUMENTS

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

gap of (*x_stride* - 1) elements will be left between outgoing source data elements. If *x_stride* = 1, source data elements are moved from locations that are contiguous along axis *vector_axis*.

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

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

$$p(u_1, ..., u_{vector\_axis-1}, 1, u_{vector\_axis+1}, ..., u_n)$$

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

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

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

vector_axis Scalar integer. Identifies the serial axis along which data is to be moved from *X* to *Y*.

DESCRIPTION

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

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

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

Given *X* and *Y* arrays of rank 2, with one axis serial and the other parallel with length equal to the number of processing nodes, the following CM Fortran pseudo code describes the gather operation:

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

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

The effect of this code is to copy a block of *count* elements of *X* (strided by *x_stride* elements and starting at a different starting index in each node) 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 node.

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

Block gather: $Y(,,,,i,,,,) = X(,,,, p(j),,,,)$
Block scatter: $Y(,,,,p(j),,,,) = X(,,,, i,,,,)$

where

$y\_index \le i \le (count-1) * y\_stride$
$j = \text{mod}(i, N)$
N = number of processing nodes along axis *vector_axis*
(for CM-5 systems without vector units)
N = number of vector units along axis *vector_axis*
(for CM-5 systems with vector units)

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 array. While some safety checking will occur when safety is enabled, it will not be able to catch all misuses of the function, so users of these functions should be sure they stay within the bounds of the arrays.

EXAMPLES

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

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

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

14.9 Partitioning of an Unstructured Mesh and Reordering of Pointers

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

generate_dual	Given an *element nodes* array, ***ien***, that describes an unstructured mesh, this routine produces the corresponding *dual connectivity* array, ***idual***.
partition_mesh	Given a dual connectivity array describing an unstructured mesh, this routine returns a permutation, ***q***, that reorders the mesh elements to form discrete *partitions*. The routine also returns the number of resulting partitions and the number of elements per partition.
reorder_pointers	Given a pointers array, ***p***, and a permutation ***q*** (for example, the permutation returned by **partition_mesh**), this routine reorders the pointers array along its last axis using ***q***.

If you derive the pointers array ***p*** from a mesh, reorder ***p*** using the permutation ***q*** returned by **partition_mesh**, and then supply these reordered pointers to the setup routine for the partitioned gather or scatter operation (described in Sections 14.10 and 14.11, respectively), the setup routine takes advantage of data locality; the communication required by the gather or scatter is reduced.

14.9.1 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 42 shows a two-dimensional unstructured mesh with 11 elements. Node numbers are not shown in this figure.

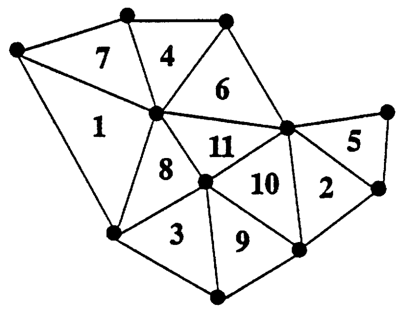

Figure 42. A two-dimensional unstructured mesh with 11 elements.

Two ways of representing an unstructured mesh are as follows:

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

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

  This array has dimensions ***ien***(*nnode*, *nel*), where *nnode* is the maximum number of nodes per element and *nel* is the number of elements in the mesh. In Figure 43, ***ien***(1, 8) = ***ien***(2, 1) = 1.

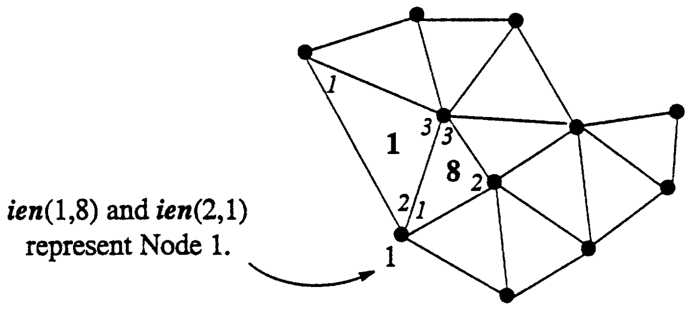

Figure 43. The ***ien*** array identifies nodes.

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

  idual(*m*, *n*) = the element that shares face *m* with element *n*;
  idual(*m*, *n*) = 0 if face *m* of element *n* is a boundary of the mesh.

This array has dimensions *idual*(*nface*, *nel*), where *nface* is the maximum number of faces per element and *nel* is the number of elements in the mesh. In Figure 44, ***idual***(1, 1) = 7, ***idual***(2, 1) = 8, and ***idual***(3, 1) = 0.

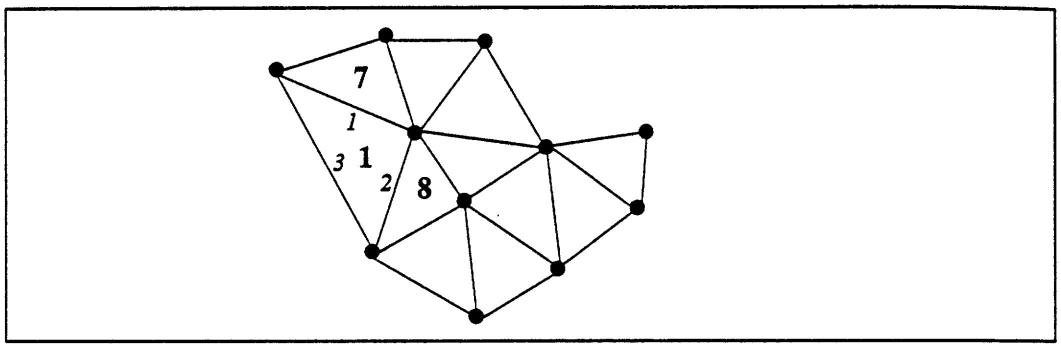

Figure 44. The ***idual*** array identifies neighboring elements.

14.9.2 Finite Element Numbering Scheme

The **generate_dual** routine generates the ***idual*** array corresponding to the ***ien*** array and element type you supply. Your ***ien*** array must use the standard finite element numbering scheme. Figure 45 shows the standard numbering schemes for the element types supported by **generate_dual**.

The **generate_dual** routine can handle higher-order element meshes as long as the nodes at the vertices of the elements are the first ones listed in ***ien***. Note that the current implementation of **generate_dual** does not support having different element types in the same mesh.

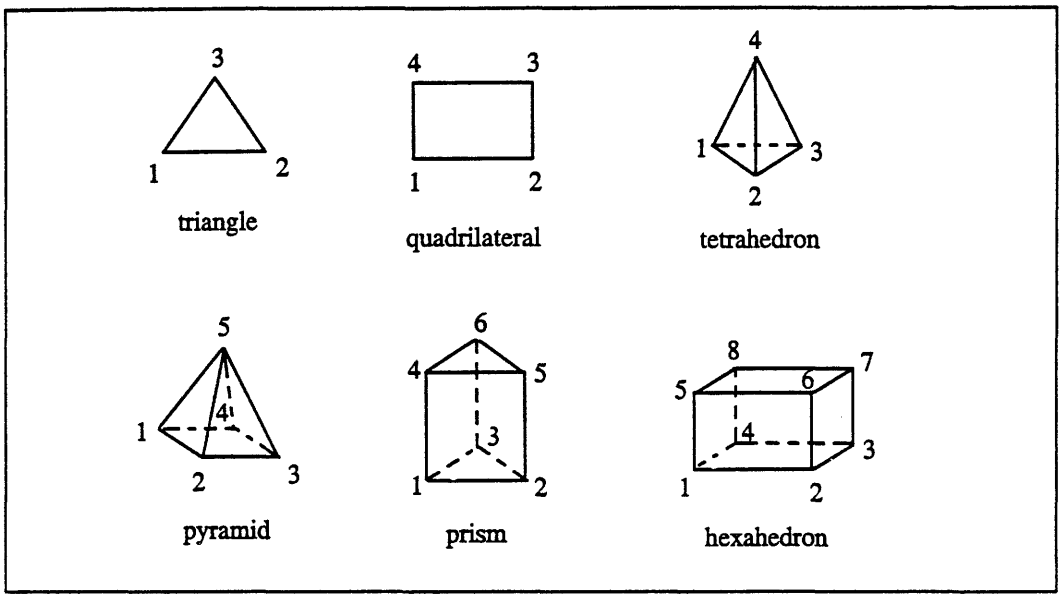

Figure 45. The standard finite element numbering schemes expected by **generate_dual**.

14.9.3 Partitioning Rules

You must supply the **partition_mesh** routine with an *idual* array that describes a mesh. The routine determines the number of resulting partitions based on the number of mesh elements and a number *N* that represents your machine configuration. The number *N* is defined by

N = *subgrid_quantum* * **CMF_number_of_processors( )**

where *subgrid_quantum* is

- 4 for a CM-2 or CM-200 (slicewise execution model).
- 1 for a CM-5 without vector units.
- 8 for a CM-5 with vector units.

Beginning with CM Fortran Version 2.1, *N* will equal **CMF_number_of_processors( )** (that is, *subgrid_quantum* will be 1) on CM-5 systems with vector units.

For the purposes of this discussion, when *N* is defined as above, the machine configuration in question is said to have *N processing entities*. Note that these "processing entities" are abstractions that represent a machine configuration.

Internally, **partition_mesh** maps them to the processing nodes in your current machine configuration.

The **partition_mesh** routine determines the number of partitions by associating mesh elements with processing entities, using the following principles:

- The mesh elements should be spread across as many processing entities as possible.
- All partitions except the last must hold the same number, k, of mesh elements; the last partition must hold $l \leq k$ elements.
- Each processing entity can be associated with at most one partition.

For example, if your current machine configuration has 128 processing entities, and your mesh contains 201 elements, **partition_mesh** associates two mesh elements with each of 100 processing entities, and the one remaining mesh element with the 101st processing entity. Thus, 101 partitions are required.

Upon completion, **partition_mesh** returns the number of processing entities in *numproc*, the number of partitions in the output parameter *numpar*, and the number of elements per partition (k, in the description above) in the *nelpar* argument. In the above example, *numproc* is set to 128, *numpar* to 101, and *nelpar* to 2.

14.9.4 The Partitioning Permutation

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

14.9.5 Mesh Partitioning Example

Suppose you supply **partition_mesh** with an ***idual*** array that represents the 11-element mesh shown in Figure 42, and the current machine configuration has four processing entities. Based on the partitioning principles listed above, **partition_mesh** determines that four partitions are required; they will hold elements 1-3, 4-6, 7-9, and 10-11, respectively. The **partition_mesh** routine might return the following q array:

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

This permutation results in the reordered mesh shown in Figure 46. Heavy lines denote partition boundaries.

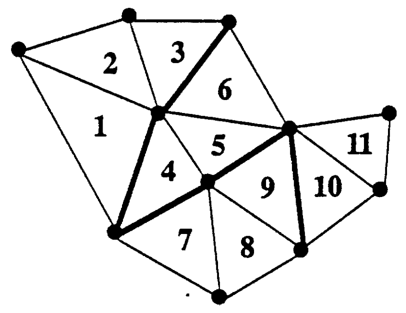

Figure 46. Partitions of a reordered mesh.

14.9.6 Reordering a Pointers Array

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

If p has rank r, **reorder_pointers** replaces element $(p_1, p_2, p_3, \ldots, p_r)$ with element $(p_1, p_2, p_3, \ldots, q(p_r))$. The reordered pointers array can then be supplied to the setup routines for the partitioned gather or scatter (see Sections 14.10 and 14.11).

Partitioning of an Unstructured Mesh and Reordering of Pointers

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

SYNTAX

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

partition_mesh (*idual, q, numproc, numpar, nelpar, ier*)

reorder_pointers (*p, q*)

ARGUMENTS

idual

Integer zero-based CM array 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. This *dual connectivity* array is defined by

idual(*m*, *n*) = the element that shares face *m* with element *n*;
idual(*m*, *n*) = 0 if face *m* of element *n* is a boundary of the mesh.

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

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

ien

Input integer CM array that describes an unstructured mesh. Must be at least one-based, with rank 2 and dimensions (*nnode*, *nel*), where *nnode* is the maximum number of nodes per element and *nel* is the number of elements in the mesh. This *element nodes* array is defined by

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

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

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

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

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

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

p Integer CM array of any rank and shape. On input to **reorder_pointers**, *p* contains pointers.

On successful completion of **reorder_pointers**, *p* contains the reordered pointers. If *p* has rank *r*, **reorder_pointers** replaces element $(p_1, p_2, p_3, ..., p_r)$ with element $(p_1, p_2, p_3, ..., q(p_r))$.

numproc Output scalar integer variable. On return, *numproc* is set to a value *N* that represents your current machine configuration and determines the number of partitions in the reordered mesh. (See the Description section for the definition of *N*).

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

nelpar	Output scalar integer variable. On return, contains the number of elements in each partition (with the possible exception of the last partition, which may contain fewer elements).
ier	Scalar integer variable. Error code. On return from **generate_dual**, *ier* has one of the following values:

0	Successful return.
-1	Invalid *elem_type*.
-2	Invalid dimensions for ***idual*** or ***ien***.
-3	***ien*** is not one-based.

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

0	Successful return.
-1	The second dimension of ***idual*** is not equal to the length of ***q***.

DESCRIPTION

The routines described here perform the following operations:

generate_dual	Given an *element nodes* array, ***ien***, that describes an unstructured mesh, this routine produces the corresponding *dual connectivity* array, ***idual***.
partition_mesh	Given a dual connectivity array describing an unstructured mesh, this routine returns a permutation, ***q***, that reorders the mesh elements to form discrete *partitions*. The routine also returns the number of resulting partitions and the number of elements per partition.
reorder_pointers	Given a pointers array, ***p***, and a permutation ***q*** (for example, the permutation returned by **partition_mesh**), this routine reorders the pointers array along its last axis using ***q***.

Partitioning Rules. The **partition_mesh** routine determines the number of partitions based on the number of mesh elements and a number *N* that represents your machine configuration. The number *N* is defined by

N = *subgrid_quantum* * **CMF_number_of_processors( )**

where *subgrid_quantum* is

- 4 for a CM-2 or CM-200 (slicewise execution model).
- 1 for a CM-5 without vector units.
- 8 for a CM-5 with vector units.

Beginning with CM Fortran Version 2.1, *N* will equal **CMF_number_of_ processors( )** (that is, *subgrid_quantum* will be 1) on CM-5 systems with vector units.

For the purposes of this discussion, when *N* is defined as above, the machine configuration in question is said to have *N processing entities*. Note that these "processing entities" are abstractions that represent a machine configuration. Internally, **partition_mesh** maps them to the processing nodes in your current machine configuration.

The **partition_mesh** routine determines the number of partitions by associating mesh elements with processing entities, using the following principles:

- The mesh elements should be spread across as many processing entities as possible.
- All partitions except the last must hold the same number, *k*, of mesh elements; the last partition must hold $l \leq k$ elements.
- Each processing entity can be associated with at most one partition.

For example, if your current machine configuration has 128 processing entities, and your mesh contains 201 elements, **partition_mesh** associates two mesh elements with each of 100 processing entities, and the one remaining mesh element with the 101st processing entity. Thus, 101 partitions are required.

Upon completion, **partition_mesh** returns the number of processing entities in *numproc*, the number of partitions in the output parameter *numpar*, and the number of elements per partition (*k*, in the description above) in the *nelpar* argument. In the above example, *numproc* is set to 128, *numpar* to 101, and *nelpar* to 2.

NOTES

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

EXAMPLES

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

```
partitioning/cmf
```

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

14.10 Partitioned Gather Utility

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

If your pointers are derived from a mesh, you can produce a partitioned pointers array using the **partition_mesh** and **reorder_pointers** routines described in Section 14.9.

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

Partitioned Gather Utility

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

SYNTAX

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

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

part_vector_gather (*y*, *x*, ***y_mask***, *trace*)

deallocate_part_gather_setup (*trace*)

ARGUMENTS

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

- *p* must have the same rank and axis extents as the array *y* you supply in any subsequent associated call to **part_ gather.**
- *p* must have the same rank and axis extents as the subarray (formed by omitting the left-most axis) of the array *y* you supply in any subsequent associated call to **part_ vector_gather.**

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

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

- In a subsequent call to **part_vector_gather**, the vector $x(:,n)$ is gathered from the source array to locations $y(:, p_1, ..., p_k)$ in the destination array during the gather operation.

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

If ***p*** has been reordered along its last axis to achieve data locality, **part_gather_setup** takes advantage of this locality and communication time is reduced.

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

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

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

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

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

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

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

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

y	CM array of any type with arbitrary shape. Destination array to which elements or vectors from the source array are gathered. All axes must be declared **:serial** except the last axis, which must have canonical (**:news**) layout. In a call to **part_vector_gather**, you must declare *y* with an extra left-most, **:serial** axis with the same extent as the left-most axis of *x*. Upon return from **part_vector_gather**, the gathered vectors will lie along this axis of *y*. The values of active elements of *y* are overwritten.
x	CM array of rank 1 (in a call to **part_gather**) or 2 (in a call to **part_vector_gather**), with the same type as *y*. Source array from which elements or vectors are gathered. In a call to **part_vector_gather**, the vectors to be gathered must lie along the left-most axis of *x*. This axis must be declared **:serial**, and must have the same extent as the left-most axis of *y*. The contents of *x* remain unchanged by **part_gather** and **part_vector_gather**.
trace	Scalar integer variable. Internal variable. The initial value you supply to **part_gather_setup** is ignored. You must supply **part_gather**, **part_vector_gather**, and **deallocate_part_gather_setup** with the value that **part_gather_setup** assigns to *trace*.
ier	Scalar integer variable. Upon return from **part_gather_setup**, contains one of the following codes:

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

DESCRIPTION

Definition. The gather operation is defined by

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

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

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

1. Call **part_gather_setup**.

2. Call **part_gather** or **part_vector_gather.**

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

3. After all **part_gather** and **part_vector_gather** calls associated with the same **part_gather_setup** call, call **deallocate_part_gather_setup** to deallocate the processing node storage space required by the setup routine.

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

Setup Phase. The **part_gather_setup** routine analyzes the gathering pattern supplied by the application in the *p* argument. Using *p* and *y_mask*, **part_gather_setup** computes an optimization, or *trace*, for the communication required by the gather operation; allocates the required storage space; and saves the trace for use in subsequent calls to the **part_gather** or **part_vector_gather** routine. The setup routine assigns appropriate value to the internal variable *trace*, which must be supplied in subsequent calls to **part_gather, part_vector_gather,** and **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 **part_gather** and **part_vector_gather** routines gather elements or vectors, respectively, from *x* into *y*, using the communication pattern saved by a previous call to **part_gather_setup.**

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

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

NOTES

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

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

EXAMPLES

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

```
partitioning/cmf/
```

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

14.11 Partitioned Scatter Utility

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

If your pointers are derived from a mesh, you can produce a partitioned pointers array using the **partition_mesh** and **reorder_pointers** routines described in Section 14.9.

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

Partitioned Scatter Utility

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

SYNTAX

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

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

part_vector_scatter (*y, x, x_mask, trace*)

deallocate_part_scatter_setup (*trace*)

ARGUMENTS

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

- *p* must have the same rank and axis extents as the array *x* you supply in any subsequent associated call to **part_ scatter.**
- *p* must have the same rank and axis extents as the subarray, formed by omitting the left-most axis, of the array *x* you supply in any subsequent associated call to **part_ vector_scatter.**

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

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

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

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

If ***p*** has been reordered along its last axis to achieve data locality, **part_scatter_setup** takes advantage of this locality and communication time is reduced.

x_mask

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

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

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

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

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

y_template

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

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

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

y	CM array of any type and rank 1 (in a call to **part_scatter**) or 2 (in a call to **part_vector_scatter**). The **part_scatter** routine adds the scattered elements from *x* to the initial values of *y*. The **part_vector_scatter** routine adds the scattered vectors from *x* to the initial values of the vectors that lie along the left-most axis of *y*. This axis must be declared **:serial**, and must have the same extent as the left-most axis of *x*.
x	CM array of arbitrary shape and of the same type and precision as *y*. Source array from which elements or vectors are scattered. All axes must be declared **:serial** except the last axis, which must have canonical (**:news**) layout. In a call to **part_vector_scatter**, you must declare *x* with an extra left-most, **:serial** axis with the same extent as the left-most axis of *y*. The vectors to be scattered must lie along this left-most axis of *x*. The contents of *x* remain unchanged by **part_scatter** and **part_vector_scatter**.
trace	Scalar integer variable. Internal variable. The initial value supplied to **part_scatter_setup** is ignored. You must supply **part_scatter**, **part_vector_scatter**, and **deallocate_part_scatter_setup** with the value assigned to *trace* by **part_scatter_setup**.

DESCRIPTION

Definition. The scatter operation is defined by

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

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

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

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

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

3. After all **part_scatter** and **part_vector_scatter** calls associated with the same **part_scatter_setup** call, call the **deallocate_part_scatter_setup** routine to deallocate the processing node storage space required by the setup routine.

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

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

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

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

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

EXAMPLES

Sample CM Fortran code that uses the partitioned gather and scatter utilities can be found on-line in the subdirectory `partitioning/cmf/` of a CMSSL examples directory whose location is site-specific.

14.12 Computation of Block Cyclic Permutations

The **compute_fe_block_cyclic_perms** routine computes the permutations required to transform any matrix from normal (elementwise consecutive) order to block cyclic order, and vice versa. Many CMSSL routines use **compute_fe_block_cyclic_perms** internally because they operate on matrices in block cyclic order to achieve improved performance.

NOTE

The **compute_fe_block_cyclic_perms** routine only *computes* block cyclic permutations; it does not *apply* any permutations to the supplied matrices. To apply the block cyclic permutations, use the **permute_cm_matrix_axis_from_fe** routine, described in Section 14.13.

The *LU* and *QR* routines described in Chapter 5 are examples of CMSSL routines that operate on matrices in block cyclic order. As a result, the *L* and *U* factors produced by the **gen_ lu_get_l** and **gen_lu_get_u** routines, and the *R* factor produced by the **gen_qr_get_r** routine, are in block cyclic order. The **compute_fe_block_cyclic_perms** routine is useful in conjunction with the *LU* and *QR* routines if you want to obtain the *L* and *U* factors or the *R* factor in elementwise consecutive order. An example of this use of **compute_fe_block_cyclic_perms** is given in Section 14.12.2.

The **compute_fe_block_cyclic_perms** routine supports multiple instances in the sense that you supply it with one or more matrices embedded in a CM array. The matrices must all have the same dimensions; therefore, they all have the same block cyclic permutations, and **compute_fe_block_cyclic_perms** computes only one set of permutations for all the instances. The "multiple-instance" format of the routine is provided for compatibility with the *QR* and *LU* operations, and with the **permute_cm_matrix_axis_from_fe** routine, which applies permutations to multiple matrices at a time.

14.12.1 Blocking, Load Balancing, and Block Cyclic Ordering

One way to understand block cyclic ordering is to consider the CMSSL routines that use blocking and load balancing (which is achieved through cyclic ordering) to achieve improved performance. Examples of such routines are

- The *LU* (Gaussian elimination) routines (described in Chapter 5).
- The *QR* routines (described in the Chapter 5).
- The routines that perform Householder reduction of Hermitian matrices to real symmetric tridiagonal form and corresponding basis transformation (described in Chapter 8).

Blocking

In the strategy called *blocking*, routines operate on and transfer blocks of data rather than single data elements. Each block resides in the memory associated with a parallel processing node. Blocking results in fewer vector-vector operations and more matrix-vector (level 2 BLAS) operations, which, when local to a node, can yield very high performance.

When you call a routine that uses blocking, you supply a *blocking factor* (usually denoted by *nblock* in the argument list) that determines the block size. For example, if you call the *LU* factorization routine, **gen_lu_factor**, and supply a block size of *b*, then in its Gaussian elimination process, **gen_lu_factor** eliminates *b* variables at a time by subtracting multiples of *b* equations from all later equations. For more information about blocking, see reference 6 in Section 14.16.

Load Balancing

In an elimination operation that does not use load balancing, the elimination runs through the columns of the matrix in order. As a result, the number of active nodes decreases as the factorization proceeds. For example, if A is an $(m \times n)$ matrix distributed across an array of nodes by assigning a contiguous p by q chunk of A to each node, then each node contains p consecutive rows and q consecutive columns of A. Thus, the first column of nodes contains the first q columns of A. After q elimination steps, all the columns of A in the first column of nodes will have been processed. Thus, after q steps, an entire column of nodes becomes inactive. Similarly, after p steps, the first row of nodes becomes inactive.

Suppose instead that the columns are processed in cyclic order (column 1 of the first column of nodes, then column 1 of the second column of nodes, and so on). Then as columns are eliminated, the active subgrid on each node shrinks, but no node becomes completely inactive until the last column is eliminated from some column of nodes. This strategy, called *cyclic ordering*, achieves *load balancing*: that is, it increases the number of nodes that are active at any one time. For more information about cyclic ordering, see reference 6 in Section 14.16.

Blocking and Load Balancing Combined: Block Cyclic Ordering

When blocking and load balancing are combined, a routine computes on a block of columns or rows as a unit. For example, rather than updating a matrix a single column at a time, a routine performs a block update of a number of columns. That is, the load balancing scheme processes blocks, rather than single columns, in cyclic order. In fact, some CMSSL routines eliminate columns *and* rows in block cyclic order.

For example, if you choose the block size to be *b*, the QR or LU factorization routine eliminates columns and rows in the following order:

- Columns 1 through *b* of the first column of node, along with rows 1 through *b* of the first row of nodes.
- Columns 1 through *b* of the second column of nodes, along with rows 1 through *b* of the second row of nodes; and so on.

After eliminating *b* columns from each column of nodes, the routine returns to columns *b* + 1 through 2*b* of node column 1, and so on.

The permutations computed by the **compute_fe_block_cyclic_perms** routine are those required to transform the rows and columns of a given matrix from normal (elementwise consecutive) order to block cyclic order, and vice versa. For example, the permutation that transforms the columns from elementwise consecutive order to block cyclic order (with a blocking factor of *b*) does the following:

- Moves the first *b* columns of the original matrix into columns 1 through *b* of the first column of nodes.
- Moves the next *b* columns of the original matrix into columns 1 through *b* of the second column of nodes; and so on until the first *b* columns of each column of nodes have been filled.

- Moves the next b columns of the original matrix are moved to columns $b + 1$ through $2b$ of node column 1.
- Continues this pattern until all columns of the original matrix have been moved.

This description is approximate; the exact details involve operations on quantities of data that are not exact multiples of b rows and columns.

Choosing the Blocking Factor

As mentioned above, the CMSSL routines that operate on matrices in block cyclic order require you to supply the blocking factor in the *nblock* argument. The optimum block size depends on the routine you are calling and the details of your application. For example, for the *QR* routines (described in Chapter 5), *nblock* = 4 is a good choice for typical applications; for very large matrices, *nblock* = 8 or even 16 may yield faster factorization. On the other hand, if you specify pivoting with the *QR* routines, the current implementation requires *nblock* = 1. Specific requirements for *nblock* are documented in the man pages for the various routines.

The blocking factor you supply when you call **compute_fe_block_cyclic_perms** depends on the context of the call. If you are using **compute_fe_block_cyclic_perms** to "undo" the block cyclic ordering of an *L*, *U*, or *R* factor, supply the same blocking factor you supplied in the original *LU* or *QR* factorization call.

14.12.2 Obtaining *L*, *U*, and *R* Factors in Elementwise Consecutive Order

Because the *LU* and *QR* routines operate in block cyclic order, the *L*, *U*, and *R* factors produced by the **gen_lu_get_l**, **gen_lu_get_u**, and **gen_qr_get_r** routines, resepctively, are stored in block cyclic order. The **compute_fe_block_cyclic_perms** routine is useful if you want to permute *L*, *U*, or *R* to "undo" the block cyclic ordering. To obtain a factor in elementwise consecutive order, you must follow these steps:

1. Before factoring the matrices embedded in a CM array *A*, compute their block cyclic permutations by supplying *A* to **compute_fe_block_cyclic_perms.**

2. Use **permute_cm_matrix_axis_from_fe** to permute the columns and rows of the matrices embedded in A to block cyclic order.

3. Use **gen_lu_factor** or **gen_qr_factor** to factor the matrices in A. (The factorization routine works in block cyclic order, but since you already permuted the matrices to block cyclic order in Step 1, the factorization actually occurs in elementwise consecutive order.)

4. Use **permute_cm_matrix_axis_from_fe** to permute the columns and rows of the factors from block cyclic order back to elementwise consecutive order.

This strategy is illustrated in the on-line sample code. (See the man page at the end of this section for the pathname.)

Computation of Block Cyclic Permutations

Given a block size and a CM array containing one or more embedded matrices, the **compute_fe_block_cyclic_perms** routine computes the permutations required to transform the rows and columns of the matrices from elementwise consecutive order to block cyclic order, and vice versa.

SYNTAX

compute_fe_block_cyclic_perms (*A, n1, n2, row_axis, col_axis, nblock,*
f, f_inv, g, g_inv, ier)

ARGUMENTS

A	CM array of any data type and rank greater than or equal to 2, containing one or more instances of a matrix for which you want to compute the block cyclic permutations. Only the rank, axis extents, and layout directives of this array are used; its contents are ignored.
n1	Scalar integer variable. The number of rows in each matrix embedded in *A*. Must be $\geq$ *n2*.
n2	Scalar integer variable. The number of columns in each matrix embedded in *A*. Must be $\leq$ *n1*.
row_axis	Scalar integer variable. Identifies the axis that counts the rows of each matrix embedded in *A*. Axis *row_axis* must have length $\geq$ *n1*.
col_axis	Scalar integer variable. Identifies the axis that counts the columns of each matrix embedded in *A*. Axis *col_axis* must have length $\geq$ *n2*.
nblock	Scalar integer variable. The blocking factor upon which the block cyclic permutations are to be based.
f	Front-end integer array of rank 1 and length $\geq$ *n1*. Upon return, the first *n1* elements contain the permutation that

transforms the rows of each matrix from elementwise consecutive order to block cyclic order. Note: this permutation has no effect on the element ordering within a given row.

f_inv	Front-end integer array of rank 1 and length $\geq n1$. Upon return, the first $n1$ elements contain the permutation that transforms the rows of each matrix from block cyclic order to elementwise consecutive order. Note: this permutation has no effect on the element ordering within a given row.
g	Front-end integer array of rank 1 and length $\geq n2$. Upon return, the first $n2$ elements contain the permutation that transforms the columns of each matrix from elementwise consecutive order to block cyclic order. Note: this permutation has no effect on the element ordering within a given column.
g_inv	Front-end integer array of rank 1 and length $\geq n2$. Upon return, the first $n2$ elements contain the permutation that transforms the columns of each matrix from block cyclic order to elementwise consecutive order. Note: this permutation has no effect on the element ordering within a given column.
ier	Scalar integer variable. Set to 0 upon successful completion. The **compute_fe_block_cyclic_perms** routine can fail under any of the following conditions:

-1	The A you supplied is not a CM array.
-2	The rank of A is < 2.
-3	*row_axis* has length < $n1$.
-4	*col_axis* has length < $n2$.
-20	One of the permutation arrays has length ≤ 0.
-30	*row_axis* or *col_axis* is ≤ 0 or > rank(A), or *row_axis* = *col_axis*.

DESCRIPTION

Given a block size and a CM array *A* containing one or more embedded matrices, the **compute_fe_block_cyclic_perms** routine computes the permutations required to transform the rows and columns of the matrices from elementwise consecutive order to block cyclic order, and from block cyclic order to elementwise consecutive order.

The permutations *f* and *f_inv* are called *row permutations*. Each row moves as a whole unit; the elements within each column are reordered. The arrays *f* and *f_inv* must have length at least *n1*.

The permutations *g* and *g_inv* are called *column permutations*. Each column moves as a whole unit; the elements within each row are reordered. The arrays *fe_col_axis_perm* and *g_inv* must have length at least *n2*.

The values in the front-end arrays are the *destinations* of the corresponding row or column elements. For example, the row permutation *into* block cyclic order is given by the mapping

$$A(\,i, :) \longrightarrow A(\,f(i), :)$$

while the column permutation *out of* block cyclic order is given by the mapping

$$A(\,:, i) \longrightarrow A(\,:, g\_inv(i))$$

where *A* is a matrix embedded in *A*. For a detailed description of block cyclic order, refer to Section 14.12.1.

EXAMPLES

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

```
lu/cmf/
```

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

14.13 Permutation Along an Axis

The **permute_cm_matrix_axis_from_fe** routine permutes the rows or columns of one or more matrices, using a permutation that is supplied in a front-end array.

The man page on the next page describes this routine and its arguments in detail, and defines row and column permutations. Figure 47 illustrates a row permutation where

$$n\_p = 4, \quad n\_q = 6, \quad f = [2 \ 4 \ 1 \ 3]$$

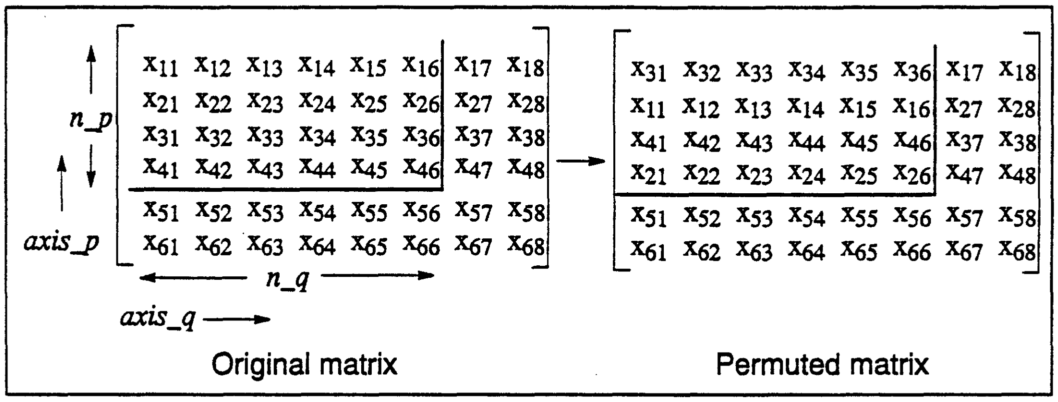

Figure 47. A row permutation.

Figure 48 illustrates a column permutation where

$$n\_p = 6, \quad n\_q = 4, \quad f = [3 \ 5 \ 2 \ 6 \ 1 \ 4]$$

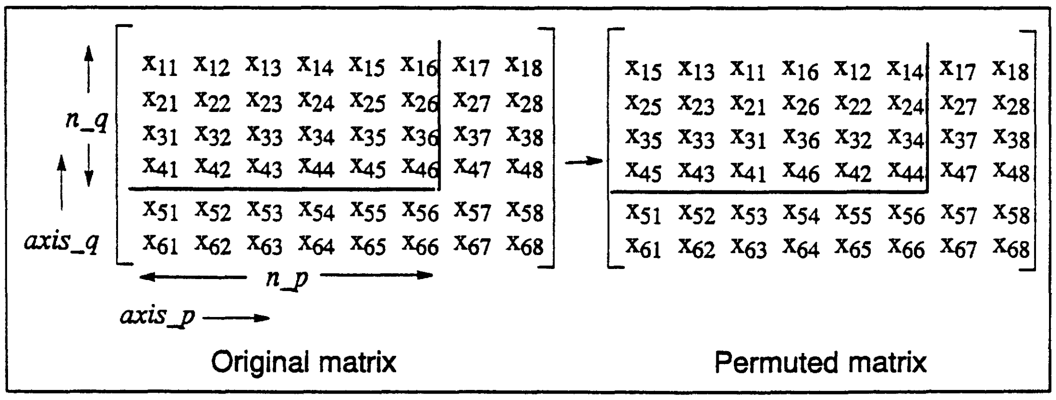

Figure 48. A column permutation.

Permutation Along an Axis

Given a CM array containing one or more embedded matrices, a front-end array containing a permutation, and a choice of row or column axis, the **permute_cm_matrix_axis_from_fe** routine permutes each matrix along the specified axis.

SYNTAX

permute_cm_matrix_axis_from_fe (*A*, *n_p*, *n_q*, *axis_p*, *axis_q*, *f*, *ier*)

ARGUMENTS

A	CM array of any data type and rank greater than or equal to 2, containing one or more embedded matrices whose rows or columns you want to permute.
n_p	Scalar integer variable. The active length of axis *axis_p*. Also the length of the permutation to be applied to axis *axis_p*.
n_q	Scalar integer variable. The active length of axis *axis_q*.
axis_p	Scalar integer variable. Identifies the axis along which elements are to be permuted. This axis counts either the rows or the columns of the matrices embedded in *A*.
axis_q	Scalar integer variable. Identifies the matrix axis other than *axis_p*.
f	Integer front-end array of rank 1 and length $\geq$ *n_p*. The first *n_p* elements of *f* must contain the new locations to which the first *n_p* elements along *axis_p* of each matrix are to be mapped.
ier	Scalar integer variable. Set to 0 upon successful completion. The **permute_cm_matrix_axis_from_fe** routine can fail under any of the following conditions:

-1	The *A* you have supplied is not a CM array.
-2	The rank of *A* is < 2.
-3	*axis_p* has length < *n_p*.

-4	*axis_q* has length < *n_q*.
-20	The *f* array has length ≤ 0.
-30	*axis_p* or *axis_q* is ≤ 0 or > rank(*A*), or *axis_p* = *axis_q*.

DESCRIPTION

Given a CM array *A* containing one or more embedded matrices, a front-end array *f* containing a permutation, and a choice of row or column axis (*axis_p*), the **permute_cm_matrix_axis_from_fe** routine permutes each matrix along *axis_p*.

If axis *axis_p* counts the rows, then the permutation is applied to the first *n_p* elements of each of the first *n_q* columns. This operation is called a *row permutation* because the first *n_q* elements of each row (up to row *n_p*, inclusive) move together as a whole unit.

If axis *axis_p* counts the columns, then the permutation is applied to the first *n_p* elements of each of the first *n_q* rows. This operation is called a *column permutation* because the first *n_q* elements of each column (up to column *n_p*, inclusive) move together as a whole unit.

The operation performed by this routine can be summarized as follows:

$A(,,,i,,,j,,,,) = A(,,,f(i),,,j,,,,,);$
$1 \leq i \leq n1$ along *axis_p*
$1 \leq j \leq n2$ along *axis_q*

A matrix element is not used in or affected by the permutation if its CM Fortran subscript in the *axis_p* dimension is greater than *n_p*, or if its CM Fortran subscript in the *axis_q* dimension is greater than *n_q*.

If *f* is the permutation *f* or *f_inv* computed by the **compute_fe_block_cyclic_perms** routine, then to transform the rows into or out of block cyclic order, you must set *axis_p*, *axis_q*, *n_p* and *n_q* to the same values as *row_axis*, *col_axis*, *n1*, and *n2*, respectively, in the **compute_fe_block_cyclic_perms** call.

Similarly, if *f* is the permutation *g* or *g_inv* computed by the **compute_fe_block_cyclic_perms** routine, then to transform the columns into or out of block cyclic order, you must set *axis_p*, *axis_q*, *n_p*, and *n_q* to the same values as *col_axis*, *row_axis*, *n2*, and *n1*, respectively, in the **compute_fe_block_cyclic_ perms** call.

EXAMPLES

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

```
lu/cmf/
```

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

14.14 Send-to-NEWS and NEWS-to-Send Reordering

On the CM-200, the **send_to_news** and **news_to_send** routines allow you to change the ordering of specified axes of a CM array from send to NEWS ordering or vice-versa.

On the CM-5, these routines have no effect, since NEWS and send ordering are the same. They are provided only for compatibility with the CM-200. (Refer to the CM Fortran documentation set for information about send and NEWS ordering.)

Send-to-NEWS and NEWS-to-Send Reordering

On the CM-200, given a CM array of any rank or type, the **send_to_news** and **news_to_send** routines change the ordering of specified axes of the array by shuffling the data in place. On the CM-5, these routines have no effect because send and NEWS ordering are the same. They are provided only for compatibility with the CM-200.

SYNTAX

send_to_news (*A*, *B*, *xform_vector*)
news_to_send (*A*, *B*, *xform_vector*)

ARGUMENTS

A	Integer front-end array of rank 1 and length **CMSSL_DESC_LENGTH** (a symbolic constant defined in the include file `cmssl-cmf.h`).
B	CM array of any rank and type. Each ***B*** axis to be reordered must be a power of 2 in length.
xform_vector	Logical front-end vector of length equal to the rank of ***B***, indicating which axes of ***B*** are to be reordered.

DESCRIPTION

On the CM-200, the **send_to_news** and **news_to_send** routines reorder the CM array ***B*** in place and, by creating a CM array descriptor in *A*, allow you to use the reordered source array in a subroutine.

On the CM-5, these routines have no effect because send and NEWS ordering are the same.

NOTES

Header File Required. Be sure to include the header file `cmssl-cmf.h` in any program that uses the **send_to_news** and **news_to_send** routines. This header file defines the symbolic constant **CMSSL_DESC_LENGTH.**

14.15 Communication Compiler

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

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

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

14.15.1 Communication Compiler Routines

The communication compiler includes the following routines:

- Setup routine: **comm_setup**

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

- Data motion and combining routines:

comm_get	**comm_send_max**
comm_send	**comm_send_min**
comm_send_add	**comm_send_or**
comm_send_and	**comm_send_xor**

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

- Compilation option routine: **comm_set_option**

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

- Deallocation routine: **deallocate_comm_setup**

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

NOTE

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

14.15.2 How to Use the Communication Compiler

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

- The type of operation must match the operation type you specified in the setup call. (The setup routine uses the same operation type for the **comm_send_add, comm_send_and, comm_send_max, comm_send_min, comm_send_or,** and **comm_ send_xor** operations, as these operations require the same setup action. A trace computed for any of these operations can be used for any of the others, assuming the other two conditions are met.)
- The source and destination arrays must have the same rank, axis extents, and layout directives as in the setup call.
- The source and destination arrays must have the same data type, and that data type must be valid for the operation to be performed.

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

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

1. Call **comm_setup** to allocate processing node memory and compute the trace.
2. Call the desired data motion or combining routine. (You can repeat this step an arbitrary number of times.)
3. Call **deallocate_comm_setup.**

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

NOTE

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

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

Communication Compiler

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

SYNTAX

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

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

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

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

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

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

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

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

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

comm_set_option (*option*, *value*, *ier*)

deallocate_comm_setup (*trace*)

ARGUMENTS

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

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

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

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

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

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

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

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

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

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

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

x

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

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

{x|y}_mask

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

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

operation

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

CMSSL_comm_by_get
CMSSL_comm_by_send

CMSSL_comm_by_send_overwrite
CMSSL_comm_by_send_with_op

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

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

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

CMSSL_method_automatic

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

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

    - If *operation* is **CMSSL_comm_by_get** and the maximum fan-out that any processing node will experience during the get is ≤ 2, **CMSSL_method_get_into_sends** is used. (For a definition of fan-out, see the description of **CMSSL_method_sort_and_scan**, below.)
    - If *operation* is **CMSSL_comm_by_get** and the maximum fan-out that any processing node will experience during the get is > 2, **CMSSL_method_sort_and_scan** is used.
    - If *operation* is **CMSSL_comm_by_send** or **CMSSL_comm_by_send_overwrite**, **CMSSL_method_nop** is used and collisions are stripped. (For details about collision stripping, see the Description section.)
    - If *operation* is **CMSSL_comm_by_send_with_op**, **CMSSL_method_serial_combining** is used.
- If *method*(2) > 999, **CMSSL_method_nop** is used.

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

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

CMSSL_method_fastgraph

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

CMSSL_method_get_into_sends

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

method:	May be either an integer front-end array of length ≥ 1 or a scalar integer.
Usage:	Valid only in **comm_setup** calls in which the operation is **CMSSL_comm_by_get**.

CMSSL_method_nop

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

method:	May be either an integer front-end array of length ≥ 1 or a scalar integer.
Usage:	Valid in any **comm_setup** call.

CMSSL_method_serial_combining

During trace creation, spreads identical send addresses along an extra, temporary serial axis. When the trace is used, messages are combined along the serial axis.

method:	May be either an integer front-end array of length ≥ 1 or a scalar integer.
Usage:	Valid only in **comm_setup** calls in which the operation is **CMSSL_comm_by_send_with_op.**

CMSSL_method_sort_and_scan

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

Fan-out applies only to get operations. The fan-out for a source location is the number of destination elements to which that location must send its data. A large variation in the fan-out means that some source locations are sending to few (or one) destination elements, while other source locations are sending to many destination elements.

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

method:	May be either an integer front-end array of length ≥ 1 or a scalar integer.
Usage:	Valid in **comm_setup** calls in which the operation is **CMSSL_comm_by_get** or **CMSSL_comm_by_send_with_op**.

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

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

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

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

-1	Invalid operation. The *operation* argument is not one of the allowed operations.
-2	Invalid method. The *method* argument is not allowed with the operation selected or is an unknown method.
-4	The scattering coordinates supplied in the *p* argument contain collisions and the *operation* is **CMSSL_comm_by_send**. This message is only a warning; the computed trace is valid.

–8	The *p* array has an illegal shape or layout, or contains invalid coordinates.

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

-1	Invalid data type. The *x* and/or *y* data type is invalid for this operation. For instance, complex data cannot be passed to **comm_send_max**. Only data types that are allowed in the combiner operation to be performed are allowed in the **comm_send_***combiner* routines.
-2	Invalid trace type. The *trace* value passed to this routine was of the wrong type for this operation. For instance, calling **comm_send_add** with a trace compiled with *operation* = **CMSSL_comm_by_get** will cause this error.
–4	Bad or unrecognized trace. The *trace* valuepassed to this routine was not usable. Possible reasons: the trace was never assigned the return value of a successful **comm_setup** call, the trace has already been deallocated, or the memory used by the trace has been corrupted.

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

-1	Invalid (unrecognized) *option* name.
-2	Invalid option value. The *value* supplied is not valid for the *option* selected.

DESCRIPTION

The communication compiler consists of the following routines:

comm_setup	Computes a message delivery optimization, or *trace*, for the specified operation using the source and destination shapes and layouts you have supplied. This trace can only be used on the current partition size. The setup routine allocates processing node and partition manager memory for storing the trace and related information. It returns an integer, *trace*,

that contains a pointer to the partition manager memory where information about the trace (including pointers to processsing node fields) is stored. You must supply the value of *trace* in subsequent calls to the data motion and combining routines, and also to **deallocate_comm_setup** (when you want to deallocate the memory associated with the trace).

comm_get

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

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

comm_send

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

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

comm_send_*combiner*

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

comm_send_add

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

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

comm_send_and

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

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

comm_send_max	where (*x_mask*) *y*(*p*|**max**) = *x* Selects the maximum value. Operates on integer and real data.
comm_send_min	where (*x_mask*) *y*(*p*|**min**) = *x* Selects the minimum value. Operates on integer and real data.
comm_send_or	where (*x_mask*) *y*(*p*|**.or.**) = *x* Performs a logical inclusive **OR** operation. Operates on logical or integer data. For integers, performs the operation on a bitwise basis.
comm_send_xor	where (*x_mask*) *y*(*p*|**.xor.**) = *x* Performs a logical exclusive **OR** operation. Operates on logical or integer data. For integers, performs the operation on a bitwise basis.

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

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

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

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

- The source and destination arrays must have the same rank, axis extents, and layout directives as in the setup call.
- The source and destination arrays must have the same data type, and that type must be valid for the operation to be performed.

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

To compute and use a trace, follow these steps:

1. Call **comm_setup** to allocate memory and compute the trace.
2. Call the desired data motion or combining routine. (You can repeat this step an arbitrary number of times.)
3. Call **deallocate_comm_setup.**

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

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

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

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

NOTES

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

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

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

The trace is essentially a pointer to a region of partition manager memory; this region of memory contains pointers to additional regions of memory allocated on both the partition manager and the processing nodes. If you destroy this pointer before deallocating the trace, there is no way for you to deallocate the trace subsequently. Therefore, be careful not to overwrite the trace pointer (for example, by allowing **comm_setup** to overwrite an allocated trace value, or by changing the value of the *trace* variable manually) unless you have deallocated the trace.

EXAMPLES

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

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

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

14.16 References

For more information on the all-to-all broadcast, see the following references:

1. Brunet, J.-Ph., and S. L. Johnsson. All-to-All Broadcast and Applications on the Connection Machine. *Int. J. Sup. App.* **6**, no. 3 (1992): 241-56.

2. Brunet, J.-Ph., J. P. Mesirov, and A. Edelman. An Optimal Hypercube Direct n-Body Solver on the Connection Machine. In *Supercomputing 90*, ICS Press, 1990. Pp. 748-52.

3. Johnsson, S. L., and C.-T. Ho. Spanning Graphs for Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Trans. Computers* **38**, no. 9 (1989): 1249-68.

4. Mathur, K. K. and S. L Johnsson. *All-to-All Communication on the Connection Machine CM-200*. Thinking Machines Corporation Technical Report TR-243, 1992.

5. Brunet, J.-Ph., A. Edelman, and J. P. Mesirov. Hypercube Algorithms for Direct *N*-Body Solvers for Different Granularities. To be published in *SIAM J. Sci. Stat. Comput.*

For information about block cyclic ordering, refer to

6. Lichtenstein, W. and S. L. Johnsson. *Block Cyclic Dense Linear Algebra*. Thinking Machines Corporation Technical Report TR-215, 1992.

For further information about the FastGraph trace compilation method of the communication compiler, see the following reference:

7. Dahl, E, D. Mapping and Compiled Communication on the Connection Machine System. *Proceedings of the Fifth Distributed Memory Computing Conference*, *IEEE*, DMCC 1990, Charleston, South Carolina, April 8-10, 1990. Pp. 756-66.

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

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

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

10. Johan, Z. Data parallel finite element techniques for large-scale computational fluid dynamics. Ph.D. Thesis, Stanford University, 1992. Also available as Thinking Machines Corporation Technical Report 244, 1992.

Index

For pages 1 through 349, see Volume I; for pages 351 through 546, see Volume II.