Chapter 6

Linear Solvers for Banded Systems

This chapter describes the CM Fortran interface to the CMSSL banded linear system solver routines. The banded system routines factor and solve tridiagonal, pentadiagonal, block tridiagonal, and block pentadiagonal systems of equations. They solve multiple systems of equations, each with one or more right-hand sides, for both real and complex data types. A choice of algorithms is offered.

The multiple-instance capability of the banded system routines in CMSSL is particularly useful in connection with Fourier Analysis Cyclic Reduction, or Alternating Direction Methods. You can specify the axis along which the systems are to be solved. No data reordering or transposition is necessary for the solution of systems along any axis.

On the CM-5, CMSSL includes two sets of banded system routines that offer nearly the same functionality:

- A "unified" set of routines. This set includes one factorization routine and one solver routine that work on all four banded system types (tridiagonal, pentadiagonal, block tridiagonal, and block pentadiagonal). Section 6.1 describes these routines.

- A set that includes three routines (a factorization routine, a solver routine, and a routine that both factors and solves) for each of the four banded system types. These routines are included in the library for compatibility with the CM-200, and are described in Section 6.2.

The two sets of banded system routines use the same array arguments. Only the ordering of some of the arrays in the calling sequence differs. For example, the "unified" routines list the arrays representing diagonals in order from lowermost to uppermost (***a***, ***b***, ***c***, ***d***, ***e***) while the other routines list them from uppermost to lowermost (***e***, ***d***, ***c***, ***b***, ***a***). In addition, the "unified" rou-

Copyright © 1994 Thinking Machines Corporation

tines allow you to supply a pivot value — a feature not included in the other routines.

Detailed descriptions of the banded system routines, including calling sequences, argument definitions, and usage information, are provided in the man pages in this chapter. Section 6.3 lists references.

Copyright © 1994 Thinking Machines Corporation

6.1 Banded System Factorization and Solver Routines (Unified)

This section describes the "unified" banded system factorization and solver routines. The following topics are covered:

- the routines and their functions
- algorithms used
- how to set up your data

6.1.1 The Routines and Their Functions

The unified banded system routines are listed below.

gen_banded_factor Given tridiagonal or block tridiagonal matrices A (represented by three arrays), or pentadiagonal, or block pentadiagonal matrices A (represented by five arrays), this routine performs the factorization $A = LU$ for each matrix, 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.

gen_banded_solve Given the factors computed by **gen_tridiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each B with the solution.

deallocate_banded This routine deallocates the memory required by the factorization and solver routines.

Copyright © 1994 Thinking Machines Corporation

6.1.2 Algorithms Used

When calling the banded system routines, you must specify the algorithm to be used. The following algorithms are available:

CMSSL_pipeline_ge Pipelined Gaussian elimination.

CMSSL_pge_piv Pipelined Gaussian elimination with pairwise pivoting. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns an error code.

CMSSL_pge_piv_val Pipelined Gaussian elimination with pairwise pivoting; replace zero pivots with a supplied value. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns an error code.

CMSSL_substr_cr Substructuring with cyclic reduction.

CMSSL_substr_bcr Substructuring with balanced cyclic reduction.

CMSSL_substr_pge Substructuring with pipelined Gaussian elimination.

CMSSL_substr_transp Substructuring with transpose. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns an error code.

The last four algorithms listed above involve a “divide and conquer” scheme based on substructuring, and differ in the technique used to solve the reduced system of equations. Performance is strongly influenced by the data layout.

Copyright © 1994 Thinking Machines Corporation

NOTE

If the axis along which the diagonal elements or blocks lie (axis *vector_axis* in the argument list) is local, the routine *always* uses Gaussian elimination (with pivoting, if you selected pivoting and have a tridiagonal system).

The algorithm descriptions that follow apply to tridiagonal systems. Block tridiagonal algorithms are the obvious extensions of the elementwise ones; pentadiagonal solvers are based on the block tridiagonal solvers and use the same algorithms.

Pipelined Gaussian Elimination

When multiple banded systems are to be solved, pipelining can be used to balance the computation load. Suppose I systems of N equations each are to be solved on p processing elements, where N/p coefficients of each system appear on each processing element. Figure 21 illustrates the pipelining for the case of $p = 4$ and $I = 6$. Each square box represents N/p rows of one of the I systems in the local memory of the processing element. The vertical axis is time, and a given box is shaded in the figure at time steps when the processing element is actively working on that part of the system.

Copyright © 1994 Thinking Machines Corporation

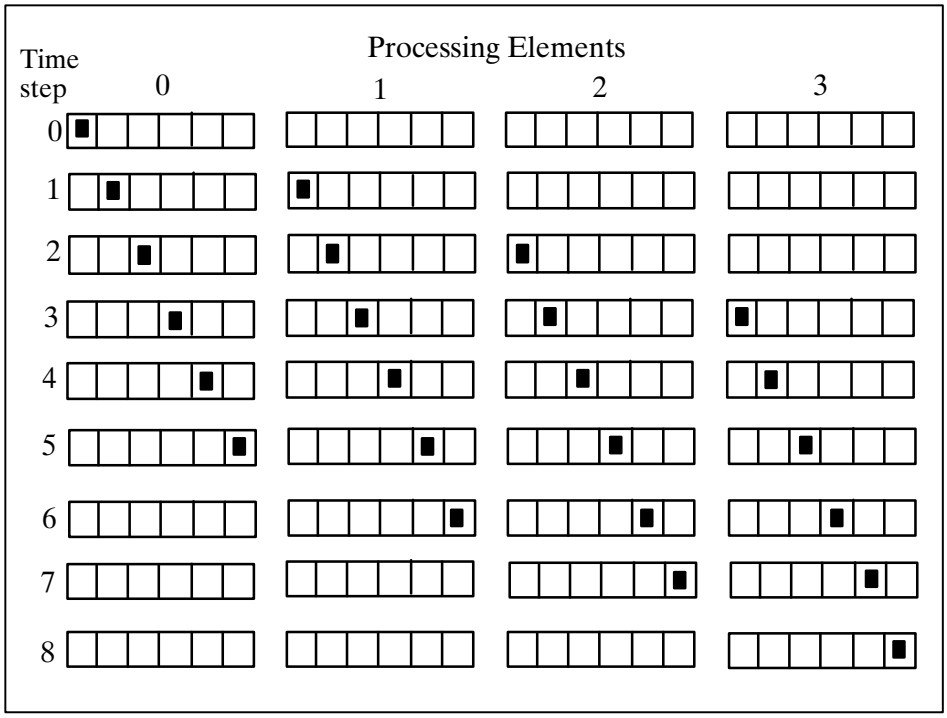

Figure 21. Pipelined Gaussian elimination.

With respect to a given system of N equations, the factorization work proceeds down the diagonal in p time steps, where each time step involves eliminating the subdiagonal element of N/p rows. Since the system is tridiagonal, the Gaussian elimination can be organized so that only one element must be communicated between adjacent processing elements between time steps. With respect to a given processing element, successive time steps repeat Gaussian elimination operations on the same N/p rows of different systems. The factorization algorithm takes $I+p-1$ arithmetic time steps, interleaved with $I+p-2$ shifting communications. The solve phase requires two similar pipelines, one to do the forward elimination and one for the back substitution.

As Figure 21 illustrates, some of the processing elements are idle during the startup and shutdown of the pipeline. In the extreme case where $I=1$, the algorithm is completely sequential. In general, the performance cost for pipelining is proportional to $p-1$, so if I is much larger than p, a good load balance is achieved.

Copyright © 1994 Thinking Machines Corporation

The CMSSL implementation of pipelined Gaussian elimination is more complex than the figure illustrates. To permit more efficient vectorization, the unit of pipelining is enlarged. Groups of g systems are treated together at each step of the pipeline, making the number of time steps per pipeline $I/g + p - 1$. A larger value of g decreases the overheads due to short vector lengths, but increases the penalty for the pipeline startup and shutdown. The optimal value of g depends on p, I, and the block size in a complicated way. The *group_size* input argument allows you to either select g or ask CMSSL to select it for you.

Pipelined Gaussian Elimination with Pairwise Pivoting

Pairwise pivoting refers to the exchange of a pair of adjacent rows whenever that exchange results in a larger divisor for use in eliminating the subdiagonal element. The pivoting makes the pipelining slower and more complex, and causes "fill" above the superdiagonal. The factor phase of the algorithm stores the "fill" in a dynamically-allocated internal buffer with size equal to that of one diagonal. In addition, information about which rows were exchanged must be stored between the factor and solve phases.

Substructuring

When N equations of a tridiagonal system are spread over p processing elements, with N/p equations in each processing element, straightforward Gaussian elimination has only one processor active at a time. One strategy for balancing the load is to let each processing element do Gaussian elimination locally in a way that reduces the total number of equations to p, one on each processing element. Called *substructuring*, this technique involves a forward elimination that zeroes out the subdiagonal, followed by a backward elimination that zeroes out the superdiagonal. Both eliminations produce "fill" equal in size to one diagonal. Substructuring must then be followed by some other technique for solving the (now completely non-local) tridiagonal system of size p. Roughly speaking, substructuring requires twice as many floating-point operations as straightforward Gaussian elimination without pivoting. See references 6 and 4 listed in Section 6.3 for more details.

CMSSL offers four choices for solving the reduced system after substructuring: transpose with local Gaussian elimination, pipelined Gaussian elimination, cyclic reduction, and balanced cyclic reduction.

Copyright © 1994 Thinking Machines Corporation

Cyclic Reduction

A tridiagonal system of irreducible linear equations $Ax = y$, where A is of dimension $N = 2^n - 1$, can be presented in matrix vector form as

$$\begin{bmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & c_3 & & & \\ & & \ddots & \vdots & \ddots & & \\ & & & & \ddots & \ddots & \\ & & & & & a_N & b_N \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \\ \\ x_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \\ \\ y_N \end{bmatrix}$$

Odd-even cyclic reduction consists of a reduction phase succeeded by a back-substitution phase. Using subscripts for equation numbers and superscripts to denote reduction and back-substitution steps, cyclic reduction is defined by the following set of equations:

Reduction

$$a_i^j = e_i \, a_{i-2^{j-1}}^{j-1}$$

$$c_i^j = f_i \, c_{i+2^{j-1}}^{j-1}$$

$$b_i^j = b_i^{j-1} + e_i \, c_{i-2^{j-1}}^{j-1} + f_i \, a_{i+2^{j-1}}^{j-1}$$

$$y_i^j = y_i^{j-1} + e_i \, y_{i-2^{j-1}}^{j-1} + f_i \, y_{i+2^{j-1}}^{j-1}$$

$$e_i = -\frac{a_i^{j-1}}{b_{i-2^{j-1}}^{j-1}}$$

$$f_i = -\frac{c_i^{j-1}}{b_{i+2^{j-1}}^{j-1}}$$

where $i = 2^j, 2 \times 2^j, 3 \times 2^j, \ldots, 2^n - 2^j$, for reduction steps $j = 1, 2, \ldots, n-1$.

Copyright © 1994 Thinking Machines Corporation

The initial conditions are

$$a_i^0 = a_i \qquad b_i^0 = b_i, \quad c_i^0 = c_i, \text{ and } y_i^0 = y_i.$$

After $n - 1$ reduction steps, only one equation of the following form remains:

$$a_{2^{n-1}}^{n-1} x_0 + b_{2^{n-1}}^{n-1} x_{2^{n-1}} + c_{2^{n-1}}^{n-1} x_{2^n} = y_{2^{n-1}}^{n-1}$$

A correct solution for

$$x_{2^{n-1}}$$

is obtained, with $x_0 = x_{N+1} = 0$. Remaining variables are obtained through back-substitution using the following equations:

Back-Substitution

$$x_{2^{n-1}} = \frac{y_{2^{n-1}}^{n-1}}{b_{2^{n-1}}^{n-1}}$$

$$x_i = \frac{y_i^{j-1} - a_i^{j-1} x_{i-2^{j-1}} - b_i^{j-1} x_{i+2^{j-1}}}{b_i^{j-1}}$$

where $i = 2^{j-1}, 3\text{x}2^{j-1}, 5\text{x}2^{j-1}, \ldots, 2^n - 2^{j-1}$, and $j = n-1, n-2, \ldots, 1$.

The algorithm described above is appropriate for serial computers. If it were executed on an *N*-processor parallel computer, half the processors would be idle at reduction step 1, three-fourths idle at reduction step 2, seven-eighths at step 3, and so on. CMSSL offers two variants of the above algorithm that have better load balance. The algorithm we refer to as *cyclic reduction* performs the eliminations in the reduction phase on *all* the equations instead of on an exponentially decreasing number in each step. The end result is *N* equations each in one unknown, and the back-substitution phase is unnecessary. Hockney and Jesshope (reference 5 in Section 6.3) refer to this algorithm as PARACR.

Copyright © 1994 Thinking Machines Corporation

The other variant of serial cyclic reduction offered by CMSSL is called *balanced cyclic reduction*. It is useful only if multiple instances or multiple right-hand sides are available. In each reduction step, half the remaining instances are shifted to the processors that would otherwise become inactive at that step. Thus, assuming at least N problem instances are available, all the processors have work to do in each reduction step. The total operation count is asymptotically a factor of n less than that of the PARACR algorithm, but since balanced cyclic reduction requires the back-substitution phase, PARACR can still be faster when there are only a few instances per processor.

Hints for Choosing an Algorithm

Performance is best for a given array when the axis along which the blocks or elements lie is local. When this axis is not local, consider the following guidelines when choosing an algorithm:

- **Number of independent systems to be solved.** If you have only a single system to solve, choose substructuring with cyclic reduction or substructuring with transpose. As the number of instances per processing element increases, substructuring with balanced cyclic reduction begins to give the best performance. For elementwise tridiagonal systems on a 32-node CM-5, the balanced cyclic reduction algorithm becomes better than the PARACR algorithm for 32 or more instances per processing element.

- **Numerical stability.** Pipelined Gaussian elimination with pairwise pivoting is the most stable algorithm. Numerical experiments have suggested that pipelined Gaussian elimination (even without pivoting) is more stable than cyclic reduction.

- **Memory usage.** All the algorithms use some temporary memory beyond that required to store the system. Pipelined Gaussian elimination uses the least, followed by pipelined gaussian elimination with pivoting, followed by the substructuring algorithms. Substructuring with cyclic reduction uses the most temporary memory.

The following performance hints also apply:

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- When using any of the substructuring algorithms, ensure that multiple instances have the smallest possible stride in memory. You can sometimes control the stride to advantage by using the CM Fortran **-noaxisreorder** switch.
- If you decide to choose the group size for pipelined Gaussian elimination, rather than letting CMSSL choose it for you, try to ensure that the group size divides the number of instances per processing element evenly; otherwise a heavy performance penalty is imposed. As a secondary consideration, it is also beneficial to choose a group size that is a power of 2.

NOTE

If you are working with an elementwise tridiagonal or pentadiagonal system that is either real single-precision or complex, is either single- or multiple-instance, and has one right-hand side, and if axis *vector_axis* is local to a processing element, you will probably achieve better performance by writing the operation in CM Fortran than by using the CMSSL banded system solver routines.

Numerical Stability

Odd-even cyclic reduction is stable for diagonally dominant or positive definite systems. For poorly conditioned systems, the algorithm may be unstable. The algorithm can be stabilized (see reference 1 in Section 6.3), but the current implementation does not include a stablization scheme.

Gaussian elimination is numerically more stable than odd-even cyclic reduction. For an analysis of the stability of Gaussian elimination and odd-even cyclic reduction, see (for instance) references 1, 3, and 8.

Copyright © 1994 Thinking Machines Corporation

6.1.3 How to Set Up Your Data

Tridiagonal and Pentadiagonal Systems

When you factor and solve an elementwise tridiagonal or pentadiagonal system, you must represent each coefficient matrix in the form of three vectors (for tridiagonal systems) or five vectors (for pentadiagonal systems). You must also supply an integer, *vector_axis*, that identifies the axis of each of these three or five vectors along which the matrix elements lie (that is, the non-instance axis).

In addition, when you call the solve routine, you must supply the argument ***B***, the CM array that contains the right-hand-side vectors *B* and is overwritten with the solution *X*.

The detailed requirements for these arrays (and the other required arguments) are provided in the man page at the end of this section. Illustrations and examples are provided below.

For tridiagonal systems, you must supply three CM arrays, ***c***, ***b***, and ***a***, containing the upper, main, and lower diagonal elements, respectively.

To solve a single system, specify the ***c***, ***b***, and ***a*** array arguments as vectors. Figure 22 shows the simplest case: solving a single system with a single right-hand side. Within matrix *A,* the vectors ***c, b***, and ***a*** are shown holding the principal and off-diagonal values. The array ***B*** is shown as two vectors: the right-hand-side vector *B* and the solution vector *X*. Notice that although they represent shorter diagonals, the vectors ***c*** and ***a*** are of the same length as ***b***. The first element of ***a*** and the last element of ***c*** are set to zero during execution.

Copyright © 1994 Thinking Machines Corporation

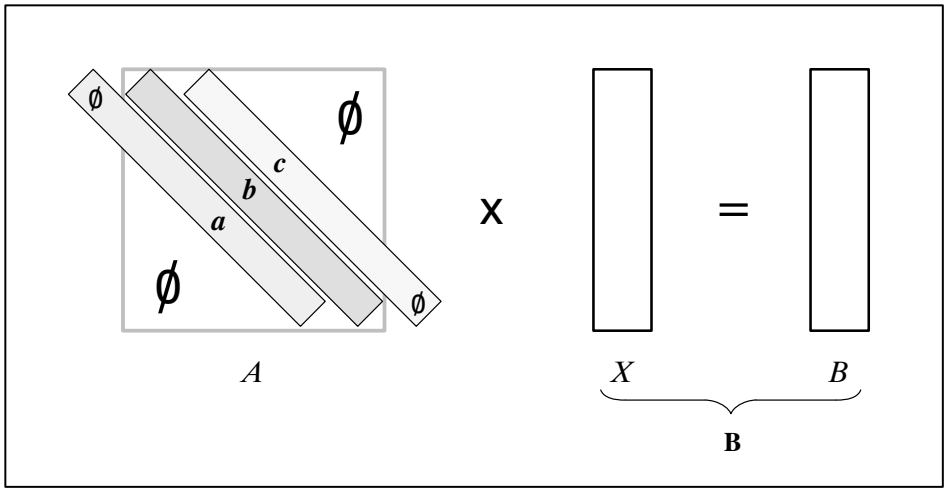

Figure 22. A single tridiagonal system with a single right-hand side.

In matrix notation, the single-system, single-solution case can be represented as shown in Figure 23.

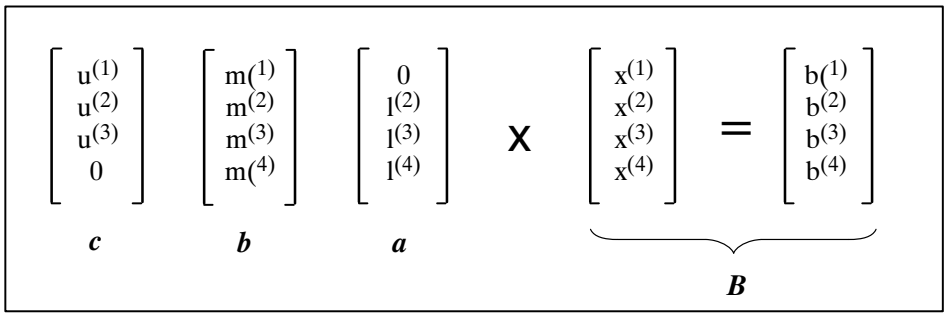

Figure 23. Matrix notation of single tridiagonal system with single right-hand side.

To solve for multiple right-hand sides, specify the ***B*** argument with a local dimension equal to the number of right-hand sides. Figure 24 shows a single tridiagonal system with *nrhs* right-hand sides. Note that the multiple right-hand-side vectors, $b^{(1)} \ldots b^{(nrhs)}$, and their associated solution vectors, $x^{(1)} \ldots x^{(nrhs)}$, are laid out along a local dimension of length *nrhs*.

Copyright © 1994 Thinking Machines Corporation

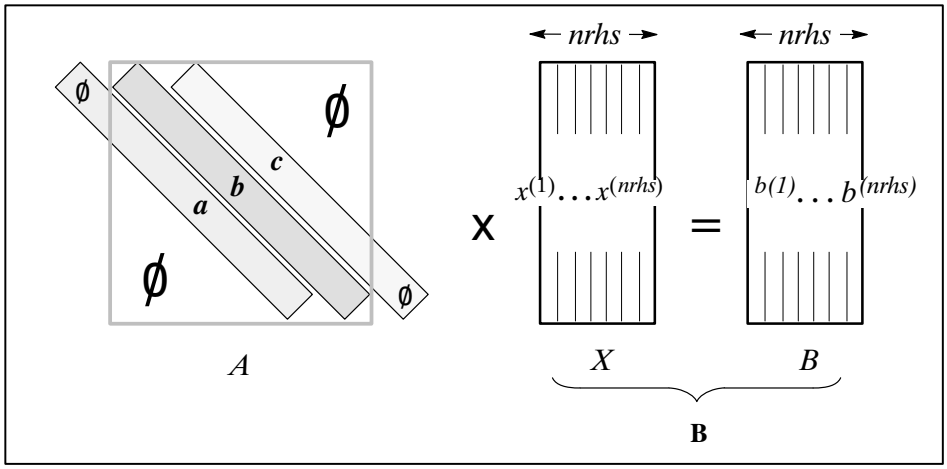

Figure 24. Single tridiagonal system with multiple right-hand sides and solutions.

To solve multiple systems in parallel, specify ***c***, ***b***, and ***a*** with at least 2 dimensions (one data axis and one instance axis) each. The data axis (specified as *vector_axis* in the argument list) represents the coefficients of each system. The instance axis specifies how many systems are represented.

Figure 25 shows multiple concurrent systems, each with a single right-hand side. The n instances of the matrix A are represented by the n sets of tridiagonal values in ***c***, ***b***, and ***a***. Similarly, B consists of the set $[b_1 \ldots b_n]$ of n right-hand-side vectors, and X consists of the set $[x_1 \ldots x_n]$ of n solutions. In this case, there is only one right-hand-side vector for each system; each is overwritten by the one solution vector for that system.

Copyright © 1994 Thinking Machines Corporation

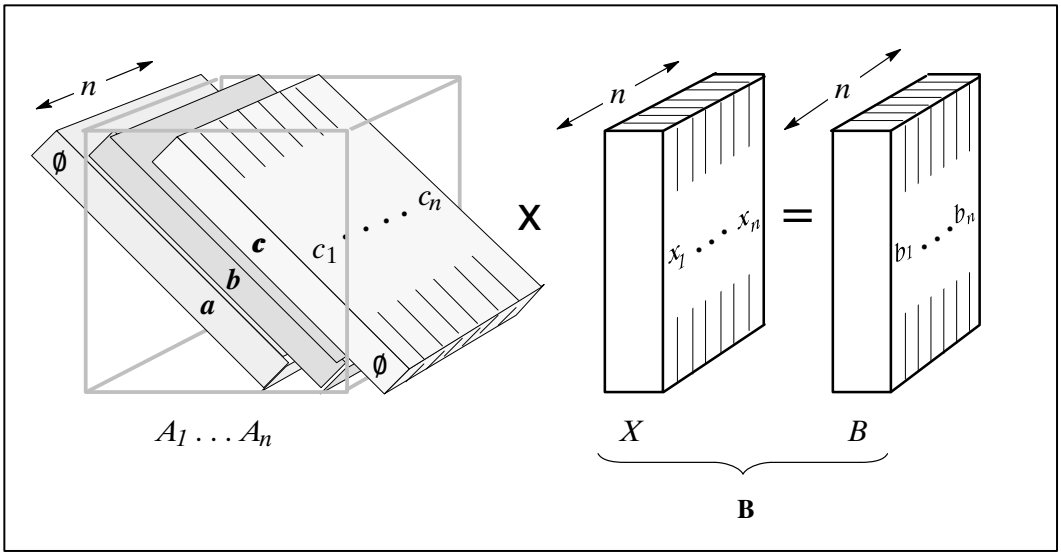

Figure 25. Multiple tridiagonal systems with single right-hand side for each system.

In matrix notation, the multiple-system, single-solution case can be represented as shown in Figure 26.

$$\begin{bmatrix} u_1^{(1)} & u_2^{(1)} & u_3^{(1)} \\ u_1^{(2)} & u_2^{(2)} & u_3^{(2)} \\ u_1^{(3)} & u_2^{(3)} & u_3^{(3)} \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} m_1^{(1)} & m_2^{(1)} & m_3^{(1)} \\ m_1^{(2)} & m_2^{(2)} & m_3^{(2)} \\ m_1^{(3)} & m_2^{(3)} & m_3^{(3)} \\ m_1^{(4)} & m_2^{(4)} & m_3^{(4)} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ l_1^{(2)} & l_2^{(2)} & l_3^{(2)} \\ l_1^{(3)} & l_2^{(3)} & l_3^{(3)} \\ l_1^{(4)} & l_2^{(4)} & l_3^{(4)} \end{bmatrix}$$

c **b** **a**

$$\mathbf{X} \quad \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} \\ x_1^{(3)} & x_2^{(3)} & x_3^{(3)} \\ x_1^{(4)} & x_2^{(4)} & x_3^{(4)} \end{bmatrix} = \begin{bmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \\ b_1^{(2)} & b_2^{(2)} & b_3^{(2)} \\ b_1^{(3)} & b_2^{(3)} & b_3^{(3)} \\ b_1^{(4)} & b_2^{(4)} & b_3^{(4)} \end{bmatrix}$$

X B

Figure 26. Matrix notation of multiple tridiagonal systems with one right-hand side each.

Copyright © 1994 Thinking Machines Corporation

Figure 27 shows *n* systems, each with *nrhs* right-hand sides. Remember that the axis of length *nrhs* must be local.

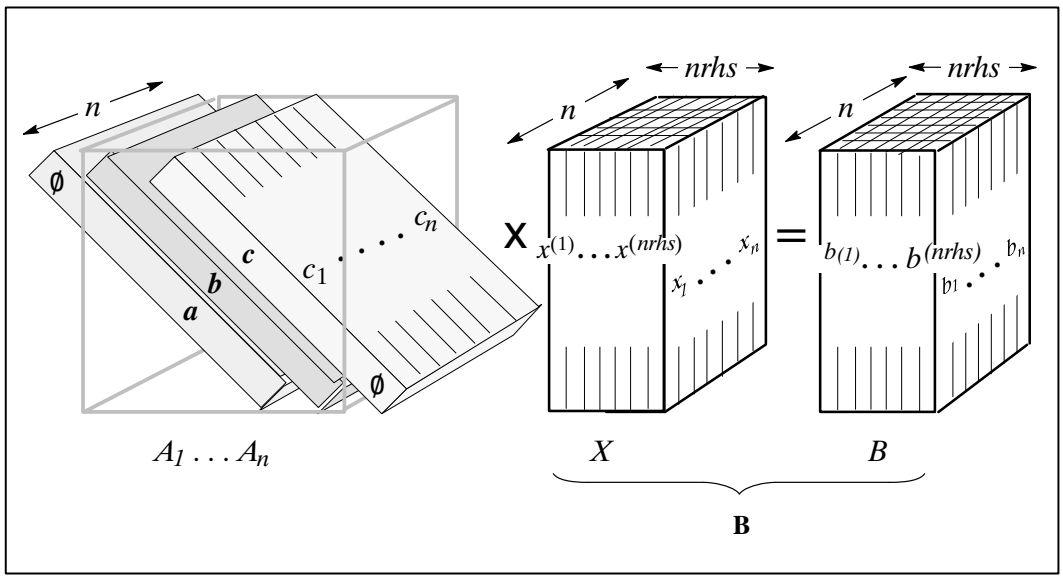

Figure 27. Multiple tridiagonal systems with multiple right-hand sides for each system.

Pentadiagonal systems are represented in the same way as tridiagonal systems, except that you must supply five CM arrays, ***e***, ***d***, ***c***, ***b***, and ***a***, containing the elements of the five diagonals of the coefficient matrices, as shown in Figure 28.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

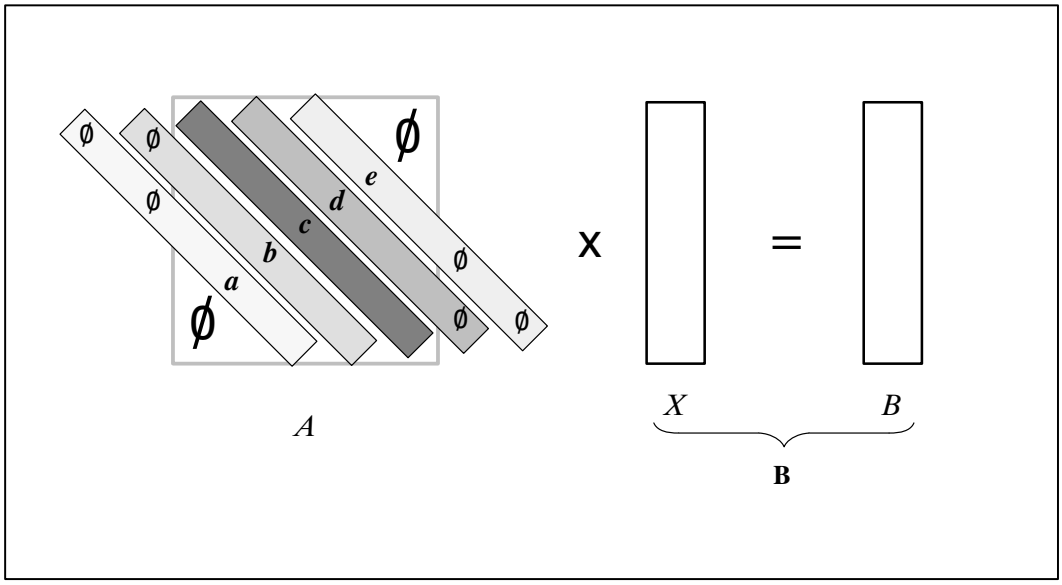

Figure 28. A single pentadiagonal system with a single right-hand side.

Block Tridiagonal and Block Pentadiagonal Systems

When you factor and solve a block tridiagonal or block pentadiagonal system, the routines assume that each matrix A is represented by three arrays (for block tridiagonal systems) or five arrays (for block pentadiagonal systems). More specifically,

- For block tridiagonal systems, you must supply three CM arrays, ***c***, ***b***, and ***a***, containing the square blocks of the coefficient matrices.
- For block pentadiagonal systems, you must supply five CM arrays, ***e***, ***d***, ***c***, ***b***, and ***a***, containing the square blocks of the coefficient matrices.

The detailed requirements for these arrays (and the other required arguments) are provided in the man page at the end of this section.

Figure 29 shows a block tridiagonal system with one instance. In the equation $AX = B$, each $n \times n$ block of A is multiplied by a vector of length n within X to produce a vector of length n within B.

Copyright © 1994 Thinking Machines Corporation

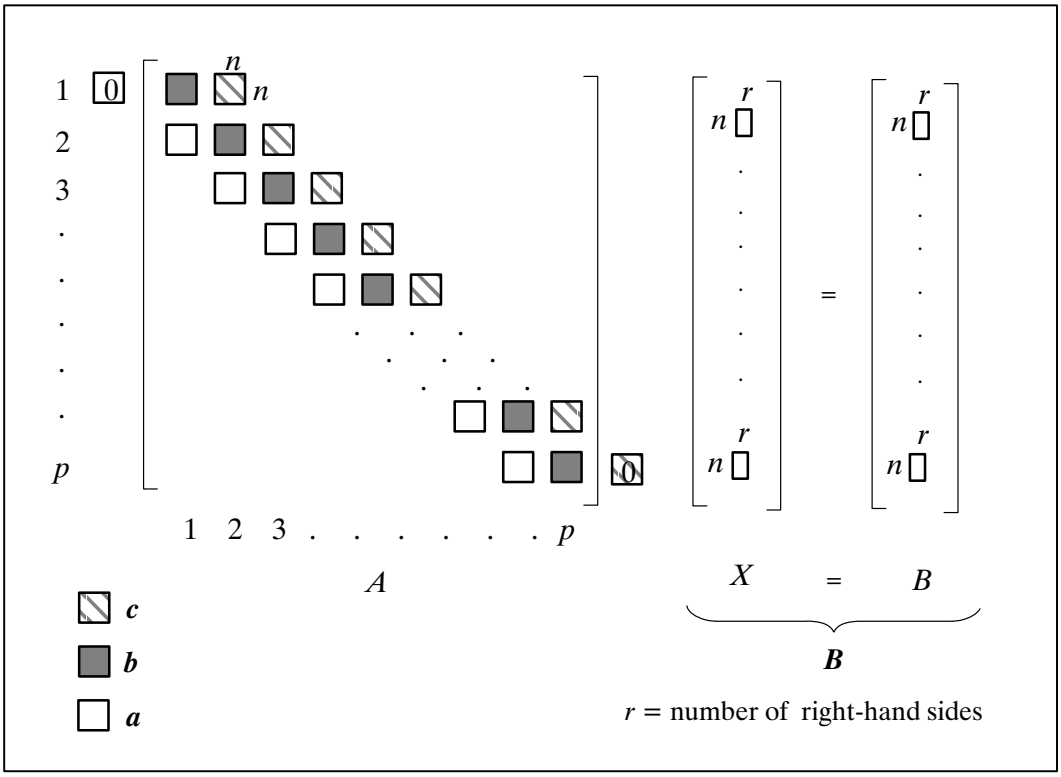

Figure 29. Single-instance block tridiagonal system.

Block pentadiagonal systems are represented in exactly the same manner, except that instead of three arrays of blocks (***c***, ***b***, and ***a***), there are five (***a*** through ***e***).

6.1.4 Need for Interface Blocks

If you supply an array section, rather than an entire array, for ***B***, ***a***, ***b***, ***c***, ***d***, or ***e***, you must use an interface block to ensure that the subsection axis corresponding to any array axis that is required to be local, is also local. An example is provided below. For information about interface blocks and about passing array sections, refer to the CM Fortran documentation set.

Copyright © 1994 Thinking Machines Corporation

In this example, the user application declares the input diagonals as follows:

```
       real , array (nblk,nblk,neqn) :: a,b,c,d,e
cmf$   layout a(:serial,:serial,:news)
cmf$   layout b(:serial,:serial,:news)
cmf$   layout c(:serial,:serial,:news)
cmf$   layout d(:serial,:serial,:news)
cmf$   layout e(:serial,:serial,:news)
       real , array (nblk,neqn) :: rhs
cmf$   layout rhs(:serial, :news)
```

In this case, the interface blocks are as follows:

```
       interface
       subroutine gen_banded_factor( sys_type,
      $      a, b, c, d, e,
      $      axis, work, type, pivot_value, group_size,
      $     ierror )
       implicit none

       real a(:,:,:) ,b(:,:,:) ,c(:,:,:) ,d(:,:,:), e(:,:,:)
cmf$   layout a(:serial,:serial,:news)
cmf$   layout b(:serial,:serial,:news)
cmf$   layout c(:serial,:serial,:news)
cmf$   layout d(:serial,:serial,:news)
cmf$   layout e(:serial,:serial,:news)

       integer sys_type, axis, work, type
       real  pivot_value
       integer  group_size, ierror
       end interface


       interface
       subroutine gen_banded_solve(sys_type,
      $      rhs, a, b, c, d, e,
      $      axis, work, ierror )
       implicit none

       real a(:,:,:) ,b(:,:,:), c(:,:,:) ,d(:,:,:), e(:,:,:)
       real rhs(:,:)
```


Copyright © 1994 Thinking Machines Corporation

```
cmf$  layout a(:serial,:serial,:news)
cmf$  layout b(:serial,:serial,:news)
cmf$  layout c(:serial,:serial,:news)
cmf$  layout d(:serial,:serial,:news)
cmf$  layout e(:serial,:serial,:news)
cmf$  layout rhs(:serial,         :news)

      integer sys_type, axis, work, ierror
      end interface
```

The calls using array subsections are as follows, where $1 \leq s, t \leq neqn$:

```
      call gen_banded_factor(3, a(:,:,s:t), b(:,:,s:t),
     $     c(:,:,s:t), d(:,:,s:t), e(:,:,s:t), 3, work,
     $     1, 0, 0, ier)
      call gen_banded_solve(3, rhs(:,s:t), a(:,:,s:t),
     $     b(:,:,s:t), c(:,:,s:t), d(:,:,s:t), e(:,:,s:t),
     $     3, work, ier)
```

Copyright © 1994 Thinking Machines Corporation

Banded System Factorization and Solver Routines (Unified)

Given one or more instances of a tridiagonal, block tridiagonal, pentadiagonal, or block pentadiagonal matrix A, the routines described below factor A, solve the system(s) $AX = B$ (where B is an array containing one or more right-hand sides), and overwrite B with the solution. Pairwise pivoting is available for tridiagonal systems. A and B must have the same data type (real or complex) and precision (single or double). In the syntax below, the solution X and the right-hand-side B are both represented by the array ***B***.

SYNTAX

sys_type = 0 (tridiagonal system):

gen_banded_factor (*sys_type, **a**, **b**, **c**, vector_axis, work, type, pivot_value, ier*)
gen_banded_solve (*sys_type, **B**, **a**, **b**, **c**, vector_axis, work, ier*)
deallocate_banded (*work*)

sys_type = 1 (block tridiagonal system):

gen_banded_factor (*sys_type, **a**, **b**, **c**, vector_axis, work, type, pivot_value, group_size, ier*)
gen_banded_solve (*sys_type, **B**, **a**, **b**, **c**, vector_axis, work, ier*)
deallocate_banded (*work*)

sys_type = 2 (pentadiagonal system):

gen_banded_factor (*sys_type, **a**, **b**, **c**, **d**, **e**, vector_axis, work, type, pivot_value, ier*)
gen_banded_solve (*sys_type, **B**, **a**, **b**, **c**, **d**, **e**, vector_axis, work, ier*)
deallocate_banded (*work*)

sys_type = 3 (block pentadiagonal system):

gen_banded_factor (*sys_type, **a**, **b**, **c**, **d**, **e**, vector_axis, work, type, pivot_value, group_size, ier*)
gen_banded_solve (*sys_type, **B**, **a**, **b**, **c**, **d**, **e**, vector_axis, work, ier*)
deallocate_banded (*work*)

Copyright © 1994 Thinking Machines Corporation

ARGUMENTS

sys_type Scalar integer variable indicating the type of system being solved. Must have one of the following values:

0	Tridiagonal
1	Block tridiagonal
2	Pentadiagonal
3	Block pentadiagonal

B CM array that contains one or more right-hand sides. Must have the same data type and precision as ***a***, ***b***, and ***c*** (for a tridiagonal system) or ***a***, ***b***, ***c***, ***d***, and ***e*** (for a pentadiagonal system). The solve routine overwrites this array with the solution.

For *sys_type* = 0 or 2 (tridiagonal or pentadiagonal systems), you may set up ***B*** in either of the following ways:

- ***B*** may have rank one greater than that of ***c***, ***b***, and ***a*** (for a tridiagonal system) or ***e***, ***d***, ***c***, ***b***, and ***a*** (for a pentadiagonal system). The first axis counts the right-hand sides and must be local. Axis *vector_axis* counts the elements within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.
- If there is only one right-hand side per instance, you may omit the first axis. That is, ***B*** may have the same rank as ***e***, ***d***, ***c***, ***b***, and ***a***. Axis *vector_axis* counts the elements within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

For *sys_type* = 1 or 3 (block tridiagonal or block pentadiagonal systems), you may set up ***B*** in either of the following ways:

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- ***B*** may have rank equal to that of ***e***, ***d***, ***c***, ***b***, and ***a***. The first axis counts the elements within the subvectors to be multiplied by the blocks of A in the equation $Ax = B$. This axis must be local, and has extent n if the blocks are n x n. The second axis counts the right-hand sides, and must also be local. Axis *vector_axis* counts the subvectors within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

- If there is only one right-hand side per instance, you may omit the second axis. That is, ***B*** may have rank one less than that of ***e***, ***d***, ***c***, ***b***, and ***a***. The first axis counts the elements within the subvectors, must be local, and has extent n if the blocks are n x n. Axis *vector_axis* counts the subvectors within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

a, b, c In tridiagonal or block tridiagonal systems: Real or complex CM arrays containing the elements or blocks that form the lower (***a***), main (***b***), and upper (***c***) diagonals of all instances of A. These three arrays must be distinct and must have the same shape, layout, data type, and precision. The first element or block along axis *vector_axis* axis of ***a***, and the last element or block along axis *vector_axis* of ***c***, are set to zero during execution.

For *sys_type* = 0 (tridiagonal systems), each array must have rank greater than or equal to 1. The *vector_axis* argument identifies the axis along which the diagonal elements lie.

For *sys_type* = 1 (block tridiagonal systems), each array must have rank greater than or equal to 3. The first two axes count the rows and columns of the blocks of A. These axes must be local, and have the same extent since the blocks must be square. The remaining axes include the instance axes (if any) and the axis along which the blocks lie. These remaining axes must occur in the same order in all three arrays. The *vector_axis* argument identifies the axis along which the blocks lie.

Copyright © 1994 Thinking Machines Corporation

a, b, c, d, e In pentadiagonal or block pentadiagonal systems: Real or complex CM arrays containing the elements or blocks that form the five diagonals of all instances of A. The array ***a*** represents the lowermost diagonal; the array ***e*** represents the uppermost diagonal. These five arrays must be distinct, but must all have the same shape, layout, data type, and precision. The first element of ***b***, the last element of ***d***, the first two elements of ***a***, and the last two elements of ***e*** are set to zero during execution.

For *sys_type* = 2 (pentadiagonal systems), ***a, b, c, d,*** and ***e*** must have rank greater than or equal to 1. The *vector_axis* argument identifies the axis along which the diagonal elements lie.

For *sys_type* = 3 (block pentadiagonal systems), ***a, b, c, d,*** and ***e*** must have rank greater than or equal to 3. The first two axes count the rows and columns of the blocks of A. These axes must be local, and have the same extent since the blocks must be square. The remaining axes include the instance axes (if any) and the axis along which the blocks lie. These remaining axes must occur in the same order in all five arrays. The *vector_axis* argument identifies the axis along which the blocks lie.

vector_axis Scalar integer variable. The axis of ***a, b, c, d,*** and ***e*** along which the diagonal elements or blocks of A lie. The value of *vector_axis* must be at least 1, but less than or equal to the rank of ***a, b, c, d,*** and ***e***. Performance is best if the axis identified by *vector_axis* is local.

work Scalar integer variable. Internal variable. Upon completion of a factor routine, *work* contains information required by the associated solve routine.

type Scalar integer that has one of the symbolic constant values (or equivalent numeric values) listed below. Selects the algorithm.

CMSSL_pipeline_ge (3)
Pipelined Gaussian elimination.

CMSSL_pge_piv (9)
Pipelined Gaussian elimination with pairwise pivoting. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns *ier* = −5.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

CMSSL_pge_piv_val (10)
: Pipelined Gaussian elimination with pairwise pivoting; replace zero pivots with a supplied value. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns *ier* = −5.

CMSSL_substr_cr (1)
: Substructuring with cyclic reduction.

CMSSL_substr_bcr (4)
: Substructuring with balanced cyclic reduction.

CMSSL_substr_pge (2)
: Substructuring with pipelined Gaussian elimination.

CMSSL_substr_transp (5)
: Substructuring with transpose. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns *ier* = −5.

If the axis along which the diagonal elements or blocks lie (axis *vector_axis*) is local, the routine *always* uses Gaussian elimination (with pivoting, if you selected pivoting and have a tridiagonal system).

pivot_value
: Scalar variable of the same data type as the banded system. When *type* = 10, this value replaces any zero pivots the routine encounters. This value is ignored if *type* is not equal to 10.

Copyright © 1994 Thinking Machines Corporation

group_size Scalar integer variable. The number of problem instances per processing element treated together in one step of Gauusian elimination. Ignored unless *type* is **CMSSL_pipeline_ge**, **CMSSL_pge_piv**, **CMSSL_pge_piv_val**, or **CMSSL_substr_pge**. With these methods, specify *group_size* = 0 if you want CMSSL to choose the group size, or specify a group size such that $1 \leq$ *group_size* $\leq$ the number of instances per processing element. A larger group size decreases the overheads due to short vector lengths, but increases the penalty for the pipeline startup and shutdown. The optimal group size depends on the number of processing elements, the number of instances per processing element, and the block size in a complicated way. If you supply a *group_size* > 0, try to ensure that the group size divides the number of instances per processing element evenly; otherwise a heavy performance penalty is imposed. As a secondary consideration, it is also beneficial to choose a *group_size* that is a power of 2.

ier Scalar integer variable. Return code. Set to 0 upon successful return, or to one of the following error codes:

−1	Input arrays have inconsistent ranks.
−2	Axes that should be local are not.
−3	Input arrays have inconsistent data types.
−4	Returned when *sys_type* = 1 or 3. The first two axes of ***a, b, c, d,*** or ***e*** do not have equal extents; or, the ***B*** axis that counts the elements within the subvectors to be multiplied by the blocks of *A* does not have the same extent as the first axis of ***a, b, c, d,*** or ***e***.
−5	Invalid *type*.
−6	Invalid *vector_axis*.
−7	The input arrays are not conformable, or the instance axes of ***B*** do not match the instance axes of the other input arrays in layout.
−8	There is an error in the *work* parameter.
−9	*group_size* is either less than 0 or greater than the number of instances per processing element.

Copyright © 1994 Thinking Machines Corporation

DESCRIPTION

The banded system factorization and solver routines perform the following operations:

gen_banded_factor Given tridiagonal or block tridiagonal matrices A (represented by three arrays), or pentadiagonal, or block pentadiagonal matrices A (represented by five arrays), this routine performs the factorization $A = LU$ for each matrix, 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.

gen_banded_solve Given the factors computed by **gen_tridiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each B with the solution.

deallocate_banded This routine deallocates the memory required by the factorization and solver routines.

Separation of the Factorization and Solution Phases. Separation of the factorization and solution phases allows you to factor one or more instances of a matrix once, and then call the solve routine multiple times, supplying the same factors but different right-hand-side arrays each time — thus avoiding the overhead of repeated factorization of the same matrices.

Upon return from the factor routine, the three or five arrays that contained the matrices A on input contain information required by the solve routine. The contents of these arrays must therefore be preserved between the factor and solve calls.

Memory Allocation and Deallocation. Each time you call the factor routine, a buffer, represented by the *work* argument, is allocated in memory. When you call the solve routine, you must supply the value returned in *work* by the associated previous factor call. The *work* buffer remains allocated until you explicitly deallocate it with **deallocate_banded**. Thus, you can call the factor routine, perform other operations, and later call the solve routine one or more times. You can also factor other sets of matrices, thus creating different *work* buffers, and keep multiple *work* buffers allocated at the same time.

Copyright © 1994 Thinking Machines Corporation

Be sure to call **deallocate_banded** to deallocate buffer space whenever you have finished working with one set of factors. If you call the factor routine repeatedly without deallocating buffer space, you will eventually run out of memory.

NOTES

Private Argument Values. The internal variable *work* is required for communicating information between the factorization and solver phases. The application must not modify the contents of this variable.

Preservation of Argument Values. Upon return from the factor routine, the arrays ***a, b, c, d,*** and ***e*** contain information required by the solve routine. The contents of these arrays must therefore be preserved between the factor and solve calls.

Distinct Variables. No overlapping of variables is allowed in these routines.

Caution. The buffer space associated with *work* depends on the size of the matrix or matrices being factored. Therefore, if you call the factor routine more than once, be sure to call **deallocate_banded** to deallocate the associated buffer space between factorization calls, or use a different *work* variable. Otherwise, a second call with the same *work* variable will allocate different buffer space but represent it with the same *work* value as in the first call, and the buffer space associated with the first call will become inaccessible.

Performance Hints. Performance is best if the axes listed below are local.

- The axis of ***a***, ***b***, ***c***, ***d***, and ***e*** along which the blocks or elements lie (axis *vector_axis*).
- The axis of ***B*** along which the subvectors or elements lie.

If you are working with an elementwise tridiagonal or pentadiagonal system that is either real single-precision or complex, is either single- or multiple-instance, and has one right-hand side, and if axis *vector_axis* is local to a processing element, you will probably achieve better performance by writing the operation in CM Fortran than by using the CMSSL banded system solver routines.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

ADI Applications. The multiple-instance implementation of the banded system solvers is excellent for applications of the alternating-direction implicit method, where a solution along each axis is required.

Need for Interface Blocks. If you supply an array section, rather than an entire array, for ***B***, ***a***, ***b***, ***c***, ***d***, or ***e***, you must use an interface block to ensure that the subsection axis corresponding to any array axis that is required to be local, is also local. For information about interface blocks and about passing array sections, refer to the CM Fortran documentation set.

EXAMPLES

Sample CM Fortran code that uses the banded system factorization and solver routines can be found on-line in the subdirectory

```
tridiag/cmf/
```

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

Copyright © 1994 Thinking Machines Corporation

6.2 Banded System Factorization and Solver Routines

The banded system factorization and solver routines described in this section are provided primarily for compatibility with the CM-200. They include a factorization routine, a solver routine, and a combined factorization and solver routine for each of the banded system types (tridiagonal, pentadiagonal, block tridiagonal, and block pentadiagonal).

These routines support multiple instances and accept either real or complex data. They provide the same algorithms as the unified banded system routines (see Section 6.1.2), with the following exceptions:

- The **gen_tridiag_solve** routine does not allow you to specify an algorithm. It uses substructuring with cyclic reduction (**CMSSL_substr_cr**), unless the axis along which the diagonal elements lie (axis *vector_axis* in the argument list) is local (the recommended layout), in which case standard Gaussian elimination is used.

- The algorithm **CMSSL_pge_piv_val** is not available with these routines; that is, you cannot supply a pivot value.

Like the unified banded system routines, the routines described in this section require an interface block if you supply an array section, rather than an entire array, for the ***B***, ***a***, ***b***, ***c***, ***d***, or ***e*** arguments. Refer to Section 6.1.4 for an example.

The man page that follows provides calling sequences, argument definitions, and usage information. Data is set up in the same way as for the unified banded system routines.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Banded System Factorization and Solver Routines

Given one or more instances of a tridiagonal, block tridiagonal, pentadiagonal, or block pentadiagonal matrix A, the routines described below factor A, solve the system(s) $AX = B$ (where B is an array containing one or more right-hand sides), and overwrite B with the solution. Pairwise pivoting is available for tridiagonal systems. A and B must have the same data type (real or complex) and precision (single or double). In the syntax below, the solution X and the right-hand-side B are both represented by the array ***B***.

SYNTAX

gen_tridiag_factor (***c, b, a,*** *vector_axis, tolerance, work, type, ier*)

gen_tridiag_solve_factored (***B, c, b, a,*** *vector_axis, tolerance, work, type, ier*)

gen_tridiag_solve (***B, c, b, a,*** *vector_axis, tolerance, ier*)

gen_pentadiag_factor (***e, d, c, b, a,*** *vector_axis, tolerance, work, type, ier*)

gen_pentadiag_solve_factored (***B, e, d, c, b, a,*** *vector_axis, tolerance, work, type, ier*)

gen_pentadiag_solve (***B, e, d, c, b, a,*** *vector_axis, tolerance, type, ier*)

block_tridiag_factor (***c, b, a,*** *vector_axis, tolerance, work, type, group_size, ier*)

block_tridiag_solve_factored (***B, c, b, a,*** *vector_axis, tolerance, work, type, ier*)

block_tridiag_solve (***B, c, b, a,*** *vector_axis, tolerance, type, group_size, ier*)

block_pentadiag_factor (***e, d, c, b, a,*** *vector_axis, tolerance, work, type, group_size, ier*)

block_pentadiag_solve_factored (***B, e, d, c, b, a,*** *vector_axis, tolerance, work, type, ier*)

block_pentadiag_solve (***B, e, d, c, b, a,*** *vector_axis, tolerance, type, group_size, ier*)

deallocate_banded_solve (*work*)

ARGUMENTS

In the descriptions below, the following terms are used to refer to the banded system routines: The *factor routines* are those ending with the suffix **_factor**. The *solve routines* are those ending with the suffix **_solve_factored**. The *factor-and-solve rou-*

Copyright © 1994 Thinking Machines Corporation

tines are those ending with the suffix **_solve**. The routines prefixed with **gen_** are referred to as the *elementwise* routines; the routines prefixed with **block_** are the *block* routines.

B CM array that contains one or more right-hand sides. Must have the same data type and precision as ***c***, ***b***, and ***a*** (for a tridiagonal system) or ***e***, ***d***, ***c***, ***b***, and ***a*** (for a pentadiagonal system). The solve and factor-and-solve routines overwrite this array with the solution.

For the elementwise banded system routines, you may set up ***B*** in either of the following ways:

- ***B*** may have rank one greater than that of ***e***, ***d***, ***c***, ***b***, and ***a***. The first axis counts the right-hand sides and must be local. Axis *vector_axis* counts the elements within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

- If there is only one right-hand side per instance, you may omit the first axis. That is, ***B*** may have the same rank as ***e***, ***d***, ***c***, ***b***, and ***a***. Axis *vector_axis* counts the elements within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

For the block banded system routines, you may set up ***B*** in either of the following ways:

- ***B*** may have rank equal to that of ***e***, ***d***, ***c***, ***b***, and ***a***. The first axis counts the elements within the subvectors to be multiplied by the blocks of A in the equation $Ax = B$. This axis must be local, and has extent n if the blocks are n x n. The second axis counts the right-hand sides, and must also be local. Axis *vector_axis* counts the subvectors within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

Copyright © 1994 Thinking Machines Corporation

- If there is only one right-hand side per instance, you may omit the second axis. That is, ***B*** may have rank one less than that of ***e***, ***d***, ***c***, ***b***, and ***a***. The first axis counts the elements within the subvectors, must be local, and has extent *n* if the blocks are *n* x *n*. Axis *vector_axis* counts the subvectors within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

c, b, a In tridiagonal or block tridiagonal systems: Real or complex CM arrays containing the elements or blocks that form the upper (***c***), main (***b***), and lower (***a***) diagonals of all instances of *A*. These three arrays must be distinct and must have the same shape, layout, data type, and precision. The last element or block along axis *vector_axis* of ***c***, and the first element or block along axis *vector_axis* of ***a***, are set to zero during execution.

For the elementwise tridiagonal routine, each array must have rank greater than or equal to 1. The *vector_axis* argument identifies the axis along which the diagonal elements lie.

For the block tridiagonal routine, each array must have rank greater than or equal to 3. The first two axes count the rows and columns of the blocks of *A*. These axes must be local, and have the same extent since the blocks must be square. The remaining axes include the instance axes (if any) and the axis along which the blocks lie. These remaining axes must occur in the same order in all three arrays. The *vector_axis* argument identifies the axis along which the blocks lie.

e, d, c, b, a In pentadiagonal or block pentadiagonal systems: Real or complex CM arrays containing the elements or blocks that form the five diagonals of all instances of *A*. The array ***a*** represents the lowermost diagonal; the array ***e*** represents the uppermost diagonal. These five arrays must be distinct, but must all have the same shape, layout, data type, and precision. The first element of ***b***, the last element of ***d***, the first two elements of ***a***, and the last two elements of ***e*** are set to zero during execution.

For the elementwise pentadiagonal routine, ***e, d, c, b,*** and ***a*** must have rank greater than or equal to 1. The *vector_axis* argument identifies the axis along which the diagonal elements lie.

Copyright © 1994 Thinking Machines Corporation

For the block pentadiagonal routine, ***e, d, c, b,*** and ***a*** must have rank greater than or equal to 3. The first two axes count the rows and columns of the blocks of A. These axes must be local, and have the same extent since the blocks must be square. The remaining axes include the instance axes (if any) and the axis along which the blocks lie. These remaining axes must occur in the same order in all five arrays. The *vector_axis* argument identifies the axis along which the blocks lie.

vector_axis Scalar integer variable. The axis of ***e, d, c, b,*** and ***a*** along which the diagonal elements or blocks of A lie. The value of *vector_axis* must be at least 1, but less than or equal to the rank of ***e, d, c, b,*** and ***a***. Performance is best if the axis identified by *vector_axis* is local.

tolerance Scalar real variable. Ignored on the CM-5.

work Scalar integer variable. Internal variable. Upon completion of a factor routine, *work* contains information required by the associated solve routine.

type Integer that has one of the following symbolic constant values (or the equivalent numeric value):

CMSSL_pipeline_ge (3)
Pipelined Gaussian elimination.

CMSSL_pge_piv (9)
Pipelined Gaussian elimination with pairwise pivoting. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns $ier = -5$.

CMSSL_substr_cr (1)
Substructuring with cyclic reduction.

CMSSL_substr_bcr (4)
Substructuring with balanced cyclic reduction.

CMSSL_substr_pge (2)
Substructuring with pipelined Gaussian elimination.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

CMSSL_substr_transp (5)
: Substructuring with transpose. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns *ier* = −5.

If the axis along which the diagonal elements or blocks lie (axis *vector_axis*) is local, the routines *always* use pipelined Gaussian elimination (with pivoting, if you selected pivoting and have a tridiagonal system).

group_size
: Scalar integer variable. The number of problem instances per processing element treated together in one step of Gauusian elimination. Ignored unless *type* is **CMSSL_pipeline_ge**, **CMSSL_pge_piv**, or **CMSSL_substr_pge**. With these methods, specify *group_size* = 0 if you want CMSSL to choose the group size, or specify a group size such that $1 \leq group_size \leq$ the number of instances per processing element. A larger group size decreases the overheads due to short vector lengths, but increases the penalty for the pipeline startup and shutdown. The optimal group size depends on the number of processing elements, the number of instances per processing element, and the block size in a complicated way. If you supply a *group_size* > 0, try to ensure that the group size divides the number of instances per processing element evenly; otherwise a heavy performance penalty is imposed. As a secondary consideration, it is also beneficial to choose a *group_size* that is a power of 2.

ier
: Scalar integer variable. Return code. Set to 0 upon successful return, or to one of the following error codes:

  −1 Input arrays have inconsistent ranks.

  −2 Axes that should be local are not.

  −3 Input arrays have inconsistent data types.

  −4 In one of the block routines, the first two axes of ***e, d, c, b,*** or ***a*** do not have equal extents; or, the ***B*** axis that counts the elements within the subvectors to be multiplied by the blocks of *A* does not have the same extent as the first axis of ***e, d, c, b,*** or ***a***.

  −5 Invalid *type*.

Copyright © 1994 Thinking Machines Corporation

−6 Invalid *vector_axis*.

−7 The input arrays are not conformable, or the instance axes of ***B*** do not match the instance axes of the other input arrays in layout.

−8 There is an error in the *work* parameter.

−9 *group_size* is either less than 0 or greater than the number of instances per processing element.

DESCRIPTION

The banded system factorization and solver routines perform the operations listed below. The factorization routine performs the factorization $A = LU$ for each matrix A, 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.

gen_tridiag_factor Given tridiagonal matrices A (represented by three arrays), this routine factors the matrices.

gen_tridiag_solve_factored Given the factors computed by **gen_tridiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each B with the solution.

gen_tridiag_solve Given tridiagonal matrices A (represented by three arrays), and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to AX $(=LUX) = B$, and overwrites each B with the solution.

gen_pentadiag_factor Given pentadiagonal matrices A (represented by five arrays), this routine factors the matrices.

gen_pentadiag_solve_factored Given the factors computed by **gen_pentadiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each B with the solution.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

gen_pentadiag_solve	Given pentadiagonal matrices A (represented by five arrays), and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to AX ($=LUX$) $= B$, and overwrites each B with the solution.
block_tridiag_factor	Given block tridiagonal matrices A (represented by three arrays), this routine factors the matrices.
block_tridiag_solve_factored	Given the factors computed by **block_tridiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each B with the solution.
block_tridiag_solve	Given block tridiagonal matrices A (represented by three arrays), and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to AX ($=LUX$) $= B$, and overwrites each B with the solution.
block_pentadiag_factor	Given block pentadiagonal matrices A (represented by five arrays), this routine factors the matrices.
block_pentadiag_solve_factored	Given the factors computed by **block_pentadiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each B with the solution.
block_pentadiag_solve	Given block pentadiagonal matrices A (represented by five arrays), and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to AX ($=LUX$) $= B$, and overwrites each B with the solution.
deallocate_banded_solve	This routine deallocates the memory required by the above factorization and solver routines.

Separation of the Factorization and Solution Phases. Calling a factor routine followed by the associated solve routine is equivalent to calling the associated factor-

Copyright © 1994 Thinking Machines Corporation

and-solve routine. However, separation of the factorization and solution phases allows you to factor one or more instances of a matrix once, and then call the appropriate solve routine multiple times, supplying the same factors but different right-hand-side arrays each time — thus avoiding the overhead of repeated factorization of the same matrices.

Upon return from a factor routine, the three or five arrays that contained the matrices *A* on input contain information required by the corresponding solve routine. The contents of these arrays must therefore be preserved between the factor and solve calls.

Memory Allocation and Deallocation. Each time you call one of the factor routines, a buffer, represented by the *work* argument, is allocated in memory. When you call one of the solve routines, you must supply the value returned in *work* by the associated previous factor call. The *work* buffer remains allocated until you explicitly deallocate it with **deallocate_banded_solve**. Thus, you can call a factor routine, perform other operations, and later call the corresponding solve routine one or more times. You can also factor other sets of matrices, thus creating different *work* buffers, and keep multiple *work* buffers allocated at the same time.

Be sure to call **deallocate_banded_solve** to deallocate buffer space whenever you have finished working with one set of factors. If you call a factor routine repeatedly without deallocating buffer space, you will eventually run out of memory.

NOTES

Private Argument Values. The internal variable *work* is required for communicating information between the factorization and solver phases. The application must not modify the contents of this variable.

Preservation of Argument Values. Upon return from a factor routine, the arrays ***e, d, c, b,*** and ***a*** contain information required by the corresponding solve routine. The contents of these arrays must therefore be preserved between the factor and solve calls.

Distinct Variables. No overlapping of variables is allowed in these routines.

Deallocation. Be sure to call **deallocate_banded_solve** to deallocate buffer space whenever you have finished working with one set of factors. If you call a factor

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

routine repeatedly without deallocating buffer space, you will eventually run out of memory.

Caution. The buffer space associated with *work* depends on the size of the matrix or matrices being factored. Therefore, if you call a factor routine more than once, be sure to call **deallocate_banded_solve** to deallocate the associated buffer space between factorization calls, or use a different *work* variable. Otherwise, a second call with the same *work* variable will allocate different buffer space but represent it with the same *work* value as in the first call, and the buffer space associated with the first call will become inaccessible.

Performance Hints. Performance is best if the axes listed below are local.

- The axis of ***a***, ***b***, ***c***, ***d***, and ***e*** along which the blocks or elements lie (axis *vector_axis*).
- The axis of ***B*** along which the subvectors or elements lie.

If you are working with an elementwise tridiagonal or pentadiagonal system that is either real single-precision or complex, is either single- or multiple-instance, and has one right-hand side, and if axis *vector_axis* is local to a processing element, you will probably achieve better performance by writing the operation in CM Fortran than by using the CMSSL banded system solver routines.

ADI Applications. The multiple-instance implementation of the banded system solvers is excellent for applications of the alternating-direction implicit method, where a solution along each axis is required.

Need for Interface Blocks. If you supply an array section, rather than an entire array, for ***B***, ***a***, ***b***, ***c***, ***d***, or ***e***, you must use an interface block to ensure that the subsection axis corresponding to any array axis that is required to be local, is also local. For information about interface blocks and about passing array sections, refer to the CM Fortran documentation set.

Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the banded system factorization and solver routines can be found on-line in the subdirectory

```
tridiag/cmf/
```

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

Copyright © 1994 Thinking Machines Corporation

6.3 References

For more information about banded system solvers, see the following references:

1. Buneman, O. *A Compact Non-Iterative Poisson Solver*. Stanford University Institute for Plasma Research, Report No. 294, 1969.

2. Buzbee, B. L., G. H. Golub, and C. W. Nielson. On Direct Methods for Solving Poisson's Equations. *SIAM J. Num. Analysis* **7** (1970): 627–56.

3. Golub, G. H., and C. F. Van Loan. *Matrix Computations.* 2d ed. Baltimore: Johns Hopkins University Press, 1989. Pp. 194–220.

4. Ho, C. and S. L. Johnsson. Optimizing Tridiagonal Solvers for Alternating Direction Methods on Boolean Cube Multiprocessors. *SIAM J. Sci. Stat. Comput*. **11**, no. 3 (1990): 563–92.

5. Hockney, R. W., and C. R. Jesshope. *Parallel Computers 2*. Bristol and Philadelphia: Adam Hilger, 1988. Pp. 475–89.

6. Johnsson, S. L. Solving Tridiagonal Systems on Ensemble Architectures. *SIAM J. Sci. Stat. Comp.* **8**, no. 3 (1987): 354–92.

7. Malcom M.A. and J. Palmer. A Fast Method for Solving a Class of Tridiagonal Systems of Linear Equations. *Comm. ACM* **17** (1974): 14–17.

8. Wilkinson, J. H. *The Algebraic Eigenvalue Problem*. Oxford: Clarendon Press, 1965.

Copyright © 1994 Thinking Machines Corporation