Chapter 1

Introduction to CMSSL for C*

This chapter contains general information about the C* interface to the Connection Machine Scientific Software Library (CMSSL). The following topics are included:

- about CMSSL
- contents of CMSSL for C*
- notes on terminology and conventions
- data types supported
- support for multiple instances
- numerical stability for the linear algebra routines
- numerical complexity

1.1 About CMSSL

CMSSL is a rapidly growing set of numerical routines that support computational applications while exploiting the massive parallelism of the Connection Machine system. CMSSL provides data parallel implementations of familiar numerical routines, offering new solutions for performance optimization, algorithm choice, and application design. The library can be linked with code written in C*.

This beta release of CMSSL for C* includes dense matrix operations, fast Fourier transforms, random number generators, and a set of communication functions including a matrix transpose, gather and scatter routines, and a data partitioning routine with optimized gather and scatter.

Copyright © 1993 Thinking Machines Corporation

1.2 Contents of CMSSL for C*

The C* interface to CMSSL consists of a set of library routines and a safety mechanism.

1.2.1 Library Routines

Listed below are the operations included in CMSSL for C*.

- **Dense Matrix Operations**
  - *Inner Product*

    The *multiple-instance* inner product routines compute one or more instances of an inner product of two vectors. Each *single-instance* inner product routine computes the global inner product over all axes of two source parallel variables. The inner product either overwrites the destination, is added to the destination, or is added to a second variable. For complex data, routines that take the conjugate of the first operand are provided.

  - *2-Norm*

    The *multiple-instance* 2-norm routine computes one or more instances of the 2-norm of a vector. The *single-instance* 2-norm routine computes the global 2-norm of a parallel variable.

  - *Outer Product*

    The outer product routines compute one or more instances of an outer product of two vectors. The result either overwrites the destination, is added to the destination, or is added to a second parallel variable. For complex data, routines that take the conjugate of the second operand vector are provided.

  - *Matrix Vector Multiplication*

    The matrix vector multiplication routines compute one or more matrix vector products. The result either overwrites the destination, is added to the destination, or is added to a second parallel variable. For complex data, routines that take the conjugate of the matrix are provided.

Copyright © 1993 Thinking Machines Corporation

- *Vector Matrix Multiplication*

  The vector matrix multiplication routines compute one or more vector matrix products. The result either overwrites the destination, is added to the destination, or is added to a second parallel variable. For complex data, routines that take the conjugate of the matrix are provided.

- *Infinity Norm*

  Computes the infinity norm(s) of one or more matrices.

- *Matrix Multiplication*

  The matrix multiplication routines compute one or more matrix products. The result either overwrites the destination, is added to the destination, or is added to a second variable. Routines that take the transpose of either or both operand matrices (or the Hermitian of either matrix, for complex data) are provided.

- **Fast Fourier Transforms (FFTs)**

  - *Simple Complex-to-Complex FFT*

    Performs a complex-to-complex Fast Fourier Transform in the same direction along all axes of a parallel variable.

  - *Detailed Complex-to-Complex FFT*

    Allows separate specification of the transform direction, scaling factor, and addressing mode along each data axis in a complex-to-complex FFT. Can improve performance over the Simple FFT in some cases. Supports multiple instances.

  - *Real-to-Complex and Complex-to-Real FFTs*

    The real-to-complex FFT computes the Fourier transform of real data; the complex-to-real FFT transforms conjugate symmetric sequences.

  - *Type Conversion Utilities*

    These utilities convert real parallel variables into complex parallel variables suitable for input to the real-to-complex FFT, and

Copyright © 1993 Thinking Machines Corporation

convert complex parallel variables (supplied in the format produced by the complex-to-real FFT) to real parallel variables.

- **Random Number Generators (RNGs)**
  - *Fast RNG*

    This lagged-Fibonacci RNG generates either real or integer pseudo-random numbers, and allows user-controlled reinitialization and checkpointing.

  - *VP RNG*

    This lagged-Fibonacci RNG produces identical streams on CM partitions of different sizes. It generates either real or integer pseudo-random numbers, and allows user-controlled reinitialization and checkpointing.

- **Communication Primitives**
  - *Matrix Transpose*

    Given a parallel variable of any type and two designated axes, this routine exchanges the two axes and returns the result in a second parallel variable.

  - *Sparse Gather Utility*

    These routines gather elements of a one-dimensional parallel variable into a parallel variable of any shape using a mapping supplied by the application. Pre-processing is performed by an associated setup routine.

  - *Sparse Scatter Utility*

    These routines scatter elements of a parallel variable of any shape to a one-dimensional parallel variable using a mapping supplied by the application. Pre-processing is performed by an associated setup routine.

  - *Sparse Vector Gather Utility*

    These routines perform the same operation as the sparse gather routines, except that the sparse vector gather operates on vectors rather than individual data elements.

Copyright © 1993 Thinking Machines Corporation

- *Sparse Vector Scatter Utility*

  These routines perform the same operation as the sparse scatter routines, except that the sparse vector scatter operates on vectors rather than individual data elements.

- *Block Gather and Scatter Utilities*

  These routines move a block of data from a source parallel variable into a destination parallel variable. The gather or scatter operation occurs along a single local axis. In the simplest case, a block of data elements is moved from a two-dimensional source (with one local dimension and one non-local dimension) to a similar destination. You can add instances by extending the non-local axis or by adding more axes (which may be local or non-local).

- *Partitioning of an Unstructured Mesh and Reordering of Mappings*

  These routines allow you to reorder a mapping derived from a mesh so that the communication required by subsequent partitioned gather and scatter operations is reduced. Four routines are provided:

  - Given *element nodes* values that describes an unstructured mesh, one routine produces the corresponding *dual connectivity* values.
  - Given dual connectivity values, a second routine returns a permutation that reorders the mesh elements to form discrete *partitions*.
  - Given a parallel variable containing a mapping and given a permutation, a third routine reorders the mapping along the last axis of the parallel variable using the permutation.
  - Given a parallel variable containing a mapping, a fourth routine changes the supplied mapping values for improved locality and returns the renumbering mapping.

  If you derive a mapping from a mesh, reorder it using the permutation returned by the partitioning routine, and then supply

Copyright © 1993 Thinking Machines Corporation

this reordered mapping to the setup routine for the partitioned gather or scatter operation, the setup routine takes advantage of data locality; the communication required by the gather or scatter is reduced.

- *Partitioned Gather Utility*

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

- *Partitioned Scatter Utility*

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

- *Vector Move (Extract and Deposit)*

  This routine moves a vector from a source parallel variable to a destination parallel variable of the same rank, type, and processing element layout. An associated utility routine returns processing element layout and subgrid shape information for any parallel variable.

Table 1 lists the CMSSL routines for C*, along with the chapters that describe them.

Copyright © 1993 Thinking Machines Corporation

Table 1. CMSSL Routines for C*.

Operation	Chapter	Routines
Inner product	3	**CMSSL_gen_inner_product** **CMSSL_gen_inner_product_noadd** **CMSSL_gen_inner_product_addto** **CMSSL_gen_inner_product_c1** **CMSSL_gen_inner_product_c1_noadd** **CMSSL_gen_inner_product_c1_addto** **CMSSL_gbl_gen_inner_product** **CMSSL_gbl_gen_inner_product_noadd** **CMSSL_gbl_gen_inner_product_addto** **CMSSL_gbl_gen_inner_product_c1** **CMSSL_gbl_gen_inner_product_c1_noadd** **CMSSL_gbl_gen_inner_product_c1_addto**
2-norm	3	**CMSSL_gen_2_norm** **CMSSL_gbl_gen_2_norm**
Outer product	3	**CMSSL_gen_outer_product** **CMSSL_gen_outer_product_noadd** **CMSSL_gen_outer_product_addto** **CMSSL_gen_outer_product_c2** **CMSSL_gen_outer_product_c2_noadd** **CMSSL_gen_outer_product_c2_addto**
Matrix vector multiplication	3	**CMSSL_gen_matrix_vector_mult** **CMSSL_gen_matrix_vector_mult_noadd** **CMSSL_gen_matrix_vector_mult_addto** **CMSSL_gen_matrix_vector_mult_c1** **CMSSL_gen_matrix_vector_mult_c1_noadd** **CMSSL_gen_matrix_vector_mult_c1_addto**

Copyright © 1993 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Vector matrix multiplication	3	**CMSSL_gen_vector_matrix_mult** **CMSSL_gen_vector_matrix_mult_noadd** **CMSSL_gen_vector_matrix_mult_addto** **CMSSL_gen_vector_matrix_mult_c2** **CMSSL_gen_vector_matrix_mult_c2_noadd** **CMSSL_gen_vector_matrix_mult_c2_addto**
Infinity norm	3	**CMSSL_gen_infinity_norm**
Matrix multiplication	3	**CMSSL_gen_matrix_mult** **CMSSL_gen_matrix_mult_noadd** **CMSSL_gen_matrix_mult_addto** **CMSSL_gen_matrix_mult_t1** **CMSSL_gen_matrix_mult_t1_noadd** **CMSSL_gen_matrix_mult_t1_addto** **CMSSL_gen_matrix_mult_h1** **CMSSL_gen_matrix_mult_h1_noadd** **CMSSL_gen_matrix_mult_h1_addto** **CMSSL_gen_matrix_mult_t2** **CMSSL_gen_matrix_mult_t2_noadd** **CMSSL_gen_matrix_mult_t2_addto** **CMSSL_gen_matrix_mult_h2** **CMSSL_gen_matrix_mult_h2_noadd** **CMSSL_gen_matrix_mult_h2_addto** **CMSSL_gen_matrix_mult_t1_t2** **CMSSL_gen_matrix_mult_t1_t2_noadd** **CMSSL_gen_matrix_mult_t1_t2_addto**

Copyright © 1993 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Complex-to-complex FFT	4	**CMSSL_fft_setup** **CMSSL_fft** **CMSSL_fft_detailed** **CMSSL_deallocate_fft_setup**
Real-to-complex and complex-to-real FFT	4	**CMSSL_fft_setup** **CMSSL_fft_detailed** **CMSSL_deallocate_fft_setup**
Type conversion utilities for the FFT	4	**CMSSL_real_from_complex** **CMSSL_complex_from_real**
Fast RNG	5	**CMSSL_initialize_fast_rng** **CMSSL_fast_rng** **CMSSL_fast_rng_state_field** **CMSSL_fast_rng_residue** **CMSSL_reinitialize_fast_rng** **CMSSL_deallocate_fast_rng**
VP RNG	5	**CMSSL_initialize_vp_rng** **CMSSL_vp_rng** **CMSSL_vp_rng_state_field** **CMSSL_vp_rng_residue** **CMSSL_reinitialize_vp_rng** **CMSSL_deallocate_vp_rng**

Copyright © 1993 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Matrix transpose	6	**CMSSL_gen_matrix_transpose**
Sparse gather	6	**CMSSL_sparse_util_gather_setup** **CMSSL_sparse_util_gather** **CMSSL_deallocate_gather_setup**
Sparse scatter	6	**CMSSL_sparse_util_scatter_setup** **CMSSL_sparse_util_scatter** **CMSSL_deallocate_scatter_setup**
Sparse vector gather	6	**CMSSL_sparse_util_vec_gather_setup** **CMSSL_sparse_util_vec_gather** **CMSSL_deallocate_vec_gather_setup**
Sparse vector scatter	6	**CMSSL_sparse_util_vec_scatter_setup** **CMSSL_sparse_util_vec_scatter** **CMSSL_deallocate_vec_scatter_setup**
Block gather and scatter utilities	6	**CMSSL_block_gather** **CMSSL_block_scatter**
Mesh partitioning, mapping reordering	6	**CMSSL_generate_dual** **CMSSL_partition_mesh** **CMSSL_reorder_mapping** **CMSSL_renumber_mapping**
Partitioned gather	6	**CMSSL_part_gather_setup** **CMSSL_part_gather** **CMSSL_part_vector_gather** **CMSSL_deallocate_part_gather_setup**
Partitioned scatter	6	**CMSSL_part_scatter_setup** **CMSSL_part_scatter** **CMSSL_part_vector_scatter** **CMSSL_deallocate_part_scatter_setup**
Vector move (extract and deposit)	6	**CMSSL_vector_move** **CMSSL_vector_move_utils**

Copyright © 1993 Thinking Machines Corporation

1.2.2 Safety Mechanism

The CMSSL safety mechanism offers two basic features: it synchronizes the CM-5 processing elements and partition manager so that you can pinpoint the area of code that generated an error, and it performs error checking and reports errors at several levels of detail. You can use the CMSSL safety mechanism either by setting an environment variable or by using library calls within a program. The safety mechanism is described in Chapter 2.

1.3 Notes on Terminology and Conventions

1.3.1 Parallel Variable Axis Descriptions

In descriptions of parallel variables throughout this manual, row and column axes are distinguished as follows:

- "The axis that counts the rows," "the row axis," and **`row_axis`** refer to *axis 0* in Figure 1.
- "The axis that counts the columns," "the column axis," and **`col_axis`** refer to *axis 1* in Figure 1.

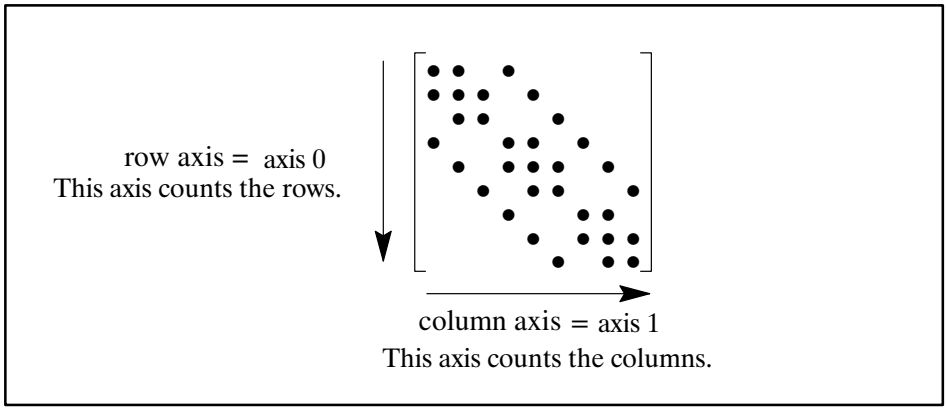

Figure 1. Row and column axes.

Copyright © 1993 Thinking Machines Corporation

1.3.2 Zero-Based Coordinates

The C* interface to CMSSL adheres to the C* convention of using 0-based axis numbering and coordinate numbering in scalar arrays and parallel variables. As a result, variables that take on coordinate values have 0-based values. For example, any variable containing a mapping in the gather, scatter, and partitioning routines, as well as variables containing mesh element numbers or node numbers, all take on 0 as a minimum value. This convention contrasts with that of CM Fortran, which uses 1-based axes and coordinates. Therefore, if you are accustomed to the CM Fortran interface to CMSSL, you must keep in mind that all coordinate values are one less in the C* interface than in the CM Fortran interface.

1.3.3 Processing Elements and Subgrids

Some sections of this manual contain implementation and performance information, and use the term *processing element*. A processing element is a vector unit (on machines with vector units) or a SPARC node (on machines without vector units, or if you compiled your code using the **`-sparc`** switch).

The mapping of parallel variable elements to processing elements is performed by the run-time system, and depends on the number of processing elements available to execute the program. You can control this mapping using the **`allocate_detailed_shape`** function, which is described in the *C* Programming Guide*.

The elements of a parallel variable residing within one processing element are said to be *local* to that processing element. The *subgrid* associated with a processing element consists of the data elements that are local to the processing element, as well as any "garbage elements" (padding) required in mapping positions of a shape to processing elements. The subgrids of a parallel variable are all the same size and are located at the same memory address within each processing element. The *subgrid length* of an axis is the number of parallel variable elements in the subgrid along that axis.

An axis is *local* if the data elements along the axis reside within one processing element. (In terms of the **`allocate_detailed_shape`** function, a local axis has a physical mask of 0.) An axis is *non-local* or *global* if the data elements along the axis span multiple processing elements.

Copyright © 1993 Thinking Machines Corporation

In most cases, you do not need to understand the implementation of a CMSSL routine at the level of processing elements in order to use the routine. Implementation and performance information is provided for users who want to tune and optimize their code.

1.3.4 Pointer Names Used for Variables

The syntax statements and argument descriptions in the man pages in this book indicate which parameters are pointers to parallel and scalar variables. To simplify the language of the descriptions, we often use the name of a pointer to refer to the variable it points to. For example, the matrix multiplication routine **CMSSL_gen_matrix_mult** requires three pointers to parallel variables (**A**, **B** and **C**). In describing the routine, we refer to **A**, **B** and **C** as parallel variables rather than pointers — for example, we discuss their shapes and the matrices embedded in them.

1.4 Data Types Supported

Table 2 shows the data types supported for each CMSSL operation. Within each subroutine call, all parallel variable arguments must match in data type, unless the argument descriptions indicate otherwise. In this table, *complex* refers to the data type **CMSSL_complex_t** and *dcomplex* refers to the data type **CMSSL_dcomplex_t**. These data types are defined in the file **<cm/cmssl_complex.h>**, which is included by **<cm/cmssl.h>**.

For a more complete discussion of C* and CMSSL data types, see Chapter 2.

Copyright © 1993 Thinking Machines Corporation

Table 2. Data Types Supported by CMSSL for C*.

Operation	Data Type				
	int	float	double	complex	dcomplex
Inner product		x	x	x	x
2-norm		x	x	x	x
Outer product		x	x	x	x
Matrix vector multiplication		x	x	x	x
Vector matrix multiplication		x	x	x	x
Infinity norm		x	x	x	x
Matrix multiplication		x	x	x	x
Simple complex-to-complex FFT				x	x
Detailed complex-to-complex FFT				x	x
Real-to-complex FFT				x	x
Complex-to-real FFT				x	x
Type conversion utilities		x	x	x	x
Fast RNG	x	x	x		
VP RNG	x	x	x		
Matrix transpose	x	x	x	x	x
Sparse gather utility	x	x	x	x	x
Sparse scatter utility	x	x	x	x	x
Sparse vector gather utility	x	x	x	x	x
Sparse vector scatter utility	x	x	x	x	x
Block gather and scatter utilities	x	x	x	x	x
Mesh partitioning, mapping reordering	x				
Partitioned gather utility	x	x	x	x	x
Partitioned scatter utility	x	x	x	x	x
Vector move	x	x	x	x	x

Copyright © 1993 Thinking Machines Corporation

1.5 Support for Multiple Instances

Many CMSSL routines support *multiple instances*: that is, they allow you to perform multiple independent operations on different data sets concurrently (see Section 1.5.1 for details). Table 3 shows which operations currently support multiple instances in C*.

Table 3. CMSSL Support for Multiple Instances in C*.

Operation	Instances	
	Single	Multiple
Inner product	x	x
2-norm	x	x
Outer product	x	x
Matrix vector multiplication	x	x
Vector matrix multiplication	x	x
Infinity norm	x	x
Matrix multiplication	x	x
Simple complex-to-complex FFT	x	
Detailed complex-to-complex FFT	x	x
Real-to-complex FFT	x	x
Complex-to-real FFT	x	x
Type conversion utilities	x	x
Fast RNG		x
VP RNG		x
Matrix transpose	x	x
Sparse gather utility	x	
Sparse scatter utility	x	
Sparse vector gather utility	x	
Sparse vector scatter utility	x	
Block gather and scatter utilities	x	x
Mesh partitioning, mapping reordering	x	
Partitioned gather utility	x	
Partitioned scatter utility	x	
Vector move	x	

Copyright © 1993 Thinking Machines Corporation

1.5.1 Defining Multiple Independent Data Sets

To perform a CMSSL operation on multiple independent data sets concurrently, you must embed the multiple independent instances of each operand or result argument in a parallel variable. The axes of the shape of the parallel variable fall into two mutually exclusive groups:

- The *data axes* define the geometry of the individual instances of the operand or result.
- The *instance axes* label the multiple instances.

Note that CMSSL routines typically work with parallel variables of differing shapes.

For example, Figure 2 illustrates a matrix vector multiplication operation in which four independent products are computed simultaneously. The four destination vectors are embedded in a two-dimensional parallel variable with one data axis (the vertical direction in the figure) and one instance axis; the four source vectors are similarly embedded in another parallel variable. The source matrices are embedded in a three-dimensional parallel variable. The instances within each variable are labeled 0 through 3.

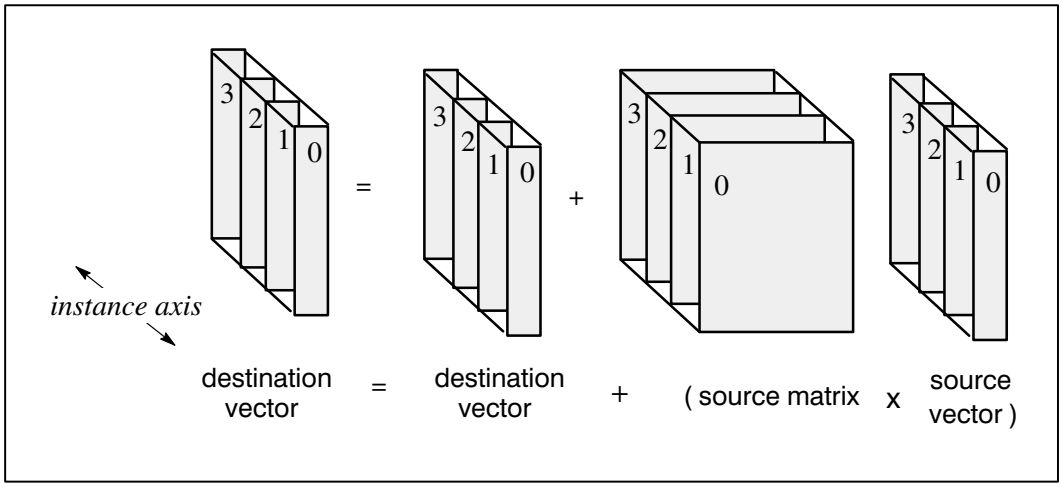

Figure 2. A multiple-instance matrix vector multiplication problem.

The logical unit on which the routine operates — sometimes called a *cell* — is defined by the data axes. The instance axes define the geometry of the

Copyright © 1993 Thinking Machines Corporation

frame in which the cells are embedded. The three-dimensional parallel variable shown in Figure 2 is a frame containing four two-dimensional cells.

The product of the lengths of the instance axes is the total number of instances. The product of the lengths of the data axes is the size of the cell.

1.5.2 Notation Used for Parallel Variables and Embedded Matrices

Throughout this book, parallel variable names (and all names of arguments to routines) are printed in **`bold typewriter`** font. If a parallel variable contains multiple instances of a matrix, the same name is often used for the parallel variable and each matrix instance it contains. The name is printed in **`bold typewriter`** to denote the parallel variable, and in ***bold italics*** to denote the embedded matrix. For example, the text might refer to "a parallel variable **`A`** containing one or more matrices ***A***."

1.5.3 Rules for Data Axes and Instance Axes

When you organize your data to form cells and frames for a multiple-instance operation, follow these rules:

- The shapes of all parallel variables containing operands and results must have the same number of instance axes.
- Counting up through the axes of the parallel variables, starting with axis 0 and excluding the data axes, corresponding instance axes must occur in the same order in each operand or result.
- The corresponding instance axes of each operand or result must have identical lengths. In some cases (indicated in the man pages for specific routines), corresponding instance axes must also have identical layouts. You may need to use **`allocate_detailed_shape`** (described in the *C* Programming Guide*) to satisfy this condition.
- The lengths of the data axes must be defined so that the operation makes sense. For example, in matrix multiplication, the data axis lengths of the operand and result matrices must obey the standard rules for axis lengths in matrix multiplication. Specific requirements for data axis lengths are provided in the descriptions of individual routines in later chapters.

Copyright © 1993 Thinking Machines Corporation

- Except where explicitly noted, CMSSL supports all combinations of layouts for data axes and instance axes. The layout that results in best performance depends on the operation. However, in most cases performance is best when the cells (that is, all of the data axes) are local to a processing element. Instance axes are typically defined as non-local axes. Some of the descriptions of individual routines in this book contain specific information about optimizing layouts. To manipulate a shape's layout, use the **`allocate_detailed_shape`** function.

Most CMSSL routines impose few or no restrictions on where the instance axes can occur in a parallel variable.

1.5.4 Specifying Single-Instance vs. Multiple-Instance Operations

CMSSL routines that support multiple instances have the same calling sequence for single-instance and multiple-instance operations. The methods you must use to specify single-instance and multiple-instance operations depend on the type of routine you are calling. Specific information is provided in the man pages included in Chapters 3 through 6. Several examples are discussed below.

Example 1. Matrix Vector Multiplication

When you call the matrix vector multiplication routine, **CMSSL_gen_matrix_vector_mult**, the dimensionality of the arguments you supply determines whether the routine performs a single-instance or multiple-instance operation, as follows:

- To perform a single-instance operation, specify each vector argument as a one-dimensional parallel variable and each matrix argument as a two-dimensional parallel variable. (Alternatively, you declare these arguments to have more dimensions, but all instance axes must have length 1.)

- To perform a multiple-instance operation, embed the multiple instances of each vector argument in a parallel variable of rank greater than 1, and embed the multiple instances of each matrix argument in a parallel variable of rank greater than 2.

Copyright © 1993 Thinking Machines Corporation

This routine requires you to specify which axes you are using as data axes for each matrix or vector argument. Chapter 3 provides details.

Example 2: Fast Fourier Transforms

When you call the Detailed complex-to-complex FFT (CCFFT) routine, you can supply a multidimensional parallel variable and specify whether you want to perform a forward transform, an inverse transform, or no transform along each axis. You can also specify axes along which no transform is performed but address bits are reversed. The axes that are transformed or bit-reversed are the data axes, and define the cell; the axes along which you perform no transformation are the instance axes.

The Simple CCFFT performs a transform along each axis of the supplied parallel variable, and therefore does not support multiple instances.

In addition to the CCFFT, the CMSSL provides a real-to-complex FFT (RCFFT) for computing the Fourier transform of real data, and a complex-to-real FFT (CRFFT) for the transformation of *conjugate symmetric* complex sequences. The Fourier Transform of a real or conjugate symmetric sequence can be computed using half the storage and half the arithmetic of a CCFFT. The RCFFT and CRFFT support multiple instances in a manner similar to that of the CCFFT.

For detailed information about multidimensional and multiple-instance FFTs, refer to Chapter 4.

Example 3: Random Number Generators

The random number generators support multiple instances in the sense that they produce multiple streams of random numbers (one stream per processing element or one stream per parallel variable element). Chapter 5 provides details.

1.6 Numerical Stability for the Linear Algebra Routines

Some of the descriptions of linear algebra routines in later chapters include information about *numerical stability*. In this book, numerical stability is defined in the standard way: an algorithm is stable if the computed result is the

Copyright © 1993 Thinking Machines Corporation

exact solution of a slightly different problem. For example, if A is the input matrix, the computed result is the true result corresponding to the matrix $A + E$, where E is small in norm compared with A.* Most of the algorithms used by CMSSL are numerically stable in this sense. However, a few are only *conditionally stable*, which means that the numerical stability may depend on the condition number of the problem. For information about the stability of specific routines, refer to the descriptions of the routines in later chapters.

1.7 Numerical Complexity

The following table lists the numbers of floating-point operations performed by some of the CMSSL routines. In this table, the term *complex operands* refers to types **`CMSSL_complex_t`** and **`CMSSL_dcomplex_t`**. These types are defined in the file **`<cm/cmssl_complex.h>`**, which is included by **`<cm/cmssl.h>`**.

Table 4. Number of Floating-Point Operations (flops) Performed by CMSSL Routines

Routine	# flops (real operands)	# flops (complex operands)
Vector length = q, number of instances = I:		
CMSSL_gen_inner_product	$2qI$	$8qI$
CMSSL_gen_inner_product_noadd	$(2q-1)I$	$(8q-2)I$
CMSSL_gen_inner_product_addto	$2qI$	$8qI$
CMSSL_gen_inner_product_c1	$2qI$	$8qI$
CMSSL_gen_inner_product_c1_noadd	$(2q-1)I$	$(8q-2)I$
CMSSL_gen_inner_product_c1_addto	$2qI$	$8qI$

* For a more formal definition, see Golub, G. H. and C. F. Van Loan, *Matrix Computations*, 2d ed. (Baltimore: Johns Hopkins University Press, 1989).

Copyright © 1993 Thinking Machines Corporation

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Product of axis lengths = Q:		
CMSSL_gbl_gen_inner_product	$2Q$	$8Q$
CMSSL_gbl_gen_inner_product_noadd	$2Q-1$	$8Q-2$
CMSSL_gbl_gen_inner_product_addto	$2Q$	$8Q$
CMSSL_gbl_gen_inner_product_c1	$2Q$	$8Q$
CMSSL_gbl_gen_inner_product_c1_noadd	$2Q-1$	$8Q-2$
CMSSL_gbl_gen_inner_product_c1_addto	$2Q$	$8Q$
Vector length = q, number of instances = I:		
CMSSL_gen_2_norm	$[(2q-1)+8]I$*	$(4q-1)+8]I$*
Product of axis lengths = Q:		
CMSSL_gbl_gen_2_norm	$(2Q-1)+8$*	$(4Q-1)+8$*
Matrix size = p x q, vector lengths = p and q, number of instances=I:		
CMSSL_gen_outer_product	$2pqI$	$8pqI$
CMSSL_gen_outer_product_noadd	pqI	$6pqI$
CMSSL_gen_outer_product_addto	$2pqI$	$8pqI$
CMSSL_gen_outer_product_c2	$2pqI$	$8pqI$
CMSSL_gen_outer_product_c2_noadd	pqI	$6pqI$
CMSSL_gen_outer_product_c2_addto	$2pqI$	$8pqI$
Matrix size = p x q, vector lengths = p and q, number of instances=I:		
CMSSL_gen_matrix_vector_mult	$2pqI$	$8pqI$
CMSSL_gen_matrix_vector_mult_noadd	$(2pq-p)I$	$(8pq-2p)I$
CMSSL_gen_matrix_vector_mult_addto	$2pqI$	$8pqI$
CMSSL_gen_matrix_vector_mult_c1	$2pqI$	$8pqI$
CMSSL_gen_matrix_vector_mult_c1_noadd	$(2pq-p)I$	$(8pq-2p)I$
CMSSL_gen_matrix_vector_mult_c1_addto	$2pqI$	$8pqI$

* The additional 8 flops are for the square root operation.

Copyright © 1993 Thinking Machines Corporation

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Matrix size = p x q, vector lengths = p and q, number of instances=I:		
CMSSL_gen_vector_matrix_mult	$2pqI$	$8pqI$
CMSSL_gen_vector_matrix_mult_noadd	$(2pq-p)I$	$(8pq-2p)I$
CMSSL_gen_vector_matrix_mult_addto	$2pqI$	$8pqI$
CMSSL_gen_vector_matrix_mult_c2	$2pqI$	$8pqI$
CMSSL_gen_vector_matrix_mult_c2_noadd	$(2pq-p)I$	$(8pq-2p)I$
CMSSL_gen_vector_matrix_mult_c2_addto	$2pqI$	$8pqI$
Matrix size = m x n, number of instances = I:		
CMSSL_gen_infinity_norm		
Matrix sizes = p x q and q x r, number of instances = I:		
CMSSL_gen_matrix_mult	$2pqrI$	$8pqrI$
CMSSL_gen_matrix_mult_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
CMSSL_gen_matrix_mult_addto	$2pqrI$	$8pqrI$
Matrix sizes = q x p and q x r, number of instances = I:		
CMSSL_gen_matrix_mult_t1	$2pqrI$	$8pqrI$
CMSSL_gen_matrix_mult_t1_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
CMSSL_gen_matrix_mult_t1_addto	$2pqrI$	$8pqrI$
CMSSL_gen_matrix_mult_h1	$2pqrI$	$8pqrI$
CMSSL_gen_matrix_mult_h1_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
CMSSL_gen_matrix_mult_h1_addto	$2pqrI$	$8pqrI$
Matrix sizes = p x q and r x q, number of instances = I:		
CMSSL_gen_matrix_mult_t2	$2pqrI$	$8pqrI$
CMSSL_gen_matrix_mult_t2_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
CMSSL_gen_matrix_mult_t2_addto	$2pqrI$	$8pqrI$
CMSSL_gen_matrix_mult_h2	$2pqrI$	$8pqrI$
CMSSL_gen_matrix_mult_h2_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
CMSSL_gen_matrix_mult_h2_addto	$2pqrI$	$8pqrI$

Copyright © 1993 Thinking Machines Corporation

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Matrix sizes = q x p and r x q, number of instances = I:		
CMSSL_gen_matrix_mult_t1_t2	$2pqrI$	$8pqrI$
CMSSL_gen_matrix_mult_t1_t2_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
CMSSL_gen_matrix_mult_t1_t2_addto	$2pqrI$	$8pqrI$
N = length of active axis; I = product of other axis lengths:		
CMSSL_fft		$5N\log N$
CMSSL_fft_detailed		$5IN\log N$

Copyright © 1993 Thinking Machines Corporation