CMSSL Version 3.2 for C*
Release Notes

Part I: Overview

1 Introduction

These release notes introduce the C* interface to Version 3.2 of the CM Scientific Software Library (CMSSL). One section of Part I is devoted to each of the following topics:

- hardware and software requirements
- porting information
- new features
- changes from the Beta release
- limitations and restrictions
- documentation for this release
- errors, feedback, and assistance

Part II provides detailed information about using CMSSL for C*.

CMSSL Version 3.2 for C* includes all the functionality of Version 3.2 Beta, as well as new functionality described in Section 4.

Copyright © 1994 Thinking Machines Corporation

2 Hardware and Software Requirements

2.1. Hardware Required

CMSSL Version 3.2 for C* supports CM-5 systems with or without vector units, and supports the nodal CMSSL library. The nodal execution model requires a CM-5 with vector units.

2.2 Software Required

CMSSL Version 3.2 for C* requires prior installation of the following software:

- CMOST Version 7.2 or higher
- C* Version 7.1.1 or higher
- CM Fortran Version 2.1 or higher

In addition, for C* programs that run in the single-node execution model, you must install CMMD Version 3.0 or higher.

For information about how to link a C* program with CMSSL, see Section 10.1.2 in Part II of these release notes.

3 Porting Information

If you wrote C* programs that call CMSSL routines and you linked the programs with CMSSL Version 3.2 Beta, recompile your programs with C* Version 7.1.1 and relink with CMSSL Version 3.2. See Section 10.1.2 in Part II of these release notes for information about how to do this.

If you have programs that call the 3.2 Beta versions of any of the mesh partitioning routines (**`CMSSL_generate_dual`**, **`CMSSL_partition_mesh`**, **`CMSSL_reorder_mapping`**, or **`CMSSL_renumber_mapping`**), you must change your code before recompiling and relinking with Version 3.2. The calling sequences of these routines have changed since the Beta release. See Section 5 for details.

Version 3.2, July 1994
Copyright © 1994 Thinking Machines Corporation

4 New Features

Version 3.2 is the first release of CMSSL for C*. The following routines were added to the C* interface to CMSSL after Version 3.2 Beta:

- external (out-of-core) matrix multiplication
- sparse matrix operations
- general linear solvers (in-core and external)
- banded linear solvers
- iterative solvers
- eigensystem analysis
- singular value decomposition
- ordinary differential equations using a Runge-Kutta method
- dense simplex (linear programming)
- histogram routines
- polyshift
- all-to-all rotation, reduction, and broadcast
- communication compiler
- auxiliary file I/O routines

For a list of all CMSSL routines for C*, see Section 9.2.1 in Part II of these release notes.

Copyright © 1994 Thinking Machines Corporation

5 Changes from the Beta Release

The following changes have occurred in the C* interface to CMSSL since Version 3.2 Beta:

- *Mesh partitioning routines*

  These routines have several new arguments:

  - The **CMSSL_generate_dual** routine has two new arguments, **face_axis** and **node_axis**. The **face_axis** argument identifies the axis of the **idual** array that counts the faces of mesh elements. The **node_axis** argument identifies the axis of the **ien** array that counts the nodes of mesh elements.

  - The **CMSSL_partition_mesh** routine now includes the **face_axis** argument.

  - The **CMSSL_reorder_mapping** routine now allows you to specify the axis along which the mapping **p** is to be reordered. (In Version 3.2 Beta, this routine always reordered a mapping along the last axis.) The new `axis` argument specifies the axis of **p** along which the reordering is to occur; the parallel variable **q** must be a permutation along axis `axis` of **p**.

  - The **CMSSL_renumber_mapping** routine now also includes an `axis` argument. This argument specifies the axis of the parallel variable **p** that counts the mesh elements represented by the mapping. If **p** has been reordered with **CMSSL_reorder_mapping**, then the value of `axis` in the **CMSSL_renumber_mapping** call must be the same as it was in the **CMSSL_reorder_mapping** call; that is, `axis` must be the axis along which **p** was reordered.

  In addition, you may now supply a null **mask** argument when you call **CMSSL_renumber_mapping**, if you do not need to mask any elements of **p**.

- *Gather and scatter routines*

  You may now supply null mask arguments in the following gather and scatter routines, if you do not need to mask parallel variable elements:

  CMSSL_sparse_util_gather_setup
  CMSSL_sparse_util_scatter_setup

Copyright © 1994 Thinking Machines Corporation

CMSSL_sparse_util_scatter
CMSSL_sparse_util_vec_gather_setup
CMSSL_sparse_util_vec_scatter_setup
CMSSL_sparse_util_vec_scatter
CMSSL_part_gather_setup
CMSSL_part_gather
CMSSL_part_vector_gather
CMSSL_part_scatter_setup
CMSSL_part_scatter
CMSSL_part_vector_scatter

- *Matrix transpose performance enhancements*

  Version 3.2 adds a performance enhancement for three-dimensional and higher-rank arrays. The **CMSSL_gen_matrix_transpose** routine yields superior performance when a local axis is exchanged with a non-local axis that is distributed across the vector units on the same processing node. By far the best performance is obtained when the non-local axis spans only two vector units (that is, only the lowest off-chip bit is set). The next best performance results when the non-local axis spans four vector units on the same processing node (the two lowest off-chip bits are set). For the general case of an axis that spans multiple processing nodes, the transpose performance does not depend significantly on the number of contiguous off-chip bits (except if the axis spans only two nodes, in which case the performance is slightly better). You can use this fact to improve the performance of transposes involving arrays of rank greater than or equal to three. With nodal CMSSL, you can also exploit this fact with two-dimensional arrays.

- *Fast Fourier Transform*

  The enhancements to **CMSSL_gen_matrix_transpose** mentioned above may help users who perform transposes explicitly when performing FFTs on multidimensional arrays.

Copyright © 1994 Thinking Machines Corporation

6 Limitations and Restrictions

The routines listed below cannot be used with the nodal CMSSL library (the library used when you compile your program in the C* single-node execution model).

`CMSSL_save_gen_lu`	`CMSSL_gen_lu_setup_ext`
`CMSSL_restore_gen_lu`	`CMSSL_gen_lu_factor_ext`
`CMSSL_save_gen_lu_fms`	`CMSSL_gen_lu_solve_ext`
`CMSSL_restore_gen_lu_fms`	`CMSSL_gen_qr_factor_ext`
`CMSSL_save_gen_qr`	`CMSSL_gen_qr_solve_ext`
`CMSSL_restore_gen_qr`	`CMSSL_save_fast_rng_temps`
`CMSSL_save_gen_qr_fms`	`CMSSL_restore_fast_rng_temps`
`CMSSL_restore_gen_qr_fms`	`CMSSL_save_vp_rng_temps`
`CMSSL_gen_matrix_mult_ext`	`CMSSL_restore_vp_rng_temps`

7 Documentation for This Release

The C* interface to CMSSL Version 3.2 will be documented in *CMSSL for C**, Version 3.2. The information summarized in these release notes will be presented in more detail in the manual.

Your software tape includes ASCII and PostScript versions of these release notes. The default location for the release notes is

```
/usr/doc/cmssl-3.2-cstar.releasenotes
```

If you do not find the release notes in the default location, check with your system administrator or the person who installs CMSSL at your site.

Version 3.2, July 1994
Copyright © 1994 Thinking Machines Corporation

NOTE

Until you receive *CMSSL for C**, Version 3.2, please refer to the header file **<cm/cmssl.h>** for routine prototypes.

7.1 On-Line Sample Code

Included with CMSSL are sample on-line programs that demonstrate how to call each CMSSL routine. You are encouraged to experiment with these sample programs.

The on-line sample programs are located in subdirectories of the CMSSL examples directory. The default location for the examples directory is

/usr/examples/cmssl

Examples for the operation *operation* are included in the subdirectory

operation/**cstar**

or

operation/*sub-operation*/**cstar**

of the examples directory. For example, the sample code for the sparse gather utility is located in the subdirectory

sparse-utilities/cstar

of the examples directory. If you do not find the on-line examples in **/usr/examples/cmssl**, check with your system administrator (or the person who installs CMSSL at your site) to find out where they were installed.

Copyright © 1994 Thinking Machines Corporation

8 Errors, Feedback, and Assistance

8.1. Bugupdate Files

For a list of fixed and outstanding bugs, please see the file **`cmssl-3.2.bugupdate`**. The default location of this file is the **`/usr/doc`** directory.

8.2 Request for Feedback

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.

If your site has an applications engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

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

Version 3.2, July 1994
Copyright © 1994 Thinking Machines Corporation

Part II: Detailed Information about New Features

9 Introduction to CMSSL for C*

This section contains introductory information about the C* interface to CMSSL. The following topics are included:

- about CMSSL
- contents of CMSSL for C*
- C* performance enhancement with CMSSL
- notes on terminology and conventions
- data types supported
- support for multiple instances

Section 10 describes how to use the C* interface to CMSSL.

9.1 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*.

CMSSL includes dense and sparse matrix operations; routines for solving dense, banded, and sparse linear systems; eigensystem analysis routines; singular value decomposition routines; fast Fourier transforms; routines for

Copyright © 1994 Thinking Machines Corporation

solving ordinary differential equations; a routine that solves minimization problems using the simplex linear programming method; random number generators; and histogramming routines. The library also provides a set of communication functions that offer a strong base for the development of computational tools. These functions support computations on problems represented by both structured and unstructured grids. Most CMSSL routines have been implemented to allow parallel computation on either multiple independent objects or a single large object. Over time, CMSSL will continue to grow into a complete set of standard scientific subroutines.

9.2 Contents of CMSSL for C*

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

9.2.1 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.

Version 3.2, July 1994
Copyright © 1994 Thinking Machines Corporation

- *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.

- *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.

- *Matrix Multiplication with External Storage*

  This routine performs the operation `Y = Y + AX` where `Y`, `A`, and `X` are matrices and `A` is too large to fit into core memory.

Copyright © 1994 Thinking Machines Corporation

- **Sparse Matrix Operations**
  - *Arbitrary Elementwise Sparse Matrix Operations*

    These routines compute the product of an arbitrary sparse matrix with a vector or dense matrix. The user application must store the non-zero elements of the sparse matrix in a packed vector. An associated setup routine provides options that may improve performance.

  - *Arbitrary Block Sparse Matrix Operations*

    These routines compute the product of a block sparse matrix with a vector or a dense matrix. Operand elements are gathered from the source vector or matrix, and product elements are scattered to the product vector or matrix, using a mapping provided by the application. An associated setup routine provides options that may improve performance.

  - *Grid Sparse Matrix Operations*

    These routines perform matrix vector, vector matrix, and matrix matrix multiplication in which the operand parallel variables are distributed across the points of a regular structured grid. These routines support multiple instances and block matrices.

- **General Linear System Solvers (In-Core)**
  - *Gaussian Elimination Routines*
    - *LU factorization routine*

      This routine uses Gaussian elimination (with or without partial pivoting) to perform the factorization `A=LU` for one or more instances of an *m* x *n* matrix `A`.

    - *LU solver routines*

      These routines use the `L` and `U` factors produced by the *LU* factorization routine to produce solutions to the systems `LUX=B` or $(LU)^T X = B$. `B` may represent one or more right-hand sides for each instance of the systems of equations.

Copyright © 1994 Thinking Machines Corporation

- *LU factor application routines*

  These routines use the factors produced by the *LU* factorization routine to solve triangular systems of equations. Included are routines for solving one or more instances of systems of equations of the form LX=B, L^TX=B, $L^{-1}X$ = B, $L^{-T}X$ = B, UX=B, and U^TX=B. B may represent one or more right-hand sides for each instance of the systems of equations.

- *LU utility routines*

  CMSSL also provides a set of utility routines associated with the LU factorization routine. For example, there are routines that explicitly compute L and U from the representation used internally in the factorization routine, save and restore internal *LU* information to or from a file, and estimate the infinity norm of each matrix A^{-1}.

▪ *Routines for Solving Linear Systems Using Householder Transformations*

- *QR factorization routine*

  This routine uses Householder transformations (with or without column pivoting) to perform the factorization A=QR for one or more instances of an m x n matrix A, where $m \geq n$. (When you specify pivoting, each matrix A is factored into three matrices: $A = QRP^{-1}$, where P is the permutation matrix that corresponds to the pivoting process.)

- *QR solver routines*

  These routines use the Q and R factors produced by the *QR* factorization routine to solve one or more instances of the systems of equations QRX=B or $(QR)^TX$=B. (With pivoting, these equations become $QRP^{-1}X$ = B and $(QRP^{-1})^TX$=B.) B may represent one or more right-hand sides for each instance of the systems of equations.

- *QR factor application routines*

  These routines use the factors produced by the *QR* factorization routine to solve triangular systems of equations (trapezoidal systems for Q). Included are routines for

Copyright © 1994 Thinking Machines Corporation

solving one or more instances of systems of equations of the form RX=B, R^TX=B, QX=B, and Q^TX=B. B may represent one or more right-hand sides for each instance of the systems of equations.

- *QR utility routines*

  CMSSL also provides a set of utility routines associated with the *QR* factorization routine. For example, there are routines that explicitly compute R from the representation used internally in the factorization routine, extract and deposit the diagonal of R, save and restore internal *QR* information to or from a file, apply the pivot permutation matrix to a supplied matrix or vector, and estimate the infinity norm.

- *Matrix Inversion*

  This routine inverts a square matrix using the Gauss-Jordan routine.

- *Gauss-Jordan System Solver*

  This routine solves (with partial or total pivoting) a system of equations of the form AX=B using a version of Gauss-Jordan elimination. B represents one or more right-hand sides.

- **General Linear System Solvers (External)**

  - *Gaussian Elimination with External Storage*

    - *External LU factorization routine*

      This routine uses block Gaussian elimination with full column pivoting to reduce an *n* x *n* matrix A to triangular form, where A is too large to fit into core memory.

    - *External LU solver routine*

      Given the factors computed by the external *LU* factorization routine, this routine solves AX = B for an arbitrary number of right-hand sides.

Version 3.2, July 1994
Copyright © 1994 Thinking Machines Corporation

- *QR Factorization and Least Squares Solution with External Storage*

  - *External QR factorization routine*

    This routine uses block Householder reflections to perform the factorization `A = QR`, where the matrix `A` is $m \times n$ (with $m \geq n$) and is too large to fit into core memory.

  - *External QR solver routine*

    Given the factors computed by the external *QR* factorization routine, this routine solves `AX = B` for an arbitrary number of right-hand sides.

- **Banded Linear System Solvers**

  - *Banded System Factorization and Solver Routines ("Unified")*

    These routines factor and solve tridiagonal, block tridiagonal, pentadiagonal, and block pentadiagonal systems. One routine performs the factorization. A second routine uses the resulting factors to solve one or more instances of systems of equations of the form `LUX = B`, where `L` and `U` are lower and upper (respectively) bidiagonal or block bidiagonal, or lower and upper (respectively) tridiagonal or block tridiagonal matrices, or permutations thereof. `B` represents one or more right-hand sides for each system of equations. You can choose from several algorithms: pipelined Gaussian elimination, pipelined Gaussian elimination with pairwise pivoting, substructuring with cyclic reduction, substructuring with balanced cyclic reduction, substructuring with pipelined Gaussian elimination, or substructuring with transpose.

  - *Banded System Factorization and Solver Routines*

    These routines perform the same operations as the banded solvers described above. For each type of system, the library provides separate factorization and solver routines as well as one routine that both factors and solves.

Copyright © 1994 Thinking Machines Corporation

- **Iterative Solvers**

  - *Krylov-Based Iterative Solvers*

    Given a matrix `A`, a right-hand-side vector `b`, and a preconditioner `M` = M_1*M_2, such that $\tilde{A}$ = $M_1^{-1}AM_2^{-1}$, these routines solve the system `Ax = b` using Krylov space iterative methods. Any matrix operations and preconditioning steps are provided by the user using a reverse communication interface. For most methods, both real and complex data types are supported.

- **Eigensystem Analysis of Real Symmetric Tridiagonal Systems**

  - *Reduction to Tridiagonal Form and Corresponding Basis Transformation*

    These routines reduce one or more real symmetric or complex Hermitian matrices to real symmetric tridiagonal form using Householder transformations. After this reduction occurs, for each instance, you can transform the coordinates of an arbitrary set of vectors from the basis of the original Hermitian matrix to that of the tridiagonal matrix, or vice versa.

  - *Eigenvalues of Real Symmetric Tridiagonal Matrices*

    This routine computes the eigenvalues of one or more real symmetric tridiagonal matrices using a parallel bisection algorithm.

  - *Eigenvectors of Real Symmetric Tridiagonal Matrices*

    This routine computes the eigenvectors corresponding to a given set of eigenvalues for one or more real symmetric tridiagonal matrices, using an inverse iteration algorithm.

- **Eigensystem Analysis of Dense Hermitian Systems**

  - *Eigensystem Analysis of Dense Hermitian Matrices*

    This routine computes the eigenvalues and eigenvectors of one or more real symmetric or complex Hermitian matrices.

  - *Generalized Eigensystem Analysis of Dense Hermitian Matrices*

    Given a parallel variable containing one or more dense Hermitian matrices `A`, and a parallel variable containing

Copyright © 1994 Thinking Machines Corporation

corresponding positive definite matrices B, this routine solves AQ = BQλ, computing the eigenvalues λ and, if desired, the eigenvectors for each instance.

- **Eigensystem Analysis of Dense Real Symmetric Systems**

  - *Eigensystem Analysis of Real Symmetric Matrices Using Jacobi Rotations*

    This routine computes the eigenvalues and eigenvectors of one or more real symmetric matrices using Jacobi rotations.

  - *Selected Eigenvalue and Eigenvector Analysis Using a* k-*Step Lanczos Method*

    This routine finds selected solutions {λ, x} to the real standard or generalized eigenvalue problem Lx = λBx. B can be positive semi-definite and is the identity for the standard eigenproblem. The operator L must be real and symmetric with respect to B; that is, BL=L^{-1}B. The algorithm used is a *k*-step Lanczos algorithm with implicit restart.

- **Eigensystem Analysis of Dense Real Systems**

  - *Selected Eigenvalue and Eigenvector Analysis Using a* k-*Step Arnoldi Method*

    This routine finds selected solutions {λ, x} to the real standard or generalized eigenvalue problem Lx = λBx. B is symmetric and can be positive semi-definite; it is the identity for the standard eigenproblem. The algorithm used is a *k*-step Arnoldi algorithm with implicit restart.

- **Eigensystem Analysis of Sparse Systems**

  The Lanczos and Arnoldi routines described above also apply to sparse systems.

Copyright © 1994 Thinking Machines Corporation

- **Singular Value Decomposition of Real Bidiagonal Matrices**

  - *Bidiagonalization and Corresponding Basis Transformation*

    Given one or more real dense matrices `A` of dimensions m x n with $m \geq n$, these routines find, for each `A`, a bidiagonal matrix `B` and matrices `U` and `V` such $A = UBV^T$. Auxiliary routines apply the transformations `U`, `V`, U^T, and V^T to arbitrary matrices or vectors, and allow you to transform an arbitrary set of vectors between the basis of the original dense matrix `A` and the basis of the corresponding bidiagonal matrix.

  - *Singular Values of Real Bidiagonal Matrices*

    This routine computes the singular values of one or more real bidiagonal matrices of the same order.

  - *Singular Vectors of Real Bidiagonal Matrices*

    This routine computes the singular vectors corresponding to a set of singular values of one or more real bidiagonal matrices of the same order.

- **Singular Value Decomposition of Dense Real Matrices**

  This routine computes the computes the singular values and, if desired, the singular vectors of one or more dense real matrices.

- **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.

Copyright © 1994 Thinking Machines Corporation

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

- **Ordinary Differential Equations**

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

    The initial value problem for a system of N coupled first-order ordinary differential equations (ODEs), $dy_i(x)/dx = f_i(x, y_1, \ldots, y_N)$ $i=1, \ldots, N$ 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. This routine solves the initial value problem by integrating explicitly the set of equations above 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.

- **Linear Programming**

  - *Dense Simplex*

    This 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 $\mathtt{c}^T\mathtt{x}$ subject to the conditions $\mathtt{Mx} = \mathtt{b}$, $0 \leq \mathtt{x} \leq \mathtt{u}$, where $\mathtt{M}$ is an $m \times n$ matrix, $\mathtt{c}$ is a coefficient vector, and $\mathtt{c}^T\mathtt{x}$ is referred to as the *cost*. The upper bound vector $\mathtt{u}$ may be infinity in one or more components.

Copyright © 1994 Thinking Machines Corporation

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

    This lagged-Fibonacci RNG generates either real or integer pseudo-random numbers, allows user-controlled reinitialization and checkpointing, and allows users to save and restore the RNG state table.

  - *VP RNG*

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

- **Statistical Analysis**
  - *Full Histogram*

    The full histogram records the distribution of values within a source parallel variable. Successive calls can provide an accumulation of totals.

  - *Range Histogram*

    The range histogram routines record the distribution of source parallel variable values within specified ranges. Successive calls can provide an accumulation of totals. Two routines are provided: one in which the histogram is stored in a scalar array, and one in which it is stored in a parallel variable.

- **Communication Primitives**
  - *Polyshift*

    This routine performs multidirectional and/or multidimensional shifts in a parallel variable.

  - *All-to-All Rotation*

    Given a parallel variable and a designated dimension, this routine performs an in-place, stepwise rotation of every value along the dimension to every location along the same dimension.

Version 3.2, July 1994
Copyright © 1994 Thinking Machines Corporation

- *All-to-All Broadcast*

  Given source and destination parallel variables of the same data type, with rank(destination) = rank(source) + 1, the all-to-all broadcast routine copies each instance of a source vector to the destination and replicates it along a selected dimension (the "broadcast dimension").

- *All-to-All Reduction*

  Given source and destination parallel variables with rank(source) = rank(destination) + 1, the all-to-all reduction routine combines sets of vectors within the source and places each result in a corresponding vector of the destination.

- *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 © 1994 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 any specified 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 this reordered mapping to the setup routine for the partitioned

Copyright © 1994 Thinking Machines Corporation

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.

- *Communication Compiler*

  A set of routines that compute and use message delivery optimizations for basic data motion and combining operations (get, send, send with overwrite, and send with combining). The communication compiler allows you to compute an optimization just once, and then use it many times in subsequent data motion and combining operations. This feature can yield significant time savings in applications that perform the same communication operation repeatedly. The communication compiler offers a variety of methods for computing an optimization.

- *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.

Copyright © 1994 Thinking Machines Corporation

- *Computation of Block Cyclic Permutations*

  This routine computes the permutations required to transform one or more matrices from normal (elementwise consecutive) order to block cyclic order, and vice versa.

- *Permutation along an Axis*

  This routine permutes the rows or columns of one or more matrices, using a permutation that is supplied in a scalar array.

- *Combination of Permutations*

  Given two source scalar arrays containing permutations, this routine generates a destination scalar array containing the combination of the permutations.

- *Zeroing Routine*

  This routine zeroes a parallel variable.

- **Auxiliary File I/O Routines**

  - *Reserving File Units*

    These routines reserve and release CM Fortran file units.

  - *Opening and Closing Files*

    These routines open and close files on parallel disk systems or on the partition manager.

  - *Rewinding Files*

    These routines rewind files on parallel disk systems or on the partition manager.

  - *Removing Files*

    These routines remove files from parallel disk systems or from the partition manager.

  - *Writing to Files*

    These routines write parallel variables to files in a variety of formats.

Copyright © 1994 Thinking Machines Corporation

- *Reading from Files*

  These routines read parallel variables from files in a variety of formats.

- *Reporting I/O Errors*

  This routine prints UNIX file system or CMFS I/O errors.

Table 1 lists the CMSSL routines for C*.

Copyright © 1994 Thinking Machines Corporation

Table 1. CMSSL Routines for C*.

Operation	Routines
Inner product	**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	**CMSSL_gen_2_norm** **CMSSL_gbl_gen_2_norm**
Outer product	**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	**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 © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Routines
Vector matrix multiplication	**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	**CMSSL_gen_infinity_norm**
Matrix multiplication	**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**
Matrix multiplication with external storage	**CMSSL_gen_matrix_mult_ext**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Routines
Arbitrary elementwise sparse matrix operations	**CMSSL_sparse_matvec_setup** **CMSSL_sparse_matvec_mult** **CMSSL_sparse_mat_gen_mat_mult** **CMSSL_deallocate_sparse_matvec_setup** **CMSSL_sparse_vecmat_setup** **CMSSL_sparse_vecmat_mult** **CMSSL_gen_mat_sparse_mat_mult** **CMSSL_deallocate_sparse_vecmat_setup**
Arbitrary block sparse matrix operations	**CMSSL_block_sparse_setup** **CMSSL_block_sparse_matrix_vector_mult** **CMSSL_vector_block_sparse_matrix_mult** **CMSSL_block_sparse_mat_gen_mat_mult** **CMSSL_gen_mat_block_sparse_mat_mult** **CMSSL_deallocate_block_sparse_setup**
Grid sparse matrix operations	**CMSSL_grid_sparse_setup** **CMSSL_grid_sparse_matrix_vector_mult** **CMSSL_vector_grid_sparse_matrix_mult** **CMSSL_grid_sparse_mat_gen_mat_mult** **CMSSL_gen_mat_grid_sparse_mat_mult** **CMSSL_deallocate_grid_sparse_setup**
Gaussian elimination	**CMSSL_gen_lu_factor** **CMSSL_save_gen_lu** **CMSSL_save_gen_lu_fms** **CMSSL_restore_gen_lu** **CMSSL_restore_gen_lu_fms** **CMSSL_gen_lu_solve** **CMSSL_gen_lu_solve_tra**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Routines
Gaussian elimination (*cont'd*)	**CMSSL_gen_lu_apply_l** **CMSSL_gen_lu_apply_l_tra** **CMSSL_gen_lu_apply_l_inv** **CMSSL_gen_lu_apply_l_inv_tra** **CMSSL_gen_lu_apply_u_inv** **CMSSL_gen_lu_apply_u_inv_tra** **CMSSL_gen_lu_get_l** **CMSSL_gen_lu_get_u** **CMSSL_gen_lu_zero_rows** **CMSSL_gen_lu_infinity_norm_inv** **CMSSL_deallocate_gen_lu**
Linear system solvers using Householder transformations	**CMSSL_gen_qr_factor** **CMSSL_save_gen_qr** **CMSSL_save_gen_qr_fms** **CMSSL_restore_gen_qr** **CMSSL_restore_gen_qr_fms** **CMSSL_gen_qr_solve** **CMSSL_gen_qr_solve_tra** **CMSSL_gen_qr_apply_q** **CMSSL_gen_qr_apply_q_tra** **CMSSL_gen_qr_apply_r_inv** **CMSSL_gen_qr_apply_r_inv_tra** **CMSSL_gen_qr_get_r** **CMSSL_gen_qr_apply_p** **CMSSL_gen_qr_apply_p_inv** **CMSSL_gen_qr_zero_rows** **CMSSL_gen_qr_extract_diag** **CMSSL_gen_qr_deposit_diag** **CMSSL_gen_qr_infinity_norm_inv** **CMSSL_gen_qr_r_infinity_norm_inv** **CMSSL_deallocate_gen_qr**
Matrix inversion	**CMSSL_gen_gj_invert**
Gauss-Jordan system solver	**CMSSL_gen_gj_solve**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Routines
Gaussian elimination with external storage	**CMSSL_gen_lu_setup_ext** **CMSSL_gen_lu_factor_ext** **CMSSL_gen_lu_solve_ext**
QR factorization and least squares solution with external storage	**CMSSL_gen_qr_factor_ext** **CMSSL_gen_qr_solve_ext**
Banded system factorization and solver routines (unified)	**CMSSL_gen_banded_factor** **CMSSL_gen_banded_solve** **CMSSL_deallocate_banded**
Banded system factorization and solver routines	**CMSSL_gen_tridiag_factor** **CMSSL_gen_tridiag_solve** **CMSSL_gen_tridiag_solve_factored** **CMSSL_gen_pentadiag_factor** **CMSSL_gen_pentadiag_solve** **CMSSL_gen_pentadiag_solve_factored** **CMSSL_block_tridiag_factor** **CMSSL_block_tridiag_solve** **CMSSL_block_tridiag_solve_factored** **CMSSL_block_pentadiag_factor** **CMSSL_block_pentadiag_solve** **CMSSL_block_pentadiag_solve_factored** **CMSSL_deallocate_banded_solve**
Krylov-based iterative solvers	**CMSSL_gen_iter_solve_setup** **CMSSL_gen_iter_solve** **CMSSL_deallocate_iter_solve**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Routines
Reduction to tridiagonal form; corresponding basis transformation	**CMSSL_sym_tred** **CMSSL_sym_to_tridiag** **CMSSL_tridiag_to_sym** **CMSSL_deallocate_sym_tred**
Eigenvalues of real symmetric tridiagonal matrices	**CMSSL_sym_tridiag_eigenvalues**
Eigenvectors of real symmetric tridiagonal matrices	**CMSSL_sym_tridiag_eigenvectors**
Eigensystem analysis of dense Hermitian matrices	**CMSSL_sym_tred_eigensystem**
Generalized eigensystem analysis of dense Hermitian matrices	**CMSSL_sym_tred_gen_eigensystem**
Eigensystem analysis using Jacobi rotations	**CMSSL_sym_jacobi_eigensystem**
Eigensystem analysis using a *k*-step Lanczos method	**CMSSL_sym_lanczos_setup** **CMSSL_sym_lanczos** **CMSSL_deallocate_sym_lanczos_setup**
Eigensystem analysis using a *k*-step Arnoldi method	**CMSSL_gen_arnoldi_setup** **CMSSL_gen_arnoldi** **CMSSL_deallocate_gen_arnoldi_setup**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Routines
Bidiagonalization and corresponding basis transformation	**CMSSL_gen_bidiag** **CMSSL_gen_bidiag_apply_u** **CMSSL_gen_bidiag_apply_u_tra** **CMSSL_bidiag_to_gen_left** **CMSSL_gen_to_bidiag_left** **CMSSL_gen_bidiag_apply_v** **CMSSL_gen_bidiag_apply_v_tra** **CMSSL_bidiag_to_gen_right** **CMSSL_gen_to_bidiag_right** **CMSSL_deallocate_gen_bidiag**
Singular values of bidiagonal matrices	**CMSSL_bidiag_svd_singular_values**
Singular vectors of bidiagonal matrices	**CMSSL_bidiag_svd_singular_vectors**
Singular value decomposition of dense real matrices	**CMSSL_gen_bidiag_singular_system**
Complex-to-complex FFT	**CMSSL_fft_setup** **CMSSL_fft** **CMSSL_fft_detailed** **CMSSL_deallocate_fft_setup**
Real-to-complex and complex-to-real FFT	**CMSSL_fft_setup** **CMSSL_fft_detailed** **CMSSL_deallocate_fft_setup**
Type conversion utilities for the FFT	**CMSSL_real_from_complex** **CMSSL_complex_from_real**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Routines
Explicit integration of ODEs (Runge-Kutta)	**CMSSL_ode_rkf_setup** **CMSSL_ode_rkf** **CMSSL_deallocate_ode_rkf_setup**
Dense simplex	**CMSSL_gen_simplex**
Fast RNG	**CMSSL_seed_fast_rng** **CMSSL_initialize_fast_rng** **CMSSL_fast_rng** **CMSSL_save_fast_rng_temps** **CMSSL_restore_fast_rng_temps** **CMSSL_fast_rng_state_field** **CMSSL_fast_rng_residue** **CMSSL_reinitialize_fast_rng** **CMSSL_deallocate_fast_rng**
VP RNG	**CMSSL_initialize_vp_rng** **CMSSL_vp_rng** **CMSSL_save_vp_rng_temps** **CMSSL_restore_vp_rng_temps** **CMSSL_vp_rng_state_field** **CMSSL_vp_rng_residue** **CMSSL_reinitialize_vp_rng** **CMSSL_deallocate_vp_rng**
Full histogram	**CMSSL_histogram**
Range histogram	**CMSSL_histogram_range** **CMSSL_histogram_range_cm**
Polyshift	**CMSSL_pshift_setup** **CMSSL_pshift_setup_looped** **CMSSL_pshift** **CMSSL_deallocate_pshift_setup**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Routines
All-to-all rotation	**CMSSL_all_to_all_setup** **CMSSL_all_to_all** **CMSSL_deallocate_all_to_all_setup**
All-to-all broadcast	**CMSSL_all_to_all_broadcast**
All-to-all reduction	**CMSSL_all_to_all_reduce**
Matrix transpose	**CMSSL_gen_matrix_transpose**
Sparse gather	**CMSSL_sparse_util_gather_setup** **CMSSL_sparse_util_gather** **CMSSL_deallocate_gather_setup**
Sparse scatter	**CMSSL_sparse_util_scatter_setup** **CMSSL_sparse_util_scatter** **CMSSL_deallocate_scatter_setup**
Sparse vector gather	**CMSSL_sparse_util_vec_gather_setup** **CMSSL_sparse_util_vec_gather** **CMSSL_deallocate_vec_gather_setup**
Sparse vector scatter	**CMSSL_sparse_util_vec_scatter_setup** **CMSSL_sparse_util_vec_scatter** **CMSSL_deallocate_vec_scatter_setup**
Block gather, block scatter	**CMSSL_block_gather** **CMSSL_block_scatter**
Mesh partitioning, mapping reordering	**CMSSL_generate_dual** **CMSSL_partition_mesh** **CMSSL_reorder_mapping** **CMSSL_renumber_mapping**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Routines
Partitioned gather	**CMSSL_part_gather_setup** **CMSSL_part_gather** **CMSSL_part_vector_gather** **CMSSL_deallocate_part_gather_setup**
Partitioned scatter	**CMSSL_part_scatter_setup** **CMSSL_part_scatter** **CMSSL_part_vector_scatter** **CMSSL_deallocate_part_scatter_setup**
Communication compiler	**CMSSL_comm_setup** **CMSSL_comm_get** **CMSSL_comm_send** **CMSSL_comm_send_overwrite** **CMSSL_comm_send_add** **CMSSL_comm_send_and** **CMSSL_comm_send_max** **CMSSL_comm_send_min** **CMSSL_comm_send_or** **CMSSL_comm_send_xor** **CMSSL_comm_set_option** **CMSSL_deallocate_comm_setup**
Vector move (extract/deposit)	**CMSSL_vector_move** **CMSSL_vector_move_utils**
Computation of block cyclic permutations	**CMSSL_compute_fe_block_cyclic_perms**
Permutation along an axis	**CMSSL_permute_cm_matrix_axis_from_fe**
Combination of permutations	**CMSSL_combine_fe_perms**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Routines
Zeroing	`CMSSL_zero_elements`

Reserving file units	`CMSSL_file_reserve_unit`
	`CMSSL_fe_file_reserve_unit`
	`CMSSL_file_release_unit`
	`CMSSL_fe_file_release_unit`

Opening and closing files	`CMSSL_file_open`
	`CMSSL_file_fdopen`
	`CMSSL_fe_file_open`
	`CMSSL_file_close`
	`CMSSL_fe_file_close`

Rewinding files	`CMSSL_file_rewind` `CMSSL_fe_file_rewind`
Removing files	`CMSSL_file_remove` `CMSSL_fe_file_remove`

Writing to files	`CMSSL_pvar_to_file`
	`CMSSL_pvar_to_file_fms`
	`CMSSL_pvar_to_file_so`

Reading from files	`CMSSL_pvar_from_file`
	`CMSSL_pvar_from_file_fms`
	`CMSSL_pvar_from_file_so`

Reporting I/O errors	`CMSSL_file_perror`

9.2.2 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

Version 3.2, July 1994
Copyright © 1994 Thinking Machines Corporation

mechanism either by setting an environment variable or by using library calls within a program. The safety mechanism is described in Section 10.5.

9.3 C* Performance Enhancement with CMSSL

The C* **`rank`** function yields better performance when the program is linked with CMSSL.

9.4 Notes on Terminology and Conventions

9.4.1 Row and Column Axes

In argument names in C* prototypes for CMSSL routines, row and column axes are distinguished as follows:

- `row_axis` refers to axis 0 in Figure 1.
- `col_axis` refer to axis 1 in Figure 1.

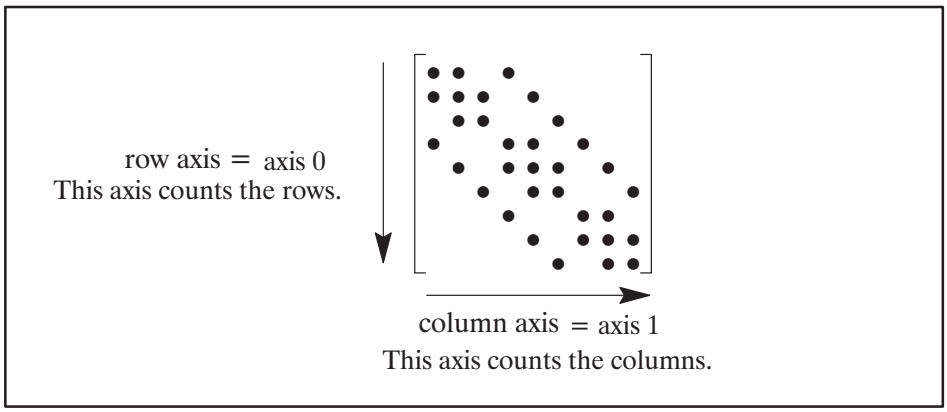

Figure 1. Row and column axes.

9.4.2 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 vari-

Copyright © 1994 Thinking Machines Corporation

ables. 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 coordinate values are one less in the C* interface than in the CM Fortran interface. (A few CMSSL routines for CM Fortran use 0-based coordinates; see Chapter 1 of *CMSSL for CM Fortran* for details.)

9.5 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. The data types `CMSSL_complex_t` and `CMSSL_dcomplex_t` 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 Section 10.2.1.

Copyright © 1994 Thinking Machines Corporation

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

Operation	Data Type				
	int	float	double	CMSSL_ complex_ t	CMSSL_ dcomplex_t
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
Matrix multiplication with external storage		x	x	x	x
Arbitrary elementwise sparse matrix operations		x	x	x	x
Arbitrary block sparse matrix operations		x	x	x	x
Grid sparse matrix operations		x	x	x	x
Gaussian elimination		x	x	x	x
Linear solvers using Householder transformations		x	x	x	x
Matrix inversion		x	x	x	x
Gauss-Jordan system solver		x	x	x	x
Gaussian elimination with external storage		x	x	x	x
QR factorization with external storage		x	x	x	x
Banded system solvers		x	x	x	x
Iterative solvers		x	x	x	x
Reduction to tridiagonal form		x	x	x	x
Corresponding basis transformation		x	x	x	x
Eigenvalues of real symmetric tridiagonal matrices		x	x		
Eigenvectors of real symmetric tridiagonal matrices		x	x		
Eigensystem analysis of dense Hermitian matrices		x	x	x	x
Generalized eigenanalysis of dense Herm. matrices		x	x	x	x
Eigensystem analysis using Jacobi rotations		x	x		
Selected eigenvalues/eigenvectors (Lanczos)		x	x		
Selected eigenvalues/eigenvectors (Arnoldi)		x	x		

Copyright © 1994 Thinking Machines Corporation

Table 2 *(Continued)*

Operation	Data Type				
	int	float	double	CMSSL_ complex_ t	CMSSL_ dcomplex_t
Bidiagonalization		x	x		
Corresponding basis transformation		x	x		
Singular values of bidiagonal matrices		x	x		
Singular vectors of bidiagonal matrices		x	x		
Singular value decomposition (dense real matrices)		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
ODEs (Runge-Kutta method)		x	x		
Dense simplex		x	x		
Fast RNG	x	x	x		
VP RNG	x	x	x		
Histogram	x	x	x		
Histogram range	x	x	x		
Polyshift	x	x	x	x	x
All-to-all rotation	x	x	x	x	x
All-to-all broadcast	x	x	x	x	x
All-to-all reduction	x	x	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, pointer reordering	x				
Partitioned gather utility	x	x	x	x	x
Partitioned scatter utility	x	x	x	x	x
Communication compiler	x	x	x	x	x

Copyright © 1994 Thinking Machines Corporation

Table 2 *(Continued)*

Operation	Data Type				
	int	float	double	CMSSL_ complex_ t	CMSSL_ dcomplex_t
Vector move	x	x	x	x	x
Computation of block cyclic permutations	x	x	x	x	x
Permutation along an axis	x	x	x	x	x
Combination of permutations	x	x	x	x	x
Zeroing routine	x	x	x	x	x

9.6 Support for Multiple Instances

Most CMSSL routines support *multiple instances*: that is, they allow you to perform multiple independent operations on different data sets concurrently (see Section 9.6.1 for details). Table 3 shows which operations currently support multiple instances in C*.

Copyright © 1994 Thinking Machines Corporation

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
Matrix multiplication with external storage	x	
Arbitrary elementwise sparse matrix operations	x	
Arbitrary block sparse matrix operations	x	
Grid sparse matrix operations	x	x
Gaussian elimination	x	x
Linear solvers using Householder transformations	x	x
Matrix inversion	x	
Gauss-Jordan system solver	x	
Gaussian elimination with external storage	x	
QR factorization with external storage	x	
Banded system solvers	x	x
Iterative solvers	x	
Reduction to tridiagonal form	x	x
Corresponding basis transformation	x	x
Eigenvalues of real symmetric tridiagonal matrices	x	x
Eigenvectors of real symmetric tridiagonal matrices	x	x
Eigenanalysis of dense Hermitian matrices	x	x
Generalized eigenanalysis of dense Hermitian matrices	x	x
Eigenanalysis using Jacobi rotations	x	x
Selected eigenvalues/eigenvectors (Lanczos)	x	
Selected eigenvalues/eigenvectors (Arnoldi)	x	

Copyright © 1994 Thinking Machines Corporation

Table 3 *(Continued)*

Operation	Instances	
	Single	Multiple
Bidiagonalization	x	x
Corresponding basis transformation	x	x
Singular values of bidiagonal matrices	x	x
Singular vectors of bidiagonal matrices	x	x
Singular value decomposition (dense real matrices)	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
Array conversion utilities	x	x
ODEs (Runge-Kutta method)	x	
Dense simplex	x	
Fast RNG		x
VP RNG		x
Histogram	x	
Histogram range	x	
Polyshift	x	x
All-to-all rotation	x	x
All-to-all broadcast	x	x
All-to-all reduction	x	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	
Communication compiler	x	x

Copyright © 1994 Thinking Machines Corporation

Table 3 *(Continued)*

Operation	Instances	
	Single	Multiple
Vector move	x	
Computation of block cyclic permutations	x	x
Permutation along an axis	x	x
Combination of permutations	x	
Zeroing routine	x	

9.6.1 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.

Copyright © 1994 Thinking Machines Corporation

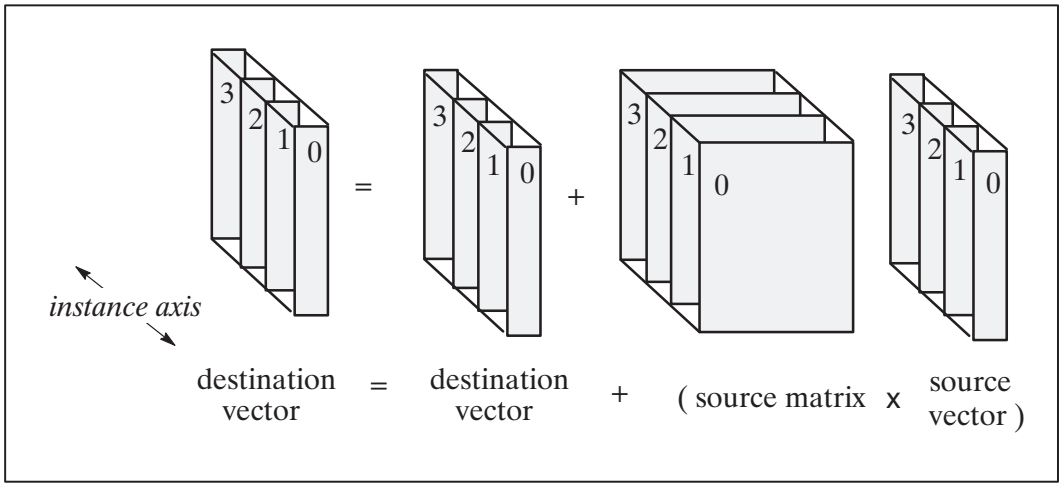

Figure 2. A multiple-instance matrix vector multiplication problem.

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 *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.

9.6.2 Notation Used for Parallel Variables and Embedded Matrices

In these release notes, parallel variable names are printed in **bold typewriter** font to distinguish them from other variable names. If a parallel variable contains multiple instances of a matrix or vector, we usually use the same name for the parallel variable and each matrix or vector instance within the parallel variable. The name is printed in **bold typewriter** font to denote the parallel variable, and in `typewriter` font to denote the embedded matrix or vector. For example, the text might refer to "a parallel variable **A** containing one or more matrices `A`."

Copyright © 1994 Thinking Machines Corporation

9.6.3 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 *CMSSL for C**.
- 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.

9.6.4 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 de-

Copyright © 1994 Thinking Machines Corporation

pend on the type of routine you are calling. 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 can 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.

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

Example 2: Solving Linear Systems Using Householder Transformations (In-Core *QR* Factorization and Solver Routines)

Figure 3 through Figure 5 show how a multiple-instance problem is set up for the in-core routines that solve linear systems using Householder transformations (the "*QR*" routines). The three-dimensional parallel variable **`A`** in Figure 3 contains four matrices to be factored (four instances of the matrix `A`). Each matrix A has dimensions m x n, and is (optionally) contained in a larger matrix embedded in **`A`**. The data axes, which count the rows and columns of the matrices, can be any two axes of **`A`**; you need not use the first and second axes for this purpose.

Copyright © 1994 Thinking Machines Corporation

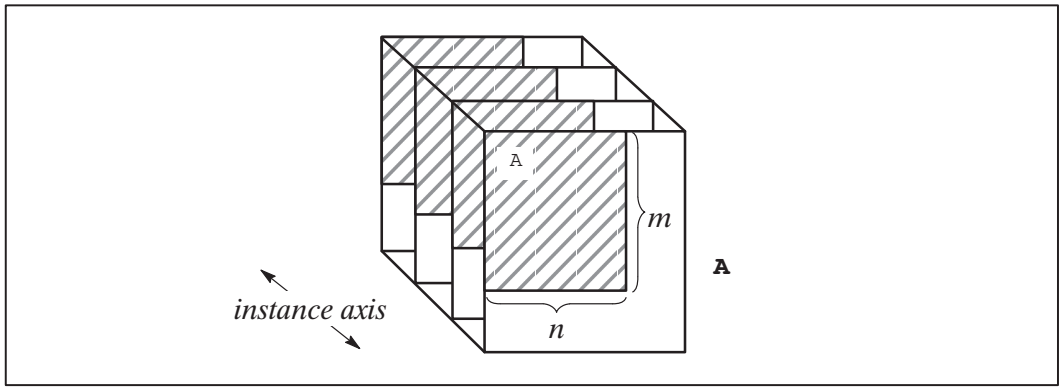

Figure 3. The matrices to be factored have size *m* x *n* and may be contained in larger matrices embedded in A.

Figure 4 shows four instances of the linear system AX = B. The right-hand-side matrices B are embedded in the parallel variable **B**; you must use the same axes to count the rows and columns of the instances in **B** as in **A**. The parameter *r* represents the number of columns, or right-hand-side vectors, in each matrix B; thus, each B has shape [*m*][*r*].

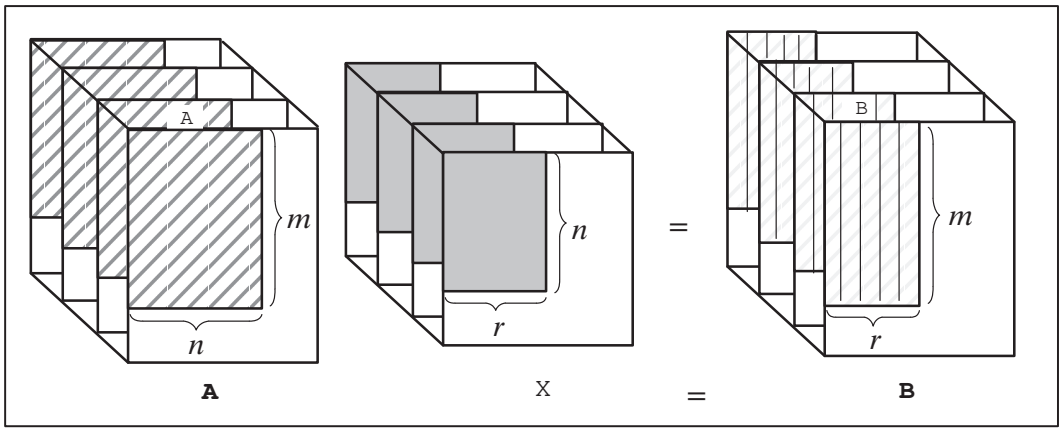

Figure 4. The linear systems AX = B

Upon completion of the solver routine, the first *n* rows of each matrix B are overwritten with the least squares solution to AX = B, as shown in Figure 5.

Copyright © 1994 Thinking Machines Corporation

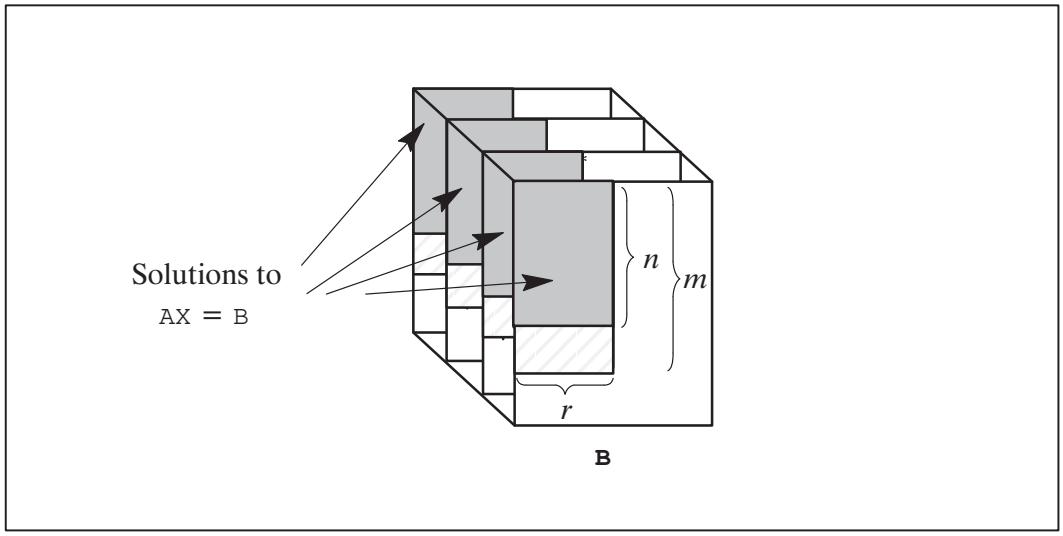

Figure 5. The matrices embedded in the parallel variable `B` are partially overwritten with the solutions when the solver routine completes.

Example 3: 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, 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.

Copyright © 1994 Thinking Machines Corporation

Example 4: 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).

10 Using the C* Interface to CMSSL

This section contains information about running C* programs that call CMSSL routines. The following topics are included:

- creating a C* program that calls CMSSL routines
- restrictions and performance guidelines
- complex data types and macros
- auxiliary file I/O routines
- the CMSSL safety mechanism

10.1 Creating a C* Program that Calls CMSSL Routines

To use CMSSL from within a C* program, follow these steps:

1. Place calls to CMSSL routines into C* code. Your code must satisfy the restrictions described in Section 10.2.
2. Include the header file **`<cm/cmssl.h>`** in each C* program unit that calls CMSSL routines.
3. Use the **`cs`** command to compile your code.
4. Include the required switches on the **`cs`** command line when linking your code; see Section 10.1.2 for details.

The remainder of this section contains details about the steps listed above.

Copyright © 1994 Thinking Machines Corporation

10.1.1 Including the CMSSL Header File

You must place the following line at the top of any C* program unit that makes a CMSSL call:

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

This line allows the C* program to access the header file **<cm/cmssl.h>**, which contains prototypes for the routines and defines the symbols and types required by the interface.

If the C* compiler cannot find the CMSSL include file, check your partition manager for the existence of a path to the appropriate directory. If the file appears to be missing, consult your system administrator or your Thinking Machines Corporation customer service representative.

10.1.2 Compiling and Linking

Two alternative sets of **cs** command lines for linking with CMSSL are presented below. Note that you must have CM Fortran installed to run CMSSL for C*.

Linking with All Required Libraries Explicitly

If the C* program *program* calls CMSSL routines, you can compile it and link it with CMSSL using one of the command lines shown below. In these command lines, you link with all required libraries explicitly.

Compiling to run without vector units:

```
% cs -g -cm5 -sparc -o program program.cs -lcmsslcscm5
-lcmffio_cmfs -lcmfcompiler -lcmfs_cm5 -lcmf77
```

Compiling to run with vector units:

```
% cs -g -cm5 -vu -o program program.cs -lcmsslcscm5vu
-lcmffio_cmfs -lcmfcompiler -lcmfs_cm5 -lcmf77
```

Compiling to use the nodal CMSSL library on machines with vector units:

```
% cs -g -cm5 -vu -node -o program program.cs
-lcmsslcscm5vu-node -lcmffio -lcmfcompiler -lcmf77
```


Copyright © 1994 Thinking Machines Corporation

The **-lcmffio_cmfs**, **-lcmfcompiler**, **-lcmfs_cm5**, and **-lcmf77** switches link with the CM Fortran support libraries required by global CMSSL. The **-lcmffio**, **-lcmfcompiler**, and **-lcmf77** switches link with the CM Fortran support libraries required by nodal CMSSL. These switches must follow the **-lcmsslcm5**, **-lcmsslcm5vu**, or **-lcmsslcm5vu-node** switch on the command line.

The **-g** switch provides Prism support, and is required only when you are linking.

If you are using nodal CMSSL, you must use CMMD Version 3.0 or higher.

Linking with CMSSL Using +lcmssl

If you are using C* Version 7.1.1 or higher and you are compiling to run with vector units, you can use the **+lcmssl** switch as an alternative to including all required libraries explicitly on the **cs** command line. The **+cmssl** switch links a program with the appropriate CMSSL library and all other required libraries, as follows:

C* Version 7.1.1 or higher: compiling to run with vector units:

```
% cs -cm5 -vu -o program program.cs +lcmssl
```

Plans for C* Version 7.2 Beta include adding support for the **+cmssl** switch for compiling to run without vector units and compiling to use the nodal library on machines with vector units. Thus, beginning with C* 7.2 Beta, you will be able to use these link lines:

C* Version 7.2 Beta or higher: compiling to run without vector units:

```
% cs -cm5 -sparc -o program program.cs +lcmssl
```

C* Version 7.2 Beta or higher: compiling to use the nodal CMSSL library on machines with vector units:

```
% cs -cm5 -vu -node -o program program.cs +lcmssl
```

The **+lcmssl** option selects the appropriate CMSSL library, as well as the required Prism and CM Fortran support libraries.

Version 3.2, July 1994
Copyright © 1994 Thinking Machines Corporation

Using the Correct Versions of CMOST, C*, and CM Fortran

CMSSL is a layered product. Any CMSSL version requires specific versions of CMOST, C*, and CM Fortran. If these dependencies are not observed, proper operation of the CMSSL routines is unlikely. See Section 2.2 for information about which versions of the C* compiler, CM Fortran, and CMOST are required.

10.1.3 Executing CMSSL Programs

Execute a C* CMSSL program just as you would any compiled C* program.

10.2 Restrictions and Performance Guidelines

To use the C* CMSSL routines effectively, you must adhere to certain restrictions and guidelines that stem from the differences in programming model between C* and CM Fortran.

C* supports certain programming styles, data objects, and implementation details that CM Fortran does not support, and vice versa. These differences are resolved in two ways:

- CMSSL imposes restrictions on supported data types and argument passing mechanisms. (Some of these restrictions may be removed in future releases of the CMSSL C* interface.)
- CMSSL supports certain C*-specific features, but at a performance cost. In particular, you can pass a parallel structure slot or array element to CMSSL, but at a performance cost, because CMSSL copies the slot or array element to a temporary parallel variable. To optimize performance, you must understand which situations incur a performance cost, and what that cost is.

Each subsection below describes a key issue and presents the related restrictions and/or performance guidelines. Sections 10.2.8 and 10.2.9 provide summaries of the restrictions and guidelines, respectively.

Copyright © 1994 Thinking Machines Corporation

10.2.1 CMSSL Expects Only Certain Numeric Data Types

The C* CMSSL interface accepts six data types for parallel variables: **int**, float, double, CMSSL_complex_t, CMSSL_dcomplex_t, and **CMSSL_logical_t**. The CMSSL_complex_t and CMSSL_dcomplex_t types are discussed in Section 10.3. Because parallel variables are passed by reference (see Section 10.2.3), data type promotion is not performed on them.

In the current release, CMSSL does not support masks or scalar logical values whose elements are C-style logical quantities (nonzero = true), and does not support data of type **bool**. It does support masks and scalar logical values whose elements are of type **CMSSL_logical_t**, defined in the header file **<cm/cmssl.h>** as follows:

```
typedef enum {
   CMSSL_true = -1,
   CMSSL_false = 0
} CMSSL_logical_t;
```

10.2.2 Context is Ignored

Except for cases in which an associated mask parameter is provided in the call, a CMSSL routine references and/or modifies any parallel variables passed to it without regard to any context established in the calling routine, for example in a **where** construct. In effect, the CMSSL routine operates within an **everywhere** block.

10.2.3 CMSSL Expects Parallel Arguments to Be Passed by Reference

C* requires that parallel variables passed by value to an external routine be of shape **current**. Because many CMSSL routines require several parallel arguments with different shapes in the same call, all parallel arguments passed to CMSSL are passed by reference.

Thus, CMSSL requires that you pass it a scalar pointer to a parallel variable. You can pass a pointer to a simple parallel variable, a member of a parallel structure, or an element of a parallel array. However, C* does not allow you to create a pointer to a left-indexed variable or to a parallel array indexed

Copyright © 1994 Thinking Machines Corporation

with a parallel index; therefore, you cannot pass such objects to CMSSL. For example, given

```
shape [16][64] sh1;
int:sh1 x[16];
```

you can pass CMSSL **x**[5], but not **x**, [.][37]**x**, [3][4]**x**, or [3][4]**x**[3]. As another example, given

```
shape [64] sh2;
double:sh2 y, x[128];
int:sh2 x_offset;
x[x_offset] = y;
```

the expression **x[x_offset]** is a legal, but because **x_offset** is parallel, **&x[x_offset]** is not legal C*; therefore, you cannot pass **x[x_offset]** to CMSSL.

Scalar input arguments are passed by value; scalar output arguments are passed by reference.

10.2.4 CMSSL Does Not Accept Sections of Parallel Variables

If you are familiar with CM Fortran as well as C*, note that you cannot pass a CMSSL routine a "section" of a C* parallel variable (that is, the equivalent of a section of a CM array in CM Fortran). The reason for this restriction is that CMSSL expects parallel variables to be passed by reference, and C* does not allow you to take the address of a parallel variable with a partially indexed shape.

10.2.5 CMSSL Expects One Data Element per Position

CMSSL is optimized to work on multiple instances of a problem at the same time. In the current release, the data axes and instance axes of a problem must all be parallel (that is, they must be axes of the data's shape). While a parallel variable may be an aggregate — for example, an array or structure — you may pass only a reference to a *single* array element or structure slot to CMSSL. Only one atomic (non-aggregate) data item per position is operated on by CMSSL routines. (The only exception to this rule is that CMSSL

Copyright © 1994 Thinking Machines Corporation

supports complex data types, which are implemented as aggregate data types in C*; see Section 10.3.) For example, given

```
shape [16][32]shape1;
float:shape1 A[8];
```

you may pass CMSSL the component **A**[5], but you may not pass it the entire array **A**; CMSSL will operate on only one element. Moreover, passing CMSSL a component of an aggregate parallel variable may have a performance cost, as discussed in Sections 10.2.6 and 10.2.7, below.

Local and Non-Local Axes

An axis of a shape can be local (reside within a single processing element) or non-local (span multiple processing elements), depending on the shape's layout. (For more information about layout, see the discussion of the **allocate_detailed_shape** function in the *C* Programming Guide*). A *processing element* is a vector unit on machines with vector units, or a SPARC node on machines without vector units.

10.2.6 CMSSL Prefers Doubleword-Aligned Arguments

CMSSL performs best when the data element in each position of a parallel argument is *doubleword-aligned* within the processing element memory — that is, begins at a memory address that is a multiple of 8. In a C* parallel variable, the first data element in each processing element begins on a doubleword boundary. However, if the data element is an aggregate (for example, an array or structure), components after the first one may not be doubleword-aligned. If you pass one of these non-doubleword-aligned components to CMSSL, CMSSL copies the data into a temporary variable that is doubleword-aligned, with a performance and memory overhead cost. (See Section 10.2.10 for specific information about the cost.)

The rules listed below govern memory alignment of the components of an aggregate data element in C*. (These are the same rules that apply to ordinary variables in C.)

- **int** and `float` components have singleword alignment.
- `double`, `CMSSL_complex_t`, and `CMSSL_dcomplex_t` components have doubleword-alignment.

Copyright © 1994 Thinking Machines Corporation

- The memory alignment of an aggregate, *agg*, equals the largest alignment of any component of *agg*. This rule applies recursively to aggregates of aggregates.

Examples

For example, if we have

```
shape [16][32]shape1;
float:shape1 A[8];
```

then while the first data element, **A**[0], is doubleword-aligned in each processing element, the second element, **A**[1], begins just four bytes after **A**[0], and therefore is not doubleword-aligned. Thus, supplying **A**[1] to CMSSL incurs a performance cost.

Similarly, if we have

```
shape [16][32]shape1;
struct{
    float aa;
    int bb;
}:shape1 zz[10];
```

then while **zz[0].aa** is doubleword-aligned, **zz[0].bb** begins just four bytes after **zz[0].aa** does, and therefore is not doubleword-aligned. Thus, supplying **zz[0].bb** to CMSSL incurs a performance cost.

As another example, if we have

```
shape [16][32]shape1;
struct{
    float aa;
    double cc;
}:shape1 zz[10];
```

the whole structure has doubleword alignment, because the largest memory alignment within it is the doubleword alignment of **cc**.

10.2.7 CMSSL Prefers Contiguous Data Elements

In the current release, CMSSL performs best when successive data elements in each position of a parallel argument are allocated contiguously within the

Copyright © 1994 Thinking Machines Corporation

processing element memory. In C*, a component of an aggregate data element is not allocated contiguously. If you pass such a component to CMSSL, CMSSL copies the data into a temporary variable that is contiguous. This copy incurs a performance and memory overhead cost. Thus, to avoid this overhead, avoid passing components of aggregate parallel arguments to CMSSL. (See Section 10.2.10 for specific information about the cost.)

Examples

As an example, suppose we have

```
shape [16][32]sh;
int x:sh [64];
```

The component **x**[0] of **x** is not contiguous, since **x**[0] in one position and **x**[0] in the next position are separated by **x**[1] through **x**[63].

As another example, consider the declaration from the previous section:

```
shape [16][32]shape1;
struct{
     float aa;
     double cc;
}:shape1 zz[10];
```

Suppose that within the memory of a given processing element, **zz** starts at memory location 208. The first element, **zz[0].aa**, occupies locations 208−211. Since **zz[0].cc** is of type `double` and therefore must be double-word-aligned, it begins at the next multiple-of-8 location, 216, and occupies locations 216−223. Locations 212−215 are unfilled. This pattern repeats for all elements of **zz**. Because the elements of **zz** are not contiguous, if you supply any element of **zz** to CMSSL, CMSSL makes a copy of the data, at a performance cost.

10.2.8 Summary of Restrictions

In the current release, CMSSL imposes the following restrictions on C* programs:

- The C* CMSSL interface accepts six data types for parallel variables: **int**, `float`, `double`, `CMSSL_complex_t`, `CMSSL_dcomplex_t`, and **CMSSL_logical_t**. In the current release, CMSSL does not support

Copyright © 1994 Thinking Machines Corporation

masks or scalar logical values whose elements are C-style logical quantities (nonzero = true), and does not support data of type **bool**. It does support masks and scalar logical values whose elements are of type **CMSSL_logical_t**.

- CMSSL references and/or modifies any parallel variables passed to it without regard to any context established in the calling routine.
- All parallel variables passed to CMSSL are passed by reference. CMSSL requires a scalar pointer to a parallel variable.
- You cannot pass a CMSSL routine a "section" of a C* parallel variable (the equivalent of a section of a CM array in CM Fortran).
- Scalar input parameters are passed by value; scalar output parameters are passed by reference.
- While a parallel variable may be an *aggregate* — for example, an array or structure — you may pass only a reference to a single array element or structure slot (a component of the aggregate) to CMSSL. Only one atomic (non-aggregate) data item per position is operated on by CMSSL routines. The only exception to this rule is that CMSSL supports complex data types, which are implemented as aggregate data types in C*.

10.2.9 Summary of Performance Guidelines

Follow these guidelines to avoid extra performance overhead:

- Be sure that the parallel arguments you pass to CMSSL are doubleword-aligned. If you pass a non-doubleword-aligned parallel argument, CMSSL copies the data into a new variable that is doubleword-aligned, with a performance cost.
- Be sure that the parallel arguments you pass to CMSSL contain contiguous data elements. If you pass a parallel argument containing non-contiguous elements, CMSSL copies the data into a new variable that is contiguous, at a performance cost.

The simplest way to satisfy these guidelines is to pass only atomic (non-aggregate) parallel arguments to CMSSL. If you pass parallel structure

Copyright © 1994 Thinking Machines Corporation

members or parallel array elements to CMSSL, you will incur the cost of the copy.

10.2.10 More on Performance Costs

As discussed above, if parallel data passed as an argument to a CMSSL routine is not contiguous in memory or is not doubleword-aligned, CMSSL guarantees contiguity and alignment of the data by allocating a contiguous, doubleword-aligned temporary parallel variable of the same shape and data type. If the argument's shape has *p* positions and the size of its data type is *s* bytes, the temporary variable will consume at least *ps* bytes of parallel heap memory for the duration of the call (somewhat more if one or more axes of its shape end in garbage positions).

Depending on how the CMSSL routine uses such a parameter, there may be time overheads as well, because local copying to and from the temporary variable consumes local memory bandwidth. Input/output parameters incur the heaviest copy cost because they are copied both in and out. Input-only parameters are copied in only; output-only parameters are copied out only.

Certain CMSSL routines, such as **`CMSSL_part_scatter_setup`** and **`CMSSL_fft_setup`**, accept one or more parallel variables as "templates" without looking at their data values at all. Templates are used only to extract information about shape, data type, and stride; the actual data values of such arguments are irrelevant to such routines. In these cases, no copy in or copy out is done (but the contiguous aligned temporary parallel variable must be allcoated anyway).

10.3 Complex Data Types and Macros

The complex data types `CMSSL_complex_t` and `CMSSL_dcomplex_t` are defined in the file **`<cm/cmssl_complex.h>`**, which is included by **`<cm/cmssl.h>`**. These types are defined so that, although each slot has single-word memory alignment, the whole `CMSSL_complex_t` or `CMSSL_dcomplex_t` is always doubleword-aligned. (Note that this is not usually true for a structure containing two `float` slots.) Thus, you can pass complex numbers to CMSSL without incurring the performance cost of a copy. To accomplish this, `CMSSL_complex_t` and `CMSSL_dcomplex_t` are defined

Copyright © 1994 Thinking Machines Corporation

using a union with a `double` data element. The definitions from **<cm/cmssl_complex.h>** are reproduced here:

```
#ifndef __CMSSL_CS_COMPLEX__
#define __CMSSL_CS_COMPLEX__

/* Force doubleword alignment for single complex */
typedef union { struct { float re_; float im_; } values;
                double alignment ; } CMSSL_complex_t;

/* Make double complex consistent with single complex */
typedef union { struct { double re_; double im_; } values;
                double alignment ; } CMSSL_dcomplex_t;
/*
 * initialize both parts with z = { <expr>, <expr> }
 * assign real part with re(z) = <expr>
 * assign imag part with im(z) = <expr>
 */
#define re(z) ((z).values.re_)
#define im(z) ((z).values.im_)

#endif __CMSSL_CS_COMPLEX__
```

Since C* does not currently support complex arithmetic functions, you must perform all complex arithmetic operations in terms of the real and imaginary parts of complex numbers. To facilitate these operations, CMSSL provides the two macros, **`re()`** and **`im()`**, defined in the include file above. These macros return the real and imaginary parts of a data element of type `CMSSL_complex_t` or `CMSSL_dcomplex_t`.

The definitions of `CMSSL_complex_t` and `CMSSL_dcomplex_t` may change in future releases. For future compatibility, we recommend that you use the **`re()`** and **`im()`** macros.

10.4 Auxiliary File I/O Routines

The C* interface to CMSSL includes the auxiliary I/O routines listed below. These routines open, close, and manipulate files.

Reserving file units	**`CMSSL_file_reserve_unit`**
	`CMSSL_fe_file_reserve_unit`

Copyright © 1994 Thinking Machines Corporation

	CMSSL_file_release_unit **CMSSL_fe_file_release_unit**
Opening and closing files	**CMSSL_file_open** **CMSSL_file_fdopen** **CMSSL_fe_file_open** **CMSSL_file_close** **CMSSL_fe_file_close**
Rewinding files	**CMSSL_file_rewind** **CMSSL_fe_file_rewind**
Removing files	**CMSSL_file_remove** **CMSSL_fe_file_remove**
Writing to files	**CMSSL_pvar_to_file** **CMSSL_pvar_to_file_fms** **CMSSL_pvar_to_file_so**
Reading from files	**CMSSL_pvar_from_file** **CMSSL_pvar_from_file_fms** **CMSSL_pvar_from_file_so**
Reporting I/O errors	**CMSSL_file_perror**

The component **fe** in a routine name indicates that the routine applies to serial files as opposed to parallel files. The **CMSSL_file_fdopen** routine provides support for sockets. The **fms** and **so** suffixes specify fixed machine size and serial order, respectively, and correspond to types of CM Fortran I/O: the **fms** routines read and write data in parallel order, and the **so** routines read and write data in serial order.

Copyright © 1994 Thinking Machines Corporation

Use the auxiliary I/O routines to open, close, and rewind the files in which the following CMSSL parallel I/O routines store data:

CMSSL_save_gen_lu	**CMSSL_gen_lu_setup_ext**
CMSSL_restore_gen_lu	**CMSSL_gen_lu_factor_ext**
CMSSL_save_gen_lu_fms	**CMSSL_gen_lu_solve_ext**
CMSSL_restore_gen_lu_fms	**CMSSL_gen_qr_factor_ext**
CMSSL_save_gen_qr	**CMSSL_gen_qr_solve_ext**
CMSSL_restore_gen_qr	**CMSSL_save_fast_rng_temps**
CMSSL_save_gen_qr_fms	**CMSSL_restore_fast_rng_temps**
CMSSL_restore_gen_qr_fms	**CMSSL_save_vp_rng_temps**
CMSSL_gen_matrix_mult_ext	**CMSSL_restore_vp_rng_temps**

Use the auxiliary I/O routines that reserve and release file units (**CMSSL_file_reserve_unit**, **CMSSL_fe_file_reserve_unit**, **CMSSL_file_release_unit**, and **CMSSL_fe_file_release_unit**) if, and only if, your application program includes both of the following:

- a C* function that calls one of the CMSSL parallel I/O routines listed above
- a CM Fortran routine that performs CM Fortran parallel I/O

It is possible that the CMSSL parallel I/O routine could use the same file unit(s) that the CM Fortran parallel I/O routine uses. To avoid this conflict and the unpredictable behavior that may result, your program should reserve all unit numbers it plans to use in the CM Fortran parallel I/O, before doing any CM Fortran parallel I/O or CMSSL parallel I/O. Supply each CM Fortran file unit number that will be used in CM Fortran parallel I/O to **CMSSL_file_reserve_unit()**. After all CM Fortran parallel I/O is finished, you may release the reserved file units by supplying each unit number in a call to **CMSSL_file_release_unit()**.

Use the **CMSSL_file_perror** routine to obtain information about I/O errors that occur when you call the CMSSL parallel I/O routines. If you supply **CMSSL_file_perror** with the value 0, **CMSSL_file_perror** prints the error message associated with the most recent value of **CMFS_errno** (an error code that is set when a CMFS library routine encounters an error). If you supply **CMSSL_file_perror** with a non-zero value *n*, **CMSSL_file_perror** prints the error message associated with the value *n* of the CM Fortran I/O status variable, IOSTAT.

Copyright © 1994 Thinking Machines Corporation

10.5 The CMSSL Safety Mechanism

You can use the CMSSL safety mechanism in two ways:

- by setting the environment variable **`CMSSL_SAFETY`**
- by using the calls **`CMSSL_get_safety`** and **`CMSSL_set_safety`** in a program

10.5.1 Safety Mechanism Features

The CMSSL safety mechanism offers two basic features: it synchronizes the CM-5 parallel 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.

Synchronization

The CM-5 parallel processing elements and partition manager operate asynchronously with respect to one another. Without the CMSSL safety mechanism, an error that occurs in the parallel processing elements is not reported to the partition manager until the next time the partition manager requests information from or checks the status of the elements. Such a request or status check is known as an *implicit synchronization* because it has the side effect of synchronizing the processing elements and partition manager, allowing the processing elements to report any accumulated errors. When an implicit synchronization occurs, there is no way to tell exactly when the reported error occurred, or which module of code produced it.

The CMSSL safety mechanism addresses this problem by forcing *explicit synchronization* between the parallel processing elements and the partition manager before, after, and within each CMSSL call in your code. The safety mechanism traps and reports errors, indicating when the errors occurred in relation to the synchronization points.

Error Checking and Reporting

The safety mechanism can perform error checking and generate run-time error information at several levels of detail. You can turn safety checking on at any level during all or part of a program. One level checks for errors in the usage and arguments of the CMSSL calls in your program; a more

Copyright © 1994 Thinking Machines Corporation

detailed level also checks for errors generated by internal CMSSL routines. Examples of errors found and reported by the safety mechanism include the following:

- A supplied or returned data element that should be numerical is not; for example, it is identified as a "Not a Number" (NaN) or as infinity. (NaNs are defined in the IEEE Standard for Binary Floating-Point Arithmetic.)
- The code generates a division by 0 (for example, because of bad data, a user error, or an internal software problem).
- The code references a memory location that it has not initialized. The safety mechanism identifies this kind of error by writing NaNs to all allocated processing element memory. If the code references a memory location without first explicitly assigning it a numerical value, the NaN at that location causes further errors that make the original erroneous reference easy to find.

As more debugging checks and safety levels are added in future releases, CMSSL safety checking will become more exhaustive.

10.5.2 Levels of Error Checking

The CMSSL safety mechanism currently provides the following levels:

0 (**`off`**)	Turns off the safety mechanism. Explicit synchronization and error checking are not performed. This level is appropriate for production runs of code that has already been thoroughly tested.
1 (**`on`**)	Checks for and reports errors caused by incorrect usage or arguments in high-level-language CMSSL calls. Performs explicit synchronization before and after each call and locates each error with respect to the synchronization points. This safety level is appropriate during program development or during runs for which a small performance penalty can be tolerated.
9 (**`full`**)	Checks for and reports all level 1 errors, and in addition any errors generated by the lower levels of code that are called by the high-level-language CMSSL

Copyright © 1994 Thinking Machines Corporation

calls. Performs explicit synchronization in these lower levels of code and locates each error with respect to the synchronization points. This level performs all implemented error checking and exacts a very high performance price. It is appropriate for detailed debugging when a problem occurs. If you cannot analyze and correct the problem, provide your local site coordinator, applications engineer, or Thinking Machines Corporation customer service representative with the output generated by level 9 safety checking.

At levels 1 and 9, some safety mechanism error messages are displayed at the terminal when you run the program; other information appears in the backtrace when you use a debugger such as Prism.

If you report a software problem to your local site coordinator, applications engineer, or Thinking Machines Corporation customer service representative, you may be asked to run your program with the CMSSL safety mechanism enabled at a level other than 0, 1, or 9. These additional levels are used for pinpointing problems in the internal software or for obtaining internal status information.

10.5.3 Setting the CMSSL Safety Environment Variable

To set the CMSSL safety level using the **CMSSL_SAFETY** environment variable, issue the command

```
setenv CMSSL_SAFETY { 0 | 1 | 9 | off | on | full }
```

choosing one of the listed options. As indicated above, 0 is equivalent to **off**, 1 to **on**, and 9 to **full**.

The advantage of using the **CMSSL_SAFETY** environment variable is that you can set or change the safety level without recompiling your code.

10.5.4 Using CMSSL Safety from within a Program

To set the CMSSL safety level, issue the following call and specify the desired level in the integer argument `n`:

```
CMSSL_set_safety (n);
```


Version 3.2, July 1994
Copyright © 1994 Thinking Machines Corporation

To obtain the current CMSSL safety level, issue the following call:

```
CMSSL_get_safety ( );
```

The advantage of using these calls from within a program is that you can set or obtain the safety level at any point within your code. However, you must recompile the code each time you change these calls.

NOTE

The inner product, 2-norm, outer product, matrix vector multiplication, vector matrix multiplication, and matrix multiplication routines perform error checking only when the CMSSL safety mechanism is on. Therefore, we strongly recommend that you turn CMSSL safety on when testing new programs that call these routines.

Copyright © 1994 Thinking Machines Corporation