Chapter 1

Introduction to CMSSL for CM Fortran

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

- about CMSSL
- contents of CMSSL for CM Fortran
- notes on terminology and conventions
- data types supported
- support for multiple instances
- numerical stability for the linear algebra routines
- numerical complexity
- CM Fortran performance enhancements with CMSSL

1.1 About CMSSL

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

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. Many 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.

1.2 Contents of CMSSL for CM Fortran

The CM Fortran interface to CMSSL consists of a set of library routines and a safety mechanism.

1.2.1 Library Routines

Listed below are the operations included in CMSSL for CM Fortran on the CM-5.

- **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 CM arrays. 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 CM array.

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 CM array, is added to the destination CM array, or is added to a second CM array. 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 CM array, is added to the destination CM array, or is added to a second CM array. 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 CM array, is added to the destination CM array, or is added to a second CM array. 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 CM array, is added to the destination CM array, or is added to a second CM array. 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 pointers 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 arrays 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 $m \times n$ 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 $LX=B$, $L^{T}X=B$, $L^{-1}X=B$, $L^{-T}X=B$, $UX=B$, and $U^{T}X=B$. 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 $m \times n$ 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 $(QR)^{T}X=B$. (With pivoting, these equations become $QRP^{-1}X = B$ and $(QRP^{-1})^{T}X=B$.) 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 A 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 $n \times n$ 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.

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, and are included in the library primarily for compatibility with the CM-200. 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 $A^{\sim} = M_1{}^{-1}AM_2{}^{-1}$, 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 arrays 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.

Copyright © 1994 Thinking Machines Corporation

- *Generalized Eigensystem Analysis of Dense Hermitian Matrices*

  Given a CM array containing one or more dense Hermitian matrices A, and a CM array containing corresponding positive definite matrices B, this routine solves $AQ = BQ\lambda$, 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 $\{\lambda, x\}$ to the real standard or generalized eigenvalue problem $Lx = \lambda Bx$. 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 $\{\lambda, x\}$ to the real standard or generalized eigenvalue problem $Lx = \lambda Bx$. 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 CM array.

  - *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.

- *Array Conversion Utilities*

  These utilities convert real arrays into complex arrays suitable for input to the real-to-complex FFT, and convert complex arrays (supplied in the format produced by the complex-to-real FFT) to real arrays.

- **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), $dy_i(x)/dx = f_i(x, y_1, \ldots, y_N)$ $i=1, ..., N$ 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 $c^{\mathrm{T}}x$ subject to the conditions $Mx = b$, $0 \leq x \leq u$, where M is an $m \times n$ matrix, c is a coefficient vector, and $c^{\mathrm{T}}x$ is referred to as the *cost*. The upper bound vector u may be infinity in one or more components.

Copyright © 1994 Thinking Machines Corporation

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

    This lagged-Fibonacci RNG is faster than the standard RNG included in CM Fortran. 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.

  - *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 array. Successive calls can provide an accumulation of totals.

  - *Range Histogram*

    The range histogram routines record the distribution of source array 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 front-end array, and one in which it is stored in a CM array.

- **Communication Primitives**
  - *Polyshift*

    This routine performs multidirectional and/or multidimensional array shifts in an array geometry.

Copyright © 1994 Thinking Machines Corporation

- *All-to-All Rotation*

  Given a real or complex array and a designated axis, this routine performs an in-place, stepwise rotation of every array value on the axis to every location along the axis.

- *All-to-All Broadcast*

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

- *All-to-All Reduction*

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

- *Matrix Transpose*

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

- *Sparse Gather Utility*

  These routines gather elements of a vector into an array using pointers supplied by the application. Pre-processing is performed by an associated setup routine.

- *Sparse Scatter Utility*

  These routines scatter elements of an array to a vector using pointers 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 CM array into a destination CM array. 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 array (with one local dimension and one non-local dimension) to a similar destination array. 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 Pointers*

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

  - Given an *element nodes* array that describes an unstructured mesh, one routine produces the corresponding *dual connectivity* array.
  - Given a dual connectivity array, a second routine returns a permutation that reorders the mesh elements to form discrete *partitions*.
  - Given a pointers array and a permutation, a third routine reorders the pointers array along any specified axis using the permutation.
  - Given a pointers array, a fourth routine changes the pointer values for improved locality and returns the renumbering mapping.

  If you derive a pointers array from a mesh, reorder it using the permutation returned by the partitioning routine, and then supply these reordered pointers to the setup routine for the partitioned gather or scatter operation, the setup routine takes

Copyright © 1994 Thinking Machines Corporation

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 pointers array that is 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 pointers array that is 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 (or *trace*) just once, and then use the trace 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 a trace.

- *Vector Move (Extract and Deposit)*

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

- *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.

Copyright © 1994 Thinking Machines Corporation

- *Permutation along an Axis*

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

- *Combination of Permutations*

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

- *Zeroing Routine*

  This routine zeroes a CM array.

- *Send-to-NEWS and NEWS-to-Send Reordering*

  On the CM-200, these routines allow you to change the ordering of specified axes of a CM array from send to NEWS ordering or vice-versa. On the CM-5, these routines have no effect because send and NEWS ordering are the same. They are provided only for compatibility with the CM-200. (Refer to the CM Fortran documentation set for information about send and NEWS ordering.)

Table 1 lists the CMSSL routines for CM Fortran on the CM-5, along with the chapters that describe them.

Copyright © 1994 Thinking Machines Corporation

Table 1. CMSSL Routines for CM Fortran.

Operation	Chapter	Routines
Inner product	3	**gen_inner_product** **gen_inner_product_noadd** **gen_inner_product_addto** **gen_inner_product_c1** **gen_inner_product_c1_noadd** **gen_inner_product_c1_addto** **gbl_gen_inner_product** **gbl_gen_inner_product_noadd** **gbl_gen_inner_product_addto** **gbl_gen_inner_product_c1** **gbl_gen_inner_product_c1_noadd** **gbl_gen_inner_product_c1_addto**
2-norm	3	**gen_2_norm** **gbl_gen_2_norm**
Outer product	3	**gen_outer_product** **gen_outer_product_noadd** **gen_outer_product_addto** **gen_outer_product_c2** **gen_outer_product_c2_noadd** **gen_outer_product_c2_addto**
Matrix vector multiplication	3	**gen_matrix_vector_mult** **gen_matrix_vector_mult_noadd** **gen_matrix_vector_mult_addto** **gen_matrix_vector_mult_c1** **gen_matrix_vector_mult_c1_noadd** **gen_matrix_vector_mult_c1_addto**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Vector matrix multiplication	3	**gen_vector_matrix_mult** **gen_vector_matrix_mult_noadd** **gen_vector_matrix_mult_addto** **gen_vector_matrix_mult_c2** **gen_vector_matrix_mult_c2_noadd** **gen_vector_matrix_mult_c2_addto**
Infinity norm	3	**gen_infinity_norm**
Matrix multiplication	3	**gen_matrix_mult** **gen_matrix_mult_noadd** **gen_matrix_mult_addto** **gen_matrix_mult_t1** **gen_matrix_mult_t1_noadd** **gen_matrix_mult_t1_addto** **gen_matrix_mult_h1** **gen_matrix_mult_h1_noadd** **gen_matrix_mult_h1_addto** **gen_matrix_mult_t2** **gen_matrix_mult_t2_noadd** **gen_matrix_mult_t2_addto** **gen_matrix_mult_h2** **gen_matrix_mult_h2_noadd** **gen_matrix_mult_h2_addto** **gen_matrix_mult_t1_t2** **gen_matrix_mult_t1_t2_noadd** **gen_matrix_mult_t1_t2_addto**
Matrix multiplication with external storage	3	**gen_matrix_mult_ext**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Arbitrary elementwise sparse matrix operations	4	**sparse_matvec_setup** **sparse_matvec_mult** **sparse_mat_gen_mat_mult** **deallocate_sparse_matvec_setup** **sparse_vecmat_setup** **sparse_vecmat_mult** **gen_mat_sparse_mat_mult** **deallocate_sparse_vecmat_setup**
Arbitrary block sparse matrix operations	4	**block_sparse_setup** **block_sparse_matrix_vector_mult** **vector_block_sparse_matrix_mult** **block_sparse_mat_gen_mat_mult** **gen_mat_block_sparse_mat_mult** **deallocate_block_sparse_setup**
Grid sparse matrix operations	4	**grid_sparse_setup** **grid_sparse_matrix_vector_mult** **vector_grid_sparse_matrix_mult** **grid_sparse_mat_gen_mat_mult** **gen_mat_grid_sparse_mat_mult** **deallocate_grid_sparse_setup**
Gaussian elimination	5	**gen_lu_factor** **save_gen_lu** **save_gen_lu_fms** **restore_gen_lu** **restore_gen_lu_fms** **gen_lu_solve** **gen_lu_solve_tra**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Gaussian elimination (cont'd)	5	**gen_lu_apply_l** **gen_lu_apply_l_tra** **gen_lu_apply_l_inv** **gen_lu_apply_l_inv_tra** **gen_lu_apply_u_inv** **gen_lu_apply_u_inv_tra** **gen_lu_get_l** **gen_lu_get_u** **gen_lu_zero_rows** **gen_lu_infinity_norm_inv** **deallocate_gen_lu**
Linear system solvers using Householder transformations	5	**gen_qr_factor** **save_gen_qr** **save_gen_qr_fms** **restore_gen_qr** **restore_gen_qr_fms** **gen_qr_solve** **gen_qr_solve_tra** **gen_qr_apply_q** **gen_qr_apply_q_tra** **gen_qr_apply_r_inv** **gen_qr_apply_r_inv_tra** **gen_qr_get_r** **gen_qr_apply_p** **gen_qr_apply_p_inv** **gen_qr_zero_rows** **gen_qr_extract_diag** **gen_qr_deposit_diag** **gen_qr_infinity_norm_inv** **gen_qr_r_infinity_norm_inv** **deallocate_gen_qr**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Matrix inversion	5	**gen_gj_invert**
Gauss-Jordan system solver	5	**gen_gj_solve**
Gaussian elimination with external storage	5	**gen_lu_setup_ext** **gen_lu_factor_ext** **gen_lu_solve_ext**
QR factorization and least squares solution with external storage	5	**gen_qr_factor_ext** **gen_qr_solve_ext**
Banded system factorization and solver routines (unified)	6	**gen_banded_factor** **gen_banded_solve** **deallocate_banded**
Banded system factorization and solver routines	6	**gen_tridiag_factor** **gen_tridiag_solve** **gen_tridiag_solve_factored** **gen_pentadiag_factor** **gen_pentadiag_solve** **gen_pentadiag_solve_factored** **block_tridiag_factor** **block_tridiag_solve** **block_tridiag_solve_factored** **block_pentadiag_factor** **block_pentadiag_solve** **block_pentadiag_solve_factored** **deallocate_banded_solve**
Krylov-based iterative solvers	7	**gen_iter_solve_setup** **gen_iter_solve** **deallocate_iter_solve**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Reduction to tridiagonal form and corresponding basis transformation	8	**sym_tred** **sym_to_tridiag** **tridiag_to_sym** **deallocate_sym_tred**
Eigenvalues of real symmetric tridiagonal matrices	8	**sym_tridiag_eigenvalues**
Eigenvectors of real symmetric tridiagonal matrices	8	**sym_tridiag_eigenvectors**
Eigensystem analysis of dense Hermitian matrices	8	**sym_tred_eigensystem**
Generalized eigensystem analysis of dense Hermitian matrices	8	**sym_tred_gen_eigensystem**
Eigensystem analysis using Jacobi rotations	8	**sym_jacobi_eigensystem**
Eigensystem analysis using a *k*-step Lanczos method	8	**sym_lanczos_setup** **sym_lanczos** **deallocate_sym_lanczos_setup**
Eigensystem analysis using a *k*-step Arnoldi method	8	**gen_arnoldi_setup** **gen_arnoldi** **deallocate_gen_arnoldi_setup**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Bidiagonalization and corresponding basis transformation	9	**gen_bidiag** **gen_bidiag_apply_u** **gen_bidiag_apply_u_tra** **bidiag_to_gen_left** **gen_to_bidiag_left** **gen_bidiag_apply_v** **gen_bidiag_apply_v_tra** **bidiag_to_gen_right** **gen_to_bidiag_right** **deallocate_gen_bidiag**
Singular values of bidiagonal matrices	9	**bidiag_svd_singular_values**
Singular vectors of bidiagonal matrices	9	**bidiag_svd_singular_vectors**
Singular value decomposition of dense real matrices	9	**gen_bidiag_singular_system**
Complex-to-complex FFT	10	**fft_setup** **fft** **fft_detailed** **deallocate_fft_setup**
Real-to-complex and complex-to-real FFT	10	**fft_setup** **fft_detailed** **deallocate_fft_setup**
Array conversion utilities for the FFT	10	**real_from_complex** **complex_from_real**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Explicit integration of ODEs (Runge-Kutta)	11	**ode_rkf_setup** **ode_rkf** **deallocate_ode_rkf_setup**
Dense simplex	12	**gen_simplex**
Fast RNG	13	**initialize_fast_rng** **fast_rng** **save_fast_rng_temps** **restore_fast_rng_temps** **fast_rng_state_field** **fast_rng_residue** **reinitialize_fast_rng** **deallocate_fast_rng**
VP RNG	13	**initialize_vp_rng** **vp_rng** **save_vp_rng_temps** **restore_vp_rng_temps** **vp_rng_state_field** **vp_rng_residue** **reinitialize_vp_rng** **deallocate_vp_rng**
Full histogram	14	**histogram**
Range histogram	14	**histogram_range** **histogram_range_cm**
Polyshift	15	**pshift_setup** **pshift_setup_looped** **pshift** **deallocate_pshift_setup**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
All-to-all rotation	15	**all_to_all_setup** **all_to_all** **deallocate_all_to_all_setup**
All-to-all broadcast	15	**all_to_all_broadcast**
All-to-all reduction	15	**all_to_all_reduce**
Matrix transpose	15	**gen_matrix_transpose**
Sparse gather	15	**sparse_util_gather_setup** **sparse_util_gather** **deallocate_gather_setup**
Sparse scatter	15	**sparse_util_scatter_setup** **sparse_util_scatter** **deallocate_scatter_setup**
Sparse vector gather	15	**sparse_util_vec_gather_setup** **sparse_util_vec_gather** **deallocate_vec_gather_setup**
Sparse vector scatter	15	**sparse_util_vec_scatter_setup** **sparse_util_vec_scatter** **deallocate_vec_scatter_setup**
Block gather and scatter utilities	15	**block_gather** **block_scatter**
Mesh partitioning, pointer reordering	15	**generate_dual** **partition_mesh** **reorder_pointers** **renumber_pointers**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Partitioned gather	15	**part_gather_setup** **part_gather** **part_vector_gather** **deallocate_part_gather_setup**
Partitioned scatter	15	**part_scatter_setup** **part_scatter** **part_vector_scatter** **deallocate_part_scatter_setup**
Communication compiler	15	**comm_setup** **comm_get** **comm_send** **comm_send_add** **comm_send_and** **comm_send_max** **comm_send_min** **comm_send_or** **comm_send_xor** **comm_set_option** **deallocate_comm_setup**
Vector move (extract and deposit)	15	**vector_move** **vector_move_utils**
Computation of block cyclic permutations	15	**compute_fe_block_cyclic_perms**
Permutation along an axis	15	**permute_cm_matrix_axis_from_fe**

Copyright © 1994 Thinking Machines Corporation

Table 1 *(Continued)*

Operation	Chapter	Routines
Combination of permutations	15	**combine_fe_perms**
Zeroing routine	15	**zero_elements**
Send-to-NEWS and NEWS-to-send reordering	15	**send_to_news** **news_to_send**

1.2.2 Nodal Library Restrictions

The routines listed below cannot be used with the nodal CMSSL library (the library used when you compile your program in the CM Fortran single-node execution model). Also, if you are using the CM Fortran global/local execution model, do not call these routines from the local program unit. (For information about linking with CMSSL in the single-node and global/local execution models, see Chapter 2.)

save_gen_lu
restore_gen_lu
save_gen_lu_fms
restore_gen_lu_fms
gen_lu_setup_ext
gen_lu_factor_ext
gen_lu_solve_ext
save_gen_qr
restore_gen_qr
save_gen_qr_fms
restore_gen_qr_fms
gen_qr_factor_ext
gen_qr_solve_ext
gen_matrix_mult_ext
save_fast_rng_temps
restore_fast_rng_temps
save_vp_rng_temps
restore_vp_rng_temps

1.2.3 Safety Mechanism

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

Copyright © 1994 Thinking Machines Corporation

1.3 Notes on Terminology and Conventions

1.3.1 Data Types

Throughout this manual, the terms “real” and “complex” refer to both single-precision and double-precision data, unless otherwise noted. For example, an array described as a “complex CM array” can be either single-precision complex or double-precision complex.

1.3.2 Array Axis Descriptions

In array descriptions throughout this manual, row and column axes are distinguished as follows:

- “The axis that counts the rows,” “the row axis,” and *row_axis* refer to *axis 1* in Figure 1.
- “The axis that counts the columns,” “the column axis,” and *col_axis* refer to *axis 2* in Figure 1.

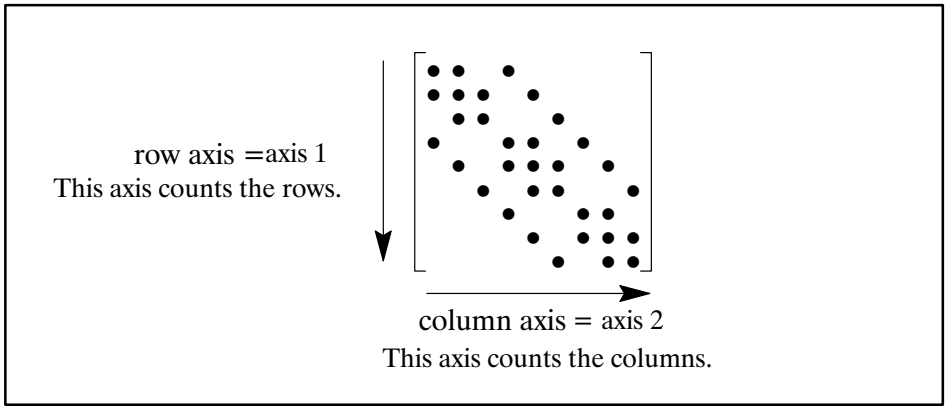

Figure 1. Row and column axes.

1.3.3 Coordinates Are One-Based

With a few exceptions (noted below), the CM Fortran interface to CMSSL adheres to the CM Fortran convention of using 1-based axis numbering and

Copyright © 1994 Thinking Machines Corporation

coordinate numbering in arrays. As a result, unless otherwise noted, variables that take on coordinate values have 1-based values. For example, pointer arrays in the gather, scatter, and partitioning routines, as well as variables containing mesh element numbers or node numbers, all take on 1 as a minimum value. This convention contrasts with that of the C* interface to CMSSL, which uses 0-based axes and coordinates. Coordinate values are therefore one less in the C* interface than in the CM Fortran interface.

The **compute_fe_block_cyclic_perms**, **permute_cm_matrix_axis_from_fe**, and **combine_fe_perms** routines depart from this convention and use 0-based coordinates; details are provided in the man pages for these routines in Chapter 15.

1.3.4 Processing Elements and Subgrids

Some sections of this manual contain implementation and performance information, and use the term *processing element*. The CM system component that serves as the processing element depends on the CM Fortran execution model. Under the vector-units model, a processing element is a vector unit. Under the (SPARC) nodes model, a processing element is a SPARC processing node. The CM Fortran utility **CMF_NUMBER_OF_PROCESSORS** returns the number of processing elements available in the current execution model.

The mapping of array elements to processing elements is performed by the run-time system, and depends on the number of processing elements available to execute the program. You can control this mapping using the detailed axis descriptors of the **CMF$LAYOUT** directive, or using the CM Fortran utility **CMF_ALLOCATE_DETAILED_ARRAY**.

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

Copyright © 1994 Thinking Machines Corporation

An axis is *local* if the array elements along the axis reside within one processing element. An axis is *non-local* or *global* if the array elements along the axis span multiple processing elements.

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

1.4 Data Types Supported

Table 2 shows the data types supported for each CMSSL operation. Within each subroutine call, all CM array arguments must match in data type and precision, unless the argument descriptions indicate otherwise.

1.4.1 Coordinates Must Be Single-Precision Integers

In the current release, variables that take on coordinate values (for example, pointer arrays) must be single-precision (32-bit) integers.

1.4.2 Support for Double-Precision Integers

The routines that perform the CMSSL operations listed below accept double-precision (64-bit) integer arguments when run in the vector-unit execution model. (However, any arguments that take on coordinate values must be 32-bit integers.)

Fast RNG	sparse gather utility
VP RNG	sparse scatter utility
histogram	sparse vector gather utility
range histogram	sparse vector scatter utility
polyshift	block gather and scatter utilities
all-to-all rotation	partitioned gather utility
all-to-all broadcast	partitioned scatter utility
all-to-all reduction	communication compiler
matrix transpose	vector move

Copyright © 1994 Thinking Machines Corporation

Table 2. Data Types Supported by CMSSL for CM Fortran.

Operation	Data Type				
	Integer	Real-4	Real-8	Cmplx-8	Cmplx-16
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				
	Integer	Real-4	Real-8	Cmplx-8	Cmplx-16
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
Array 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				
	Integer	Real-4	Real-8	Cmplx-8	Cmplx-16
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
Send-to-NEWS, NEWS-to-send reordering	x	x	x	x	x

1.5 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. Table 3 shows which operations currently support multiple instances in CM Fortran.

Copyright © 1994 Thinking Machines Corporation

Table 3. CMSSL Support for Multiple Instances in CM Fortran.

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

Copyright © 1994 Thinking Machines Corporation

Table 3 *(Continued)*

Operation	Instances: Single	Instances: Multiple
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, pointer reordering	x	
Partitioned gather utility	x	
Partitioned scatter utility	x	
Communication compiler	x	x
Vector move	x	
Computation of block cyclic permutations	x	x
Permutation along an axis	x	x
Combination of permutations	x	
Zeroing routine	x	
Send-to-NEWS, NEWS-to-send reordering	x	x

1.5.1 Defining Multiple Independent Data Sets

To perform a CMSSL operation on multiple independent data sets concurrently, you must embed the multiple independent instances of each operand or result argument in a CM array. The axes of the array 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.

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 CM array with one data axis (the vertical direction in the figure) and one instance axis; the four source vectors are similarly embedded in another CM array. The source ma-

Copyright © 1994 Thinking Machines Corporation

trices are embedded in a three-dimensional CM array. The instances within each array are labeled 1 through 4.

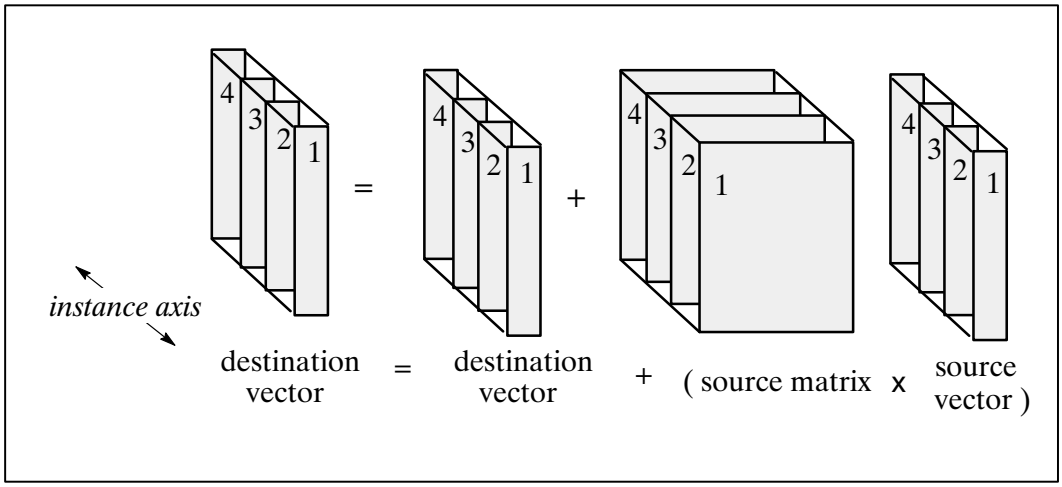

Figure 2. A multiple-instance matrix vector multiplication problem.

The structure defined by the data axes is the object of interest — the logical unit on which the routine operates. This structure is sometimes referred to as a *cell*. The instance axes define the geometry of the larger structure, or *frame*, in which the cells are embedded. The three-dimensional array shown in Figure 2 is a frame containing four two-dimensional cells.

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

1.5.2 Notation Used for CM Arrays and Embedded Matrices

Throughout this book, CM array names are printed in ***bold italics***. If a CM array contains multiple instances of a matrix, the same name is often used for the CM array and each matrix instance it contains. The name is printed in ***bold italics*** to denote the CM array, and in *italics* to denote the embedded matrix. For example, the text might refer to "a CM array ***A*** containing one or more matrices *A*."

Copyright © 1994 Thinking Machines Corporation

1.5.3 Rules for Data Axes and Instance Axes

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

- All operand and result arrays must have the same number of instance axes.
- Counting up through the axes of the arrays, starting with axis 1 and excluding the data axes, corresponding instance axes must occur in the same order in each operand or result array.
- The corresponding instance axes of each operand or result array must have identical extents. In some cases (indicated in the man pages for specific routines), corresponding instance axes must also have identical layout directives.
- The extents of the data axes must be defined so that the operation makes sense. For example, in matrix multiplication, the data axis extents of the operand and result matrices must obey the standard rules for axis extents in matrix multiplication. Specific requirements for data axis extents are provided in the descriptions of individual routines in later chapters.
- Except where explicitly noted, CMSSL supports all combinations of layout directives 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 are local to a processing element. To achieve this state, use the detailed axis descriptors of the CM Fortran **CMF$LAYOUT** directive. Instance axes are typically defined as non-local axes. Some of the descriptions of individual routines in this book contain specific information about optimizing array layouts.

Most CMSSL routines impose few or no restrictions on where the instance axes can occur in an array.

1.5.4 Specifying Single-Instance vs. Multiple-Instance Operations

CMSSL routines that support multiple instances have the same calling sequence for single-instance and multiple-instance operations. The methods you must use to specify single-instance and multiple-instance operations depend on the type of routine you are calling. Specific information is provided

Copyright © 1994 Thinking Machines Corporation

in the man pages included in Chapters 3 through 15. Several examples are discussed below.

Example 1. Matrix Vector Multiplication

When you call the matrix vector multiplication routine, **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 CM array and each matrix argument as a two-dimensional CM array. (Alternatively, you can declare these arguments to have more dimensions, but all instance axes must have extent 1.)
- To perform a multiple-instance operation, embed the multiple instances of each vector argument in a CM array of rank greater than 1, and embed the multiple instances of each matrix argument in a CM array 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. Chapter 3 provides details.

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 array $\boldsymbol{A}$ 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 the array $\boldsymbol{A}$. The data axes, which count the rows and columns of the matrices, can be any two axes of the array $\boldsymbol{A}$; you need not use the first and second axes for this purpose.

Copyright © 1994 Thinking Machines Corporation

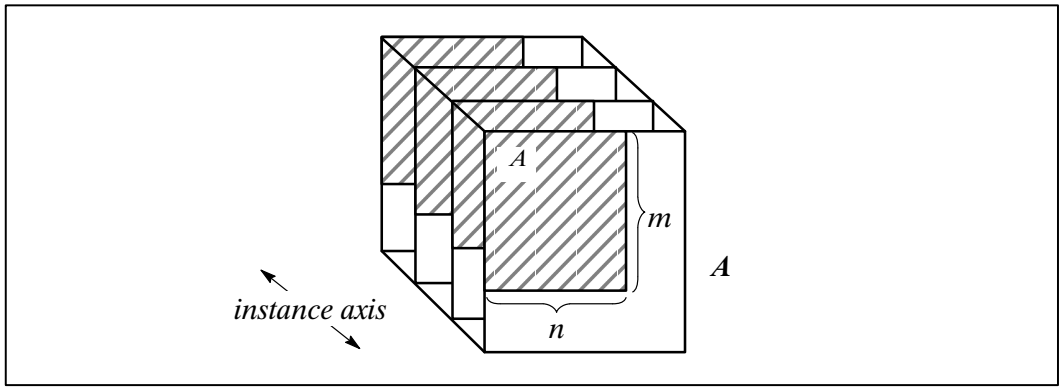

Figure 3. The matrices to be factored have size *m* x *n* and may be contained in larger matrices embedded in *A*.

Figure 4 shows four instances of the linear system $AX = B$. The right-hand-side matrices B are embedded in the array ***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 size $m \times r$.

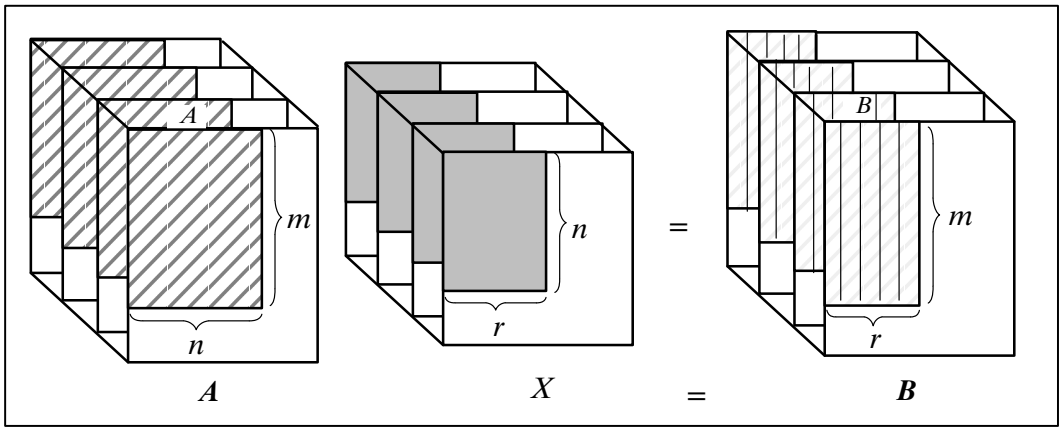

Figure 4. The linear systems *AX = B*

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. For more information about the *QR* routines, refer to Chapter 5.

Copyright © 1994 Thinking Machines Corporation

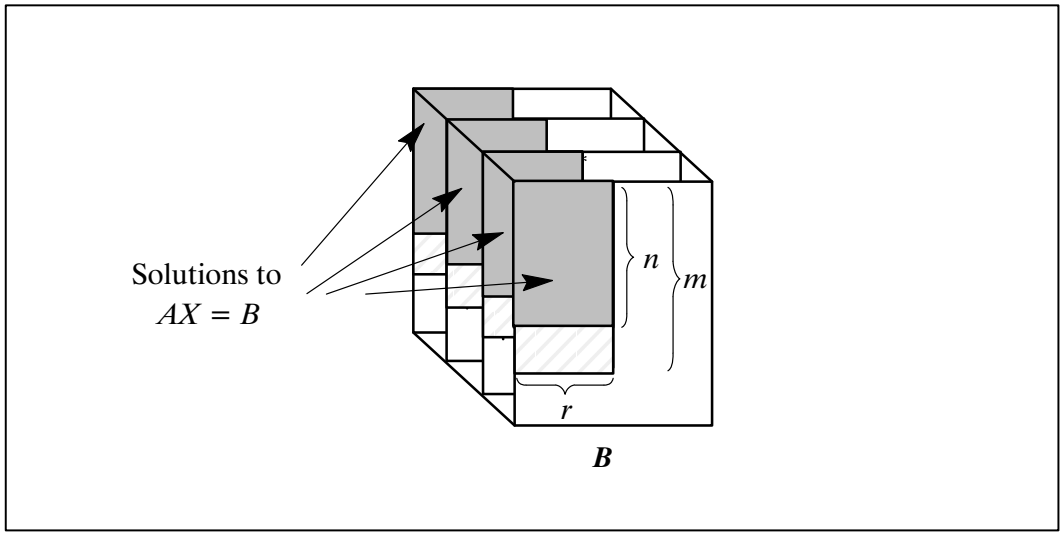

Figure 5. The matrices embedded in the array *B* are partially overwritten with the solutions when the solver routine completes.

Example 3: Fast Fourier Transforms

When you call the Detailed complex-to-complex FFT (CCFFT) routine, you can supply a multidimensional CM array 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 array, 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

For detailed information about multidimensional and multiple-instance FFTs, refer to Chapter 10.

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 array element). Chapter 13 provides details.

Example 5: Polyshift

The polyshift routines perform multidirectional and/or multidimensional sequences of **CSHIFT** and/or **EOSHIFT** operations on a CM array. The axes along which shifts are performed are the data axes; all other axes are the instance axes. Chapter 15 provides details.

Example 6: All-to-All Rotation

The all-to-all rotation routines perform a stepwise rotation along a selected axis of an arbitrary array. Every array element visits every location along the axis. Each step corresponds to a data permutation along the axis, and is typically followed by computations.

In the all-to-all rotation, the axis along which the rotation occurs is the data axis, and all other axes are instance axes. Each one-dimensional cell undergoes an all-to-all rotation. Within a multidimensional array, the multiple instances of the all-to-all rotation have different permutation patterns. For example, if the elements of a two-dimensional array are rotated along the rows, each row may have a different permutation path.

For more information about the all-to-all rotation, refer to Chapter 15.

1.6 Numerical Stability for the Linear Algebra Routines

Some of the descriptions of linear algebra routines in later chapters include information about *numerical stability*. In this book, numerical stability is defined in the standard way: an algorithm is stable if the computed result is the exact solution of a slightly different problem. For example, if A is the input

Copyright © 1994 Thinking Machines Corporation

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

1.7 Numerical Complexity

The following table lists the numbers of floating-point operations performed by some CMSSL routines.

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

Routine	# flops (real operands)	# flops (complex operands)
Vector length = q, number of instances = I:		
gen_inner_product	$2qI$	$8qI$
gen_inner_product_noadd	$(2q-1)I$	$(8q-2)I$
gen_inner_product_addto	$2qI$	$8qI$
gen_inner_product_c1	$2qI$	$8qI$
gen_inner_product_c1_noadd	$(2q-1)I$	$(8q-2)I$
gen_inner_product_c1_addto	$2qI$	$8qI$

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

Copyright © 1994 Thinking Machines Corporation

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Product of axis extents = Q:		
gbl_gen_inner_product	$2Q$	$8Q$
gbl_gen_inner_product_noadd	$2Q-1$	$8Q-2$
gbl_gen_inner_product_addto	$2Q$	$8Q$
gbl_gen_inner_product_c1	$2Q$	$8Q$
gbl_gen_inner_product_c1_noadd	$2Q-1$	$8Q-2$
gbl_gen_inner_product_c1_addto	$2Q$	$8Q$
Vector length = q, number of instances = I:		
gen_2_norm	$[(2q-1)+8]I$*	$(4q-1)+8]I$*
Product of axis extents = Q:		
gbl_gen_2_norm	$(2Q-1)+8$*	$(4Q-1)+8$*
Matrix size = p x q, vector lengths = p and q, number of instances = I:		
gen_outer_product	$2pqI$	$8pqI$
gen_outer_product_noadd	pqI	$6pqI$
gen_outer_product_addto	$2pqI$	$8pqI$
gen_outer_product_c2	$2pqI$	$8pqI$
gen_outer_product_c2_noadd	pqI	$6pqI$
gen_outer_product_c2_addto	$2pqI$	$8pqI$
Matrix size = p x q, vector lengths = p and q, number of instances = I:		
gen_matrix_vector_mult	$2pqI$	$8pqI$
gen_matrix_vector_mult_noadd	$(2pq-p)I$	$(8pq-2p)I$
gen_matrix_vector_mult_addto	$2pqI$	$8pqI$
gen_matrix_vector_mult_c1	$2pqI$	$8pqI$
gen_matrix_vector_mult_c1_noadd	$(2pq-p)I$	$(8pq-2p)I$
gen_matrix_vector_mult_c1_addto	$2pqI$	$8pqI$

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

Copyright © 1994 Thinking Machines Corporation

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Matrix size = p x q, vector lengths = p and q, number of instances = I:		
gen_vector_matrix_mult	$2pqI$	$8pqI$
gen_vector_matrix_mult_noadd	$(2pq-p)I$	$(8pq-2p)I$
gen_vector_matrix_mult_addto	$2pqI$	$8pqI$
gen_vector_matrix_mult_c2	$2pqI$	$8pqI$
gen_vector_matrix_mult_c2_noadd	$(2pq-p)I$	$(8pq-2p)I$
gen_vector_matrix_mult_c2_addto	$2pqI$	$8pqI$
Matrix sizes = p x q and q x r, number of instances = I:		
gen_matrix_mult	$2pqrI$	$8pqrI$
gen_matrix_mult_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_addto	$2pqrI$	$8pqrI$
Matrix sizes = q x p and q x r, number of instances = I:		
gen_matrix_mult_t1	$2pqrI$	$8pqrI$
gen_matrix_mult_t1_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_t1_addto	$2pqrI$	$8pqrI$
gen_matrix_mult_h1	$2pqrI$	$8pqrI$
gen_matrix_mult_h1_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_h1_addto	$2pqrI$	$8pqrI$
Matrix sizes = p x q and r x q, number of instances = I:		
gen_matrix_mult_t2	$2pqrI$	$8pqrI$
gen_matrix_mult_t2_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_t2_addto	$2pqrI$	$8pqrI$
gen_matrix_mult_h2	$2pqrI$	$8pqrI$
gen_matrix_mult_h2_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_h2_addto	$2pqrI$	$8pqrI$

Copyright © 1994 Thinking Machines Corporation

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Matrix sizes = q x p and r x q, number of instances = I:		
gen_matrix_mult_t1_t2	$2pqrI$	$8pqrI$
gen_matrix_mult_t1_t2_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_t1_t2_addto	$2pqrI$	$8pqrI$
Matrix sizes = p x q and q x r:		
gen_matrix_mult _ext	$2pqr$	$8pqr$
Sparse matrix in packed vector form has length n:		
sparse_matvec_mult	$2n$	$8n$
sparse_vecmat_mult	$2n$	$8n$
Sparse matrix in packed vector form has length n; matrix has r columns:		
sparse_mat_gen_mat_mult	$2nr$	$8nr$
gen_mat_sparse_mat_mult	$2nr$	$8nr$
Block sparse matrix has p blocks of size m x n:		
block_sparse_matrix_vector_mult	$2mnp$	$8mnp$
vector_block_sparse_matrix_mult	$2mnp$	$8mnp$
block_sparse_mat_gen_mat_mult	$2mnp$	$8mnp$
gen_mat_block_sparse_mat_mult	$2mnp$	$8mnp$
Block size = p x q; product of extents of grid axes = N; number of instances = I:		
grid_sparse_matrix_vector_mult	$2pqNI$	$8pqNI$
vector_grid_sparse_matrix_mult	$2pqNI$	$8pqNI$
Block sizes = p x q, q x r, and p x r; product of extents of grid axes = N; number of instances = I:		
grid_sparse_mat_gen_mat_mult	$2pqrNI$	$8pqrNI$
gen_mat_grid_sparse_mat_mult	$2pqrNI$	$8pqrNI$

Copyright © 1994 Thinking Machines Corporation

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Matrix size = m x n, r = number of right-hand sides, I = number of instances:		
gen_lu_factor	$[m-(n/3)]n^2I$	$4[m-(n/3)]n^2I$
gen_lu_solve	$2rmnI$	$8rmnI$
gen_lu_solve_tra	$2rmnI$	$8rmnI$
gen_lu_apply_l_inv	$r(2m-n)nI$	$4r(2m-n)nI$
gen_lu_apply_u_inv	rn^2I	$4rn^2I$
gen_lu_apply_l_inv_tra	$r(2m-n)nI$	$4r(2m-n)nI$
gen_lu_apply_u_inv_tra	rn^2I	$4rn^2I$
Matrix size = m x n, r = number of right-hand sides, I = number of instances:		
gen_qr_factor	$2[m-(n/3)]n^2I$	$8[m-(n/3)]n^2I$
gen_qr_solve	$r(4m-n)nI$	$4r(4m-n)nI$
gen_qr_solve_tra	$r(4m-n)nI$	$4r(4m-n)nI$
gen_qr_apply_q	$2r(2m-n)nI$	$8r(2m-n)nI$
gen_qr_apply_q_tra	$2r(2m-n)nI$	$8r(2m-n)nI$
gen_qr_apply_r_inv	rn^2I	$4rn^2I$
gen_qr_apply_r_inv_tra	rn^2I	$4rn^2I$
Matrix size = n x n, r = number of right-hand sides:		
gen_gj_invert	$2n^3$	$8n^3$
gen_gj_solve	$2/3(n^3+2n^2r)$	$8/3(n^3+8n^2r)$
Matrix size = n x n, r = number of right-hand sides:		
gen_lu_factor_ext	$(2/3)n^3$	$(8/3)n^3$
gen_lu_solve_ext	$2n^2r$	$8n^2r$
Matrix size = m x n, r = number of right-hand sides:		
gen_qr_factor_ext	$2[m-(n/3)]n^2$	$8[m-(n/3)]n^2$
gen_qr_solve_ext	$r(4m-n)n$	$4r(4m-n)n$

Copyright © 1994 Thinking Machines Corporation

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
gen_banded_factor	See **gen_tridiag_factor**	
gen_banded_solve	and related routines, below	

n = number of equations or block equations (=length of diagonal)
I = number of instances; r = number of right-hand sides per system
b = block size = length of axis 1 of a, b, and c in **block_tridiag_factor**
p = number of processing elements spanned by axis *axis*
q = product of numbers of processing elements spanned by the instance axes (pq = the total number of processing elements)

The flop count for each **_solve** routine is the sum of the flop counts for the corresponding **_factor** and **_solve_factored** routines.

gen_tridiag_factor*		
CMSSL_pipeline_ge	$8nI$	$26nI$
CMSSL_pge_piv[_val]	$12nI$	$41nI$
CMSSL_substr_cr	$I(14n+14p\log p)$	$I(54n+54p\log p)$
CMSSL_substr_pge	$I(14n+8p)$	$I(54n+26p)$
CMSSL_substr_transp	$I(14n+7p)$	$I(54n+24p$
CMSSL_substr_bcr	$14nI+14\max(pq\log p, (p-1)I)$ (real)	
	$54nI+54\max(pq\log p, (p-1)I)$ (complex)	
gen_tridiag_solve_factored*		
CMSSL_pipeline_ge	$6nIr$	$22nIr$
CMSSL_pge_piv[_val]	$7nIr$	$30nIr$
CMSSL_substr_cr	$Ir(9n+4p\log p)$	$Ir(38n+16p\log p)$
CMSSL_substr_pge	$Ir(9n+5p)$	$Ir(38n+22p)$
CMSSL_substr_transp	$Ir(9n+5p)$	$Ir(38n+22p)$
CMSSL_substr_bcr	$9nIr+9\max(pq\log p, (p-1)Ir)$ (real)	
	$38nIr+38\max(pq\log p, (p-1)Ir$ (complex)	

* Whenever the equation axis (axis *axis*) is local to a processing element, the flop count is equal to that of **CMSSL_pipeline_ge**. Furthermore, some flop counts involving p and q are valid only for problems that "fit the machine" (problems that do not require garbage masks).

Copyright © 1994 Thinking Machines Corporation

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
block_tridiag_factor		
CMSSL_pipleline_ge	$(14/3)nIb^3$	$(56/3)nIb^3$
CMSSL_substr_cr	$Ib^3(14n+14p\log p)$	$Ib^3(56n+56p\log p)$
CMSSL_substr_pge	$Ib^3(14n+[14/3]p)$	$Ib^3(56n+[56/3]p)$
CMSSL_substr_bcr	$b^3(14nI+14\max(pq\log p, (p-1)I))$ (real)	
	$b^3(56nI+56\max(pq\log p, (p-1)I))$ (complex)	
block_tridiag_solve_factored		
CMSSL_pipeline_ge	$6nIrb^2$	$24nIrb^2$
CMSSL_substr_cr	$Irb^2(10n+4p\log p)$	$Irb^2(40n+16p\log p)$
CMSSL_substr_pge	$Irb^2(10n+6p)$	$Irb^2(40n+24p)$
CMSSL_substr_bcr	$b^2(10nIr+10\max(pq\log p, (p-1)Ir))$ (real)	
	$b^2(40nIr+40\max(pq\log p, (p-1)Ir))$ (complex)	

Matrix size = n; number of vectors to be transformed = r:

sym_tred	$(4/3)n^3$
sym_to_tridiag	$2n^2r$
tridiag_to_sym	$2n^2r$

Matrix size = m x n, r = number of right-hand sides, I = number of instances:

gen_bidiag	$4n^2(m-n/3)I$
gen_bidiag_apply_u	$(2nrhs)(2m-n)nI$
gen_bidiag_apply_u_tra	$(2nrhs)(2m-n)nI$
bidiag_to_gen_left	$(2nrhs)(2m-n)nI$
gen_to_bidiag_left	$(2nrhs)(2m-n)nI$
gen_bidiag_apply_v	$(2nrhs)n^2I$
gen_bidiag_apply_v_tra	$(2nrhs)n^2I$
bidiag_to_gen_right	$(2nrhs)n^2I$
gen_to_bidiag_right	$(2nrhs)n^2I$

[*] Whenever the equation axis (axis *axis*) is local to a processing element, the flop count is equal to that of **CMSSL_pipeline_ge**. Furthermore, some flop counts involving p and q are valid only for problems that "fit the machine" (problems that do not require garbage masks).

Copyright © 1994 Thinking Machines Corporation

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
N = length of active axis; I = product of other axis lengths:		
fft	$5N\log N$	
fft_detailed	$5IN\log N$	
m = number of rows, n = number of columns:		
gen_simplex	approx. $2mn$ flops/iteration	

1.8 CM Fortran Performance Enhancements with CMSSL

The following CM Fortran intrinsic and utilities yield better performance when the program is linked with CMSSL:

MATMUL
CMF_RANDOM
CMF_RANDOM_LONG_S_INTEGER
CMF_ORDER
CMF_SORT
CMF_RANK

For example, when CMSSL is linked in, a call to **CMF_random** generates a call to **fast_rng**; a call to **CMF_randomize**(*seed*) generates a call to **initialize_fast_rng** with the same seed value and default parameters for *table_lag*, *short_lag*, and *width*. (The CMSSL random number generators are described in Chapter 13.)

Copyright © 1994 Thinking Machines Corporation

NOTE

Since **CMF_RANDOMIZE** generates a call to **initialize_fast_rng** when CMSSL is linked in, if you call **initialize_fast_rng** explicitly after calling **CMF_RANDOMIZE** (for example, to change the CMSSL Fast RNG parameters), you will receive **initialize_fast_rng** return code −1, which indicates that a prior initialization was overwritten. This code is informational only and does not indicate an error.

Copyright © 1994 Thinking Machines Corporation