

NASA Technical Memorandum 4259

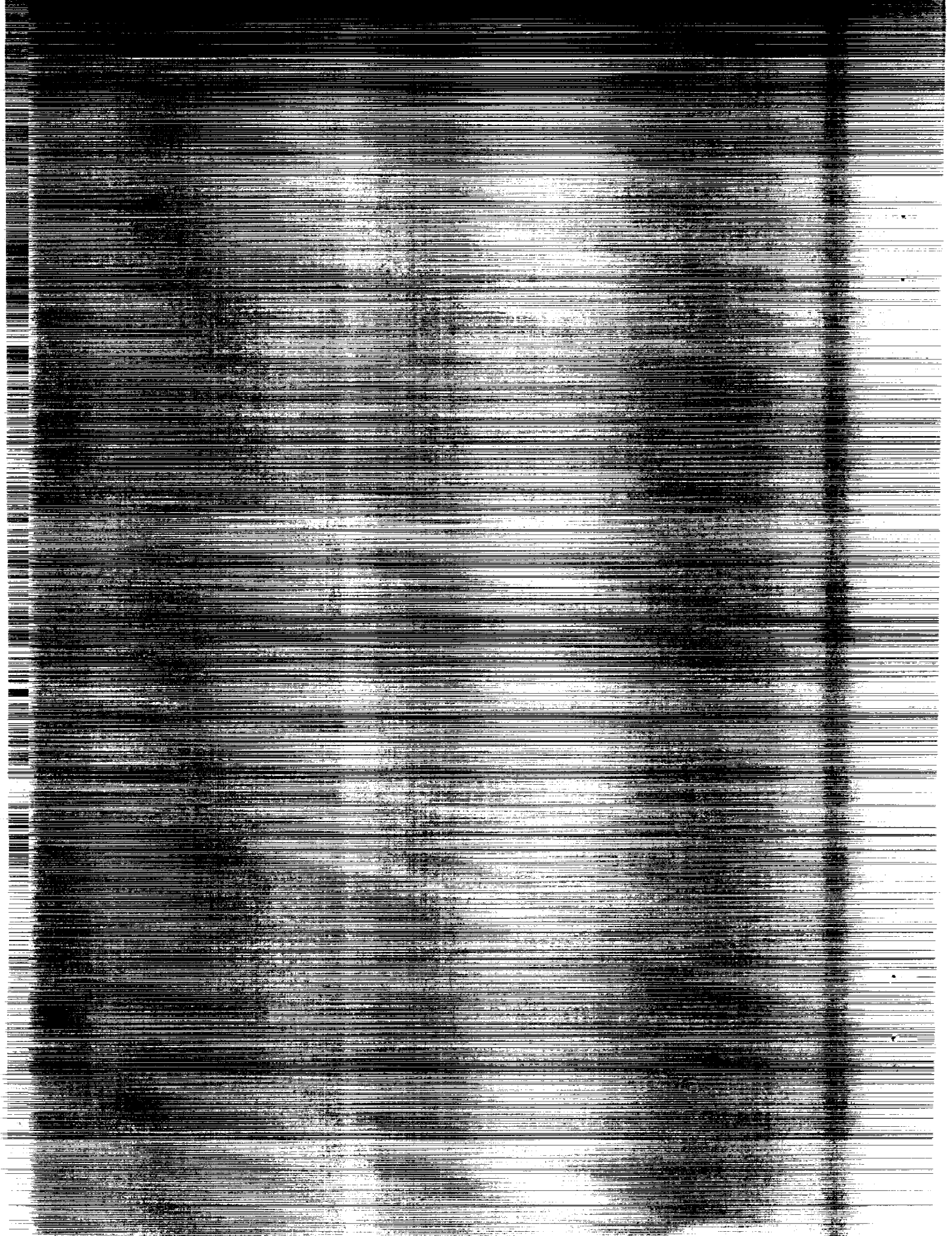
Efficient Multitasking
of Choleski Matrix
Factorization on
CRAY Supercomputers

Andrea L. Overman and Eugene L. Poole

SEPTEMBER 1991

(NASA-TM-4259) EFFICIENT MULTITASKING OF N92-10302
CHOLESKI MATRIX FACTORIZATION ON CRAY
SUPERCOMPUTERS (NASA) 63 p CSCL 09B

Unclas
H1/61 0000169



NASA Technical Memorandum 4259

Efficient Multitasking
of Choleski Matrix
Factorization on
CRAY Supercomputers

Andrea L. Overman
Langley Research Center
Hampton, Virginia

Eugene L. Poole
Analytical Services & Materials, Inc.
Hampton, Virginia



National Aeronautics and
Space Administration

Office of Management

Scientific and Technical
Information Program

1991

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.



Report Documentation Page

1. Report No. NASA TM-4259	2. Government Accession No.	3. Recipient's Catalog No.	
4. Title and Subtitle Efficient Multitasking of Choleski Matrix Factorization on CRAY Supercomputers		5. Report Date September 1991	6. Performing Organization Code
		7. Author(s) Andrea L. Overman and Eugene L. Poole	8. Performing Organization Report No. L-16901
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 505-90-52-02	11. Contract or Grant No.
		12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546-0001	
13. Type of Report and Period Covered Technical Memorandum		14. Sponsoring Agency Code	
		15. Supplementary Notes Andrea L. Overman: Langley Research Center, Hampton, Virginia. Eugene L. Poole: Analytical Services & Materials, Inc., Hampton, Virginia.	
16. Abstract This paper describes a Choleski method used to solve linear systems of equations that arise in large-scale structural analyses. The method uses a novel variable-band storage scheme and is structured to exploit fast local memory caches while minimizing data access delays between main memory and vector registers. Several parallel implementations of this method are described for the CRAY-2 and CRAY Y-MP computers demonstrating the use of microtasking and autotasking directives. A portable parallel language, FORCE, is used for comparison with the microtasked and autotasked implementations. Results are presented comparing the matrix factorization times for three representative structural analysis problems from runs made in both dedicated and multiuser modes on both the CRAY-2 and CRAY Y-MP computers. Both CPU (central processing unit) times and wall clock times are given for the parallel implementations and are compared with single-processor times of the same algorithm. Computation rates over 1 GIGAFLOP (1 billion floating point operations per second) on a four-processor CRAY-2 and over 2 GIGAFLOPS on an eight-processor CRAY Y-MP are demonstrated as measured by wall clock time in a dedicated environment. Reduced wall clock times for the parallel implementations relative to the single-processor implementation of the same Choleski algorithm are also demonstrated for runs made in a multiuser environment.			
17. Key Words (Suggested by Author(s)) Choleski factorization CRAY-2/CRAY Y-MP Parallel processing/vectorization Macrotasking, microtasking, and autotasking FORCE preprocessor		18. Distribution Statement Unclassified—Unlimited Subject Category 60	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 60	22. Price A04

Abstract

This paper describes a Choleski method used to solve linear systems of equations that arise in large-scale structural analyses. The method uses a novel variable-band storage scheme and is structured to exploit fast local memory caches while minimizing data access delays between main memory and vector registers. Several parallel implementations of this method are described for the CRAY-2 and CRAY Y-MP computers demonstrating the use of microtasking and autotasking directives. A portable parallel language, FORCE, is used for comparison with the microtasked and autotasked implementations. Results are presented comparing the matrix factorization times for three representative structural analysis problems from runs made in both dedicated and multiuser modes on both the CRAY-2 and CRAY Y-MP computers. Both CPU (central processing unit) times and wall clock times are given for the parallel implementations and are compared with single-processor times of the same algorithm. Computation rates over 1 GIGAFLOP (1 billion floating point operations per second) on a four-processor CRAY-2 and over 2 GIGAFLOPS on an eight-processor CRAY Y-MP are demonstrated as measured by wall clock time in a dedicated environment. Reduced wall clock times for the parallel implementations relative to the single-processor implementation of the same Choleski algorithm are also demonstrated for runs made in a multiuser environment.

1. Introduction

The computational cost of computerized structural analysis is often dominated by the cost of solving a very large system of algebraic equations associated with a finite element model. This linear system of equations has the form

$$\mathbf{K}\mathbf{u} = \mathbf{f} \quad (1)$$

where \mathbf{K} is the symmetric positive-definite stiffness matrix, \mathbf{f} is the load vector, and \mathbf{u} is the vector of generalized displacements. The linear systems can be as large as several hundred thousand degrees of freedom and often require significant computing resources, in terms of both memory and execution time. The development of algorithms to solve these linear systems on existing and future high-performance supercomputers that have multiple parallel high-speed vector CPU's (central processing units) can significantly reduce the computer time required for structural analysis. Efficient algorithms that exploit both vectorization and parallelization gain the maximum benefit on such supercomputers.

One widely used method for solving equation (1) is the Choleski method (ref. 1). In the Choleski method the stiffness matrix \mathbf{K} is factored into the product of the triangular matrices $\mathbf{L}\mathbf{L}^T$, followed by

a forward-and-backward solution using the triangular systems

$$\mathbf{L}\mathbf{z} = \mathbf{f} \quad (\text{forward}) \quad (2a)$$

$$\mathbf{L}^T\mathbf{u} = \mathbf{z} \quad (\text{backward}) \quad (2b)$$

The factorization of \mathbf{K} , which is by far the most computationally expensive part of the Choleski method, can be carried out in many ways. The particular implementation of the Choleski method that is the most effective for a class of problems depends on both the structure of the linear systems and the architectural features of the computer used for the analysis. Though the structure of \mathbf{K} for many structural analysis applications is initially very sparse, usually 20 to 50 nonzero coefficients in a given row of the matrix, the factored matrix \mathbf{L} contains many more coefficients produced during the factorization step. For many structural analysis problems, a reordering of the nodes in the finite element model can be carried out that minimizes the bandwidth of the factored matrix, thereby significantly reducing matrix storage requirements and the number of computations required. A novel variable-band matrix storage scheme is used with the Choleski implementations described in this paper to reduce the computational cost of the factorization. The reverse Cuthill-McKee

reordering algorithm (ref. 2) is used to minimize the bandwidth of the linear systems.

In this paper a vectorized variable-band Choleski method, denoted as VBAND, is described for single-processor implementation. A parallel Choleski equation solver, based on the VBAND method and denoted as PVBAND, is also presented. Several implementations of the PVBAND method are described demonstrating three different approaches for implementing parallel algorithms on CRAY multiprocessor supercomputers (manufactured by Cray Research, Inc.). The performance of the parallel implementations is demonstrated by comparing factorization times for several large representative structural analysis problems. In a dedicated computing environment, maximum computation rates over 1 GIGAFLOP (1 billion floating point operations per second) on a four-processor CRAY-2 and over 2 GIGAFLOPS on an eight-processor CRAY Y-MP are achieved for the matrix factorization portion of the computerized structural analyses. In a multiuser environment, reduced wall clock times are demonstrated for the parallel methods as compared with the wall clock times for the single-processor VBAND Choleski algorithm.

The outline of this paper is as follows: In section 2, architectural features of the CRAY multiprocessor supercomputers are described. Key hardware features of the CRAY-2 and CRAY Y-MP computers are compared and contrasted, and implementation issues related to exploiting vectorization as well as parallel processing on CRAY supercomputers are discussed. CRAY multitasking alternatives (macrotasking, microtasking, and autotasking) are introduced. In section 3, both single-processor and parallel implementations of the variable-band Choleski method are described. Detailed discussions of the differences between the parallel implementations on the CRAY computers are included. The implementation details provide an aid for understanding the differences in performance across the parallel implementations. In section 4, the role of equation solvers in computerized structural analysis is briefly discussed. Three representative structural analysis problems that are used to compare the parallel implementations of the variable-band Choleski method are described. In section 5, numerical results are presented from multiple runs in both dedicated and multiuser environments, and in section 6, concluding remarks are given. Finally, an appendix is given containing FORTRAN listings for several of the subroutines described in this paper.

2. Parallel-Vector Computer Architectures

The experiments described in this report were performed on a CRAY-2 at the NASA Langley Research Center (LaRC) and a CRAY Y-MP at the NASA Ames Research Center (ARC). This section briefly describes the key architectural features of these computers including a discussion of the major hardware differences of the CRAY-2 and CRAY Y-MP machines that affect algorithm design. In addition, optimization issues related to exploiting both the vector and parallel processing capabilities of CRAY supercomputers are briefly introduced. Differences between macrotasking, microtasking, and autotasking that affect algorithm performance are discussed at the end of the parallel processing subsection.

CRAY Hardware

The CRAY-2 and the CRAY Y-MP are shared-memory multiprocessor computers that have a maximum of four and eight central processing units (CPU's), respectively. Each CPU has multiple vector arithmetic and logic functional units that access very large main memories through eight high-speed vector registers. The maximum computation rate, commonly measured in units known as MFLOPS (millions of floating point operations per second), is attained when both addition and multiplication vector functional units are operating simultaneously, thus producing two results every machine clock period. The CRAY-2 has a clock period of 4.1 nanoseconds (nsec) that results in a maximum theoretical rate of 488 MFLOPS per processor. The CRAY Y-MP used for the experiments in this paper has a clock period of 6.0 nsec that results in a maximum theoretical rate of 333 MFLOPS per processor. These theoretical peak rates are seldom achieved because of several factors, two of which are discussed next.

Memory access delays. One factor that reduces the computation rate is memory access delay. Some delay is always incurred if the operands for a vector addition or multiplication must be transferred from main memory to the vector registers. As soon as the elements are in the vector registers they can be accessed at the maximum computation rate. Both the CRAY Y-MP and the CRAY-2 have 128 million words of main memory arranged in banks. Since memory locations in the same bank cannot be accessed in successive clock periods, contiguous array elements are stored in an interleaved pattern across the banks. This storage pattern allows contiguous array elements to be accessed at successive clock periods after an initial access delay. In general, the

does not have chaining, the memory references for a given set of operands must be completed before computations can begin. This problem is greatly reduced for FORTRAN DO loops containing many addition and multiplication instructions and several different operands since memory references on one set of operands can be carried out while multiplications or additions are taking place on another set of operands. One technique used to increase the ratio of computations to memory references is referred to as *loop unrolling*. In this technique, nested loops containing single-vector instructions, such as SAXPY's or inner products, are combined within a single loop containing several vector computations. This technique allows vectors to remain in the vector registers longer before results are stored back to main memory, thus resulting in a more efficient use of the multiple functional units.

Parallel Processing

In addition to the hardware features of the CRAY-2 and CRAY Y-MP that improve performance on a single CPU, parallel processing is available. The individual CPU's share a large main memory, and synchronization is accomplished using both the shared-memory and special hardware semaphore registers. Because current CRAY computers have small numbers of very powerful processors, the best overall speedups are obtained by parallelizing highly vectorized codes. The speedup from parallel processing is always less than p , where p is the number of processors, whereas the speedup due to vectorization often exceeds 20 or more on a single processor for many codes. Three types of multitasking are available on the CRAY computers and are described below. The PVBAND method was implemented using all three types of multitasking to determine the best approach for both multiuser and dedicated modes. Details of the parallel implementations are given in section 3.

Macrotasking. The earliest form of parallel processing developed for CRAY computers was macrotasking. Macrotasking is intended to be used with large codes running in a dedicated environment. Macrotasked codes perform best when separate parallel tasks, requiring little synchronization, are defined at the subroutine level. Macrotasking is the most difficult form of multitasking used on CRAY machines. Since parallelism must be defined at a higher level, macrotasking typically requires more code changes and data analysis than microtasking and autotasking. Parallelism is defined through the insertion of library subroutines, making the code un-

portable to other machines. (See refs. 3 and 4 for detailed discussions on macrotasking.)

Macrotasking requires extensive task management that is handled by the library scheduler, a collection of library subroutines. Parallel *tasks*, independent portions of work, are initiated through calls to the TSKSTART and are synchronized by *events*, *locks*, or the TSKWAIT subroutine. TSKSTART creates the *logical CPU's*, or *processes*, if they have not been created previously. When a task changes state, the library scheduler places the task in the appropriate library queue. For example, the scheduler periodically checks the *Ready* queue for tasks that can be attached to logical CPU's. Attachment to a logical CPU designates a task as schedulable for execution on a physical CPU. On the other hand, if a task must wait for another task to be completed, the library scheduler disconnects the waiting task from the logical CPU and places it in the *Suspended* queue. In addition, if no new tasks are ready, the logical CPU is "put to sleep," marking that process as unschedulable. That logical CPU must be reactivated later.

A portable parallel language, FORCE, developed for shared-memory parallel computers (refs. 5 and 6) has been implemented on the CRAY computers using macrotasking. FORCE is a machine-independent parallel language that allows programmers to implement parallel algorithms using a fixed syntax. Both parallel constructs (such as barriers and critical regions) and data types (i.e., private and shared variables) can be programmed without detailed knowledge of the implementation of the parallel constructs and data-type definitions on each computer. FORCE uses the UNIX *sed* editor and a set of script files to substitute machine specific code for each of the FORCE statements. In addition to replacing FORCE statements in the user's program, the FORCE preprocessing phase converts the user's main program to a subroutine that is called by a standard driver routine at run time. The driver routine is not seen by the user and serves to create, through calls to TSKSTART, the fixed number of tasks specified by the user at run time. The entire user's program is called from the main driver program by each task. Sequential portions of the code must be placed within FORCE barrier statements or other explicit code statements. At the end of the program, the FORCE driver routine terminates the tasks with the TSKWAIT subroutine.

Microtasking. Microtasking was designed to lower the overhead for CPU synchronization and to allow for more efficient multitasking in a multiuser environment. In a multiuser environment,

maximum transfer rate between main memory and the vector registers occurs when contiguous array elements are accessed from main memory.

There are two key differences in the architectures of the CRAY-2 and CRAY Y-MP that affect memory access delays. The first is the number of paths to memory. On the CRAY-2, there is only a single path between main memory and the vector registers. On the CRAY Y-MP, there are four paths to main memory; two can be used to read from memory, one to write from the registers to main memory, and one for input/output (I/O) operations. The second key difference is the use of local memory caches on the CRAY-2. Each processor on the CRAY-2 has a 16384-word local memory cache. Local memory is accessible only through the vector registers, and only contiguous access of array elements is possible. However, the time delay incurred when accessing array elements stored in local memory is significantly less than that when accessing array elements stored in main memory. As a result, vectors that are used repeatedly in computations may be stored in local memory in order to significantly reduce memory access delays.

Floating point operations and chaining.

Another key factor that affects the computation rate is the type of operations performed to carry out an algorithm. For example, vector SAXPY operations (single precision $a \times X + Y$) are efficient on CRAY computers since they contain both addition and multiplication instructions that can be carried out nearly simultaneously. However, inner-product instructions ($Sum = \sum_{i=1}^n x_i y_i$) are somewhat less efficient since they require a summation of many vector elements into a single scalar value.

An additional feature that improves the performance of floating point operations on the CRAY Y-MP is the chaining of operations and memory accesses. Chaining occurs whenever vector instructions on a set of operands are allowed to overlap, thus reducing the start-up overhead associated with individual vector instructions. For example, on the CRAY Y-MP, a vector add of 2 arrays multiplied by a constant begins by issuing a load instruction for the first 64 elements of the first array that is followed, on the very next clock period, by a second load instruction for the first 64 elements of the second array. As soon as the first operands of the two arrays are in the vector registers, the addition operation begins. The multiplication operation begins as soon as the first result is available from the add functional unit. Finally, the store instruction to transfer the first 64 results back to main memory is issued as soon as the

first results are available from the multiply functional unit. If the length of the arrays is n , then this example requires $O(n)$ clock periods on the CRAY Y-MP. On the CRAY-2, the same example would require $O(3n)$ clock periods to load the two vector operands and store the result using the single path to memory and an additional $O(2n)$ clock periods to complete the addition and multiplication instructions.

Although functional unit chaining, or tailgating, is available on some CRAY-2 machines, most CRAY-2 computers have no chaining capability. However, the CRAY-2 computers can benefit from simultaneous memory access and computations if these operations take place on different sets of operands. Loops that contain several addition and multiplication operations with several independent sets of operands are often able to benefit from simultaneous memory access and computations. In general, loops that execute at high vector speeds on the CRAY-2 also perform well on the CRAY Y-MP, but the converse is not necessarily true.

Vector Optimization

In order to achieve high computation rates on vector computers, algorithms must be designed to exploit the key hardware features discussed above. The challenge to the software developer is to design algorithms that minimize the memory access delays while utilizing the full computing power of multiple vector functional units. Three key vector optimization issues are vector length, vector stride, and the ratio of memory references to computations. Long vector operations (i.e., operations on hundreds or even thousands of array elements in a single loop) are desirable since the cost due to loop overhead and initial memory access is amortized over a larger number of computations. Vector stride refers to the memory access pattern for arrays. Stride one vector operations occur when arrays are accessed contiguously. This occurs, for example, in FORTRAN DO loops where the first dimension of an array reference is incremented by one for each loop iteration. Although much improvement has been made on vector computers in recent years to improve memory access for strides other than one, stride one vector operations are still faster on both the CRAY-2 and CRAY Y-MP. On the CRAY-2, the arrangement of memory banks into quadrants makes even stride accesses particularly bad, especially when the stride is a multiple of four.

The ratio of computations to memory references for vector computations is the third key issue on vector optimization. This ratio is particularly important for the CRAY-2 since there is only one path between memory and vector registers. Because the CRAY-2

microtasked codes dynamically adjust to the available resources, thus making efficient use of processors that are available for short periods of time. Since smaller grained parallelism can be efficiently exploited using microtasking, parallelism is typically defined at the DO loop level. Outermost loops are usually parallelized, whereas inner loops contain vector computations. Microtasking is implemented through the use of compiler directives that appear as comments to other compilers, thus maintaining portability to other computers. Discussions of microtasking can be found in references 3 and 4.

Microtasking is designed for multiuser environments and does not require the task management used by macrotasking. In microtasking, additional processes referred to as *slave* processes are created at the beginning of the program and immediately enter a library subroutine called PARK where they remain until summoned by the *master* process. Meanwhile, the *master* process executes all serial code outside the microtasked subroutines. Upon entering a microtasked subroutine, the master process signals the slaves to enter the parallel region. If a slave process is active, i.e., if the process is executing on a physical CPU, the slave process enters the parallel region. The code within microtasked subroutines is executed by all active processes unless it is contained within control structures. Within a control structure, parallel work is distributed to active processes, and all work within a control structure must be completed before any CPU can proceed past that point. A unique number associated with each control structure as well as the number of active processes within each control structure are stored in special shared variables. These variables are updated by several special library routines that are called from the user's program. The calls to these routines are automatically inserted by the microtasking preprocessor at compilation time. All variables in COMMON blocks, SAVE statements, DATA statements, or in the argument list of a microtasked subroutine are shared variables and can be accessed by all CPU's. All other variables, such as loop indices, are private variables, and each CPU accesses a separate copy.

Autotasking. Autotasking is the most recent parallel programming tool available to programmers on CRAY computers. (See ref. 7 for a complete description of autotasking.) Autotasking expands the features of microtasking by adding the capability to automatically detect some forms of parallelism at the DO loop level and by improving the efficiency of certain parallel constructs. One major difference between microtasking and autotasking is the definition

of parallel regions. In microtasking, the parallel region extends to the subroutine boundaries; whereas in autotasking, multiple parallel regions may exist within a subroutine. The master process executes all code outside the parallel regions and must wait for the slave processes to finish before proceeding past the boundary of a parallel region. Compiler directives, similar to microtasking directives, may be used to specify parallel regions within the code. In this paper, the automatic parallelizing feature of autotasking was not used because it could not detect the parallelism in VBAND. Expanded capabilities for automatic parallelization are planned in future releases of the compiling system.

Multiuser and dedicated environments. The maximum benefit of parallel processing on the CRAY computers to a single user is experienced in dedicated mode. On dedicated systems, parallel work is executed on independent CPU's and all the CPU's are available for a single user. However, users also benefit from parallel processing in multiuser batch modes in many cases. On multiuser systems, multi-tasked programs typically receive more of the available computing resources than a single-processor implementation of the same code. Multiuser computing environments for the CRAY-2 and CRAY Y-MP define multitasking runs in terms of parallel work that may or may not run concurrently on independent CPU's. In macrotasking, a fixed number of tasks are explicitly defined, and the synchronization of these parallel tasks can often result in significant delays on heavily loaded systems. On the other hand, microtasking and autotasking, rather than defining a fixed number of tasks, distribute the work only to the processes that are active at execution time. Those active processes are synchronized at the end of each control structure and at the end of each parallel region. In this paper, both multiuser and dedicated modes of operation are discussed with results presented for both modes.

3. Description of Methods

This section briefly discusses the variable-band data structure and follows with descriptions of the sequential vectorized VBAND Choleski method and the parallel PVBAND Choleski method. The implementations of the sequential VBAND method illustrate the techniques used to obtain computation rates of greater than 200 MFLOPS on the CRAY-2 and CRAY Y-MP. The description of the PVBAND method illustrates the general approach used to parallelize the VBAND method. Details of the three parallel implementations are discussed in a separate subsection. The details presented for the macrotasking,

microtasking, and autotasking implementations are essential to understanding the differences in performance for the parallel implementations. Some results are given for the smallest example problem, the High-Speed Civil Transport (HSCT) aircraft discussed in section 4, to compare several implementations of the VBAND and PVBAND methods.

Variable-Band Data Structure

The variable-band data structure is described in references 8 and 9. The lower triangular part of the symmetric input stiffness matrix \mathbf{K} in equation (1) is stored by columns, beginning with the main diagonal down to the last nonzero entry in each column, including zeros. Two important adjustments are made to the lengths of each column of \mathbf{K} to account for fill-in during factorization and to ensure that columns

in groups corresponding to the level of loop unrolling end in the same row. This data storage scheme is different from traditional skyline or profile Choleski methods that store the *upper* triangular part of \mathbf{K} by columns beginning with the main diagonal and storing all coefficients up to the first nonzero coefficient in each column. The skyline scheme is equivalent to storing the lower triangular part of \mathbf{K} by rows, whereas the variable-band scheme stores the lower triangular part of \mathbf{K} by columns. The advantage of the skyline scheme is that it does not require additional adjustments for fill-in and, in some cases, requires considerably less storage than the variable-band storage scheme. The big disadvantage of the skyline scheme is that the main computations in the associated Choleski methods are inner products that are much slower than the SAXPY operations used by the VBAND method on vector computers.

Variable-band storage:

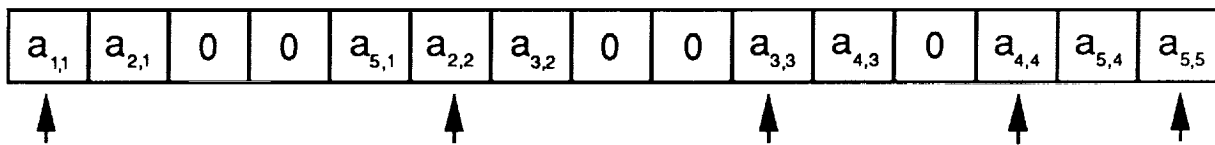
Lower triangular part by columns from diagonal down to last nonzero coefficient in each column, with additional adjustments for fill

$a_{1,1}$	$a_{1,2}$			$a_{1,5}$
$a_{2,1}$	$a_{2,2}$	$a_{2,3}$		0
0	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	0
0	0	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$
$a_{5,1}$	0	0	$a_{5,4}$	$a_{5,5}$

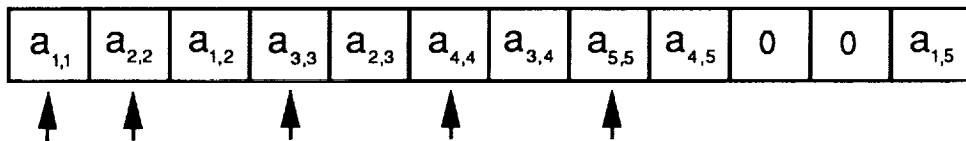
Skyline storage:

Upper triangular part by columns from diagonal up to last nonzero coefficient in each column

(a) Symmetric matrix.



(b) Storage of variable-band scheme in single-dimensional array.



(c) Storage of skyline scheme in single-dimensional array.

Figure 1. Example of variable-band and skyline matrix storage schemes. In parts (b) and (c), arrows indicate pointers to starting coefficient in each column; zeros indicate memory locations allocated for fill elements during factorization.

Variable-band example. An example of the variable-band storage scheme and skyline storage scheme is given in figure 1. The symmetric input matrix has nonzero coefficients in the positions shown in figure 1(a). The position of each coefficient within a single-dimensioned array is shown for each storage scheme in figures 1(b) and 1(c). Auxiliary arrays store the starting location of each diagonal element in the matrix, denoted in figure 1 by arrows. The zeros in figure 1 represent zero coefficients that are stored by both schemes to allow for possible fill-in coefficients. During factorization the coefficients of \mathbf{K} are modified, and many of the zero coefficients are replaced with nonzero values. The factored matrix \mathbf{L} overwrites \mathbf{K} . In the skyline storage scheme, no space is allocated for elements $a_{1,3}$ and $a_{1,4}$ in the first row of the upper triangular part of the input matrix since they lie above the last nonzero coefficient in columns 3 and 4, respectively. However, in the variable-band scheme, the corresponding elements of column 1 of the lower triangular part of the input matrix, $a_{3,1}$ and $a_{4,1}$, lie between nonzero coefficients in column 1 of \mathbf{L} and extra storage is allocated. Additional adjustments are made, if necessary, to the column lengths in both the skyline and variable-band storage schemes if loop unrolling is added to the factorization implementation. In general, loop unrolling requires that groups of columns end in the same row. For example, if loop unrolling to level 3 is used, then columns are arranged in groups of three and extra zero coefficients are allocated as required to ensure that each group of columns ends in the same row. In practice, the adjustment for loop unrolling increases the overall matrix storage requirement very little.

Matrix assembly. The specialized variable-band data structure used for the Choleski methods described in this paper requires some interface with the matrix assembly process when used in an existing finite element code. Matrix assembly routines can be written to directly assemble finite element stiffness matrices into the variable bandwidth form, but this approach may not be desirable in realistic applications. The variable-band storage scheme is required only for factorization. For portions of the analysis computations, which may require matrix-vector multiplications or the addition of two matrices, the variable-band storage scheme would be very inefficient because it would require a great many additional operations. This is also true for very large problems with high bandwidths, where iterative solvers may be used in place of direct Choleski methods. For these reasons, a more practical approach is to assemble the stiffness matrices into a general

sparse matrix format that can be quickly converted to the variable-band format when the variable-band Choleski solver is to be used.

The conversion from a general sparse matrix storage scheme, in which the lower triangular nonzero coefficients are stored by columns, to the variable-band data storage scheme can be carried out efficiently as follows: The lengths of each column in the variable-band matrix are initially determined from the row indices stored for each nonzero coefficient. Then, the row lengths are adjusted to account for fill and loop unrolling as previously described. Next, the row index pointers for each nonzero coefficient are converted to location pointers for each coefficient within the variable-band matrix array. Finally, the variable-band array is initialized to zero and the nonzero coefficients are assigned using the integer location pointers. Most of the operations used to convert from sparse to variable-band data storage schemes vectorize, and the time required to convert from sparse to variable-band storage is small compared with the factorization time. In the appendix, FORTRAN listings for two subroutines used for this conversion are given.

Sequential VBAND Choleski Method

This section describes a basic Choleski factorization method and several modifications that significantly improve the computation rate on vector computers. In the Choleski method a symmetric, positive-definite matrix \mathbf{K} is factored into the product $\mathbf{L}\mathbf{L}^T$. The factorization can be carried out in many ways but, in general, elements of \mathbf{L} are computed from \mathbf{K} a column (or row) at a time beginning with column (or row) 1 of \mathbf{K} and proceeding to the last column (or row). Only the lower part of \mathbf{K} is stored, using the variable-band storage format described above, and \mathbf{L} , which is computed by modifying \mathbf{K} , is stored in the same space as the original matrix \mathbf{K} .

Choleski factorization in ijk forms. Reference 10 describes Choleski factorization for vector computers in terms of the so-called ijk forms. The ijk forms describe Choleski factorizations in terms of the nesting order of three loops that compute the modifications of \mathbf{K} into \mathbf{L} . These modifications are computed in the innermost loop in Choleski factorization implementations and are of the form

$$\mathbf{K}_{i,j} \leftarrow \mathbf{K}_{i,j} - \mathbf{L}_{i,k}\mathbf{L}_{k,j} \quad (3)$$

The notation \leftarrow , used in equation (3) and hereafter, indicates that the variable is updated and overwritten with a new value specified by the right-hand expression. The order of the three letters i, j ,

and k indicates the order of the three loops. For example, traditional skyline or profile Choleski methods are of the ijk form where the inner vectorized loop varies the index k in equation (3) to compute the inner product of rows i and j of \mathbf{L} . The middle loop, denoted by j , determines that row j of \mathbf{L} is used in the inner-product computation with row i of \mathbf{L} to update element $\mathbf{K}_{i,j}$. After the inner-product update of $\mathbf{K}_{i,j}$, element $\mathbf{L}_{i,j}$ is finished by dividing $\mathbf{K}_{i,j}$ by the diagonal element $\mathbf{L}_{j,j}$. The outer loop is the i loop, specifying that the computations proceed by rows beginning with row 1 and ending with the last row.

A second example, the kji form, is better suited for the variable-band data structure. In the basic kji form, the outermost loop in the factorization is the k loop. This means that for each k , a new column of \mathbf{L} is finished, and then each j in the middle loop of the factorization corresponds to column j of \mathbf{K} that is modified by the SAXPY operation using column k of \mathbf{L} in the innermost loop. Because the primary computation is a vector SAXPY operation, the kji form is well suited for vector computers such as the CRAY-2 and CRAY Y-MP. An additional benefit of the kji form is that the k th column of \mathbf{L} , which is used for many SAXPY updates, can be stored in the fast local cache memory on the CRAY-2.

Modified kji VBAND method. The VBAND Choleski method, described in reference 8, is a modified form of the basic kji Choleski method. The VBAND method is shown in figure 2 in pseudocode instructions. In this method the basic kji form is used for the column updates in LOOP3, but the jki form is used to finish groups of r columns of \mathbf{L} . The jki form also uses SAXPY operations, but the order of the SAXPY column updates is different. The combination of these two forms works as follows:

1. jki portion: Columns $k, k+1, \dots, k+r-1$ of \mathbf{L} are completed (where r corresponds to the level of loop unrolling) at each iteration of the outer loop k .
2. kji portion: Appropriate columns of \mathbf{K} are modified using columns $k, k+1, \dots, k+r-1$ of \mathbf{L} , beginning with column $k+r$ and ending with the column corresponding to the last nonzero coefficient in the groups of r columns of \mathbf{L} .

The jki portion corresponds to the code required between LOOP1 and LOOP2 in figure 2. The first column k , computed in the jki portion, requires only a vector divide. Column $k+1$ is modified using the just-completed column k in a single SAXPY update

```

LOOP1  $k = 1$  to  $n - r - 1$  in steps of  $r$ 
      {Complete columns  $k, k+1, \dots, k+r-1$  of  $\mathbf{L}$ }
       $lastrk = k + \text{len}(k) - 1$ 
      LOOP2  $j = k+r$  to  $lastrk$ 
            IF  $\mathbf{L}_{j,k}, \dots, \mathbf{L}_{j,k+r-1}$  are not all zero THEN
              {Update column  $j$  of  $\mathbf{L}$  using  $r$  columns of  $\mathbf{L}$  via LOOP3}
              LOOP3  $i = j$  to  $lastrk$ 
                     $\mathbf{K}_{i,j} \leftarrow \mathbf{K}_{i,j} - \mathbf{L}_{i,k} * \mathbf{L}_{j,k}$ 
                     $- \mathbf{L}_{i,k+1} * \mathbf{L}_{j,k+1} - \dots$ 
                     $- \mathbf{L}_{i,k+r-1} * \mathbf{L}_{j,k+r-1}$ 
              END LOOP3
            ENDIF
      END LOOP2
END LOOP1
{Finish any remaining columns of  $\mathbf{L}$  (if  $n$  is not a multiple of  $r$ )}
```

Figure 2. Sequential VBAND Choleski method.

and is then divided by the diagonal element of column $k+1$. Column $k+2$ is modified next by using both columns k and $k+1$ (a double SAXPY loop) and then is finished by a vector divide. Each successive column uses one additional column in the update portion up to loop unrolling level $r-1$. The columns are copied into temporary arrays that are stored in local memory on the CRAY-2. On the CRAY Y-MP, the r columns are also copied into temporary arrays to reduce loop indexing overhead. The kji portion, corresponding to LOOP3 in figure 2, consists entirely of the multiple SAXPY updates. The SAXPY updates are skipped if all r scalar multipliers are zero. This *zero-checking* option reduces the computation rate slightly but reduces overall CPU time considerably by eliminating many unnecessary operations. This feature is especially important in the variable-band method for problems where the increase in storage requirements for the variable-band scheme is large compared with the requirements for a skyline storage scheme. If the zero-checking option is not used in these cases, the number of computations for the VBAND method can be much greater than the number of computations required for a skyline Choleski method.

The key computation in the VBAND algorithm is the multiple SAXPY update loop. Over 97 percent of the computations in the factorization occur in this loop for each of the three problems solved for this study. The multiple SAXPY computations are carried out efficiently on both the CRAY-2 and CRAY Y-MP computers because they

Table 1. VBAND Comparisons for High-Speed Civil Transport Problem

[16 146 equations; Maximum semibandwidth = 594; Average semibandwidth = 319]

Method ^a	CPU time ^b (second)	Wall time ^b (timef)	Rate, MFLOPS	Speedup ^c
NASA Langley CRAY-2 computer in dedicated mode				
LL1	27.9	28.0	73	1.0
LL6	10.8	10.8	188	2.6
LL6L	8.5	8.6	238	3.3
LL6LZ	6.2	6.2	216	4.5
NASA Amcs CRAY Y-MP computer in dedicated mode				
LL1	14.7	14.7	138	1.0
LL6	8.5	8.6	236	1.7
LL6L	7.7	7.7	263	1.9
LL6LZ	5.6	5.6	242	2.6

^aMethods:

- LL1 variable-band Choleski, single SAXPY update
- LL6 variable-band Choleski, multiple (6) SAXPY update
- LL6L variable-band Choleski, multiple (6) SAXPY update, uses temporary storage for six update columns (stored in local memory on CRAY-2)
- LL6LZ variable-band Choleski, multiple (6) SAXPY update, uses temporary storage, skips SAXPY updates if all six scalar multipliers are zero

^bCRAY FORTRAN timing routines *second* and *timef* are used for CPU and wall clock times, respectively, which are given in seconds.^cSpeedup is computed using wall clock times relative to LL1.

allow overlapping of memory accesses with simultaneous use of both the add and multiply functional units. All columns of \mathbf{L} that are accessed in LOOP3 of figure 2 on the CRAY-2 are stored in local memory. Various values for r were tried with this method and the value of 6 was found to be best in most cases. As the level of loop unrolling increases, the amount of serial computation increases and the performance gains level off. If r is too large, the columns may not fit in the local memory of the CRAY-2. In practice, only approximately 10K of the 16K words can be used because the operating system and compiler also use the local memory.

VBAND examples. Table 1 gives a comparison of the basic kji Choleski method, denoted as LL1, with several implementations that add the improvements described above. Runs were made using the H ζ CT example problem, described in section 4, on both the CRAY-2 and the CRAY Y-MP computers in dedicated mode. The speedup of the vectorized LL1 method relative to LL1 compiled with vectorization inhibited is much greater than the speedups shown in table 1. All methods that begin with the

symbols LL6 use the combined jki and kji Choleski forms already described with the level of loop unrolling set to 6. The L designation after the 6 denotes the use of temporary arrays to store the six columns of \mathbf{L} used at each iteration of the outermost loop. On the CRAY-2 these arrays are stored in local memory by using the compiler directive *cdir\$ regfile*. Even though the CRAY Y-MP does not have local memory, the use of temporary arrays for the six columns reduced the indexing overhead and improved the computation rate by approximately 9 percent. (Compare LL6 and LL6L for the Y-MP runs.) The addition of zero checking, denoted by the letter "Z," decreased the computation rate somewhat but reduced the overall time substantially. For the High-Speed Civil Transport problem, the factorization required 2.033 billion operations without zero checking and 1.346 billion operations with zero checking.

A comparison of the CRAY-2 and CRAY Y-MP times in table 1 shows that the techniques used to improve performance result in a greater improvement for the CRAY-2 than for the CRAY Y-MP. The LL1 method is nearly twice as fast on the CRAY Y-MP compared with the CRAY-2 initially.

However, the local-memory implementations LL6L and LL6LZ run at nearly the same rate on both CRAY computers. The VBAND Choleski method, denoted by LL6LZ in table 1, is 4.5 times faster than the vectorized basic *kji* method, LL1, on the CRAY-2, but it is only 2.6 times faster than the LL1 method on the CRAY Y-MP. This difference in results for the two CRAY computers shows that algorithms that are carefully designed to exploit key features of the CRAY-2 architecture also run well on the CRAY Y-MP, but the converse is not necessarily true. The following paragraphs describe the approach used to parallelize this method on the CRAY-2 and CRAY Y-MP.

Parallel PVBAND Choleski Method

The sequential VBAND Choleski method is well suited for parallel implementation on multiprocessor computers such as the CRAY-2 and CRAY Y-MP. FORTRAN listings of an autotasked subroutine and a FORCE subroutine for the PVBAND method are given in the appendix. The basic PVBAND Choleski method is shown in figure 3 in pseudocode instructions. The *jki* part of the PVBAND method, i.e., the completion of columns $k, k + 1, \dots, k + r - 1$ of \mathbf{L} in figure 2, must be finished before the parallel updates using those columns begin. In this paper, the region surrounding the *jki* portion of the PVBAND method is referred to as the barrier region. A barrier region begins at a point at which all CPU's must arrive before any can proceed; then the work within the barrier region is completed by one CPU before the remaining CPU's resume execution. The number of barrier regions required for the PVBAND method is n/r , where n is the number of equations and r is the level of loop unrolling. The number of computations required within the barrier region is very small (between 0.6 and 2.3 percent for the three example problems considered in this paper) compared with the total number of computations required for the factorization. The multiple SAXPY updates can be computed in parallel with no synchronization or communication required. The following two subsections describe general differences in the implementation of the barrier regions and parallel loops using FORCE (macrotasking), microtasking, and autotasking. A separate section is devoted to detailed descriptions of the different parallel implementations.

Barrier regions. In general, the FORCE implementations used in the PVBAND method assume a fixed number of tasks and must wait for all tasks to arrive at a barrier region, let one task complete the work within the barrier, and then exit sequentially. An excellent discussion of barrier algorithms used in

```

LOOP1  $k = 1$  to  $n - r - 1$  in steps of  $r$ 
  Begin barrier region
    {Complete columns  $k, k + 1, \dots, k + r - 1$  of  $\mathbf{L}$ }
  End barrier region
   $lastrk = k + \text{len}(k) - 1$ 
  Parallel LOOP2  $j = k + r$  to  $lastrk$ 
    IF  $\mathbf{L}_{j,k}, \dots, \mathbf{L}_{j,k+r-1}$  are not all zero
    THEN
      {Update columns  $j$  of  $\mathbf{L}$  using  $r$ 
      columns of  $\mathbf{L}$  via LOOP3}
      LOOP3  $i = j$  to  $lastrk$ 
         $\mathbf{K}_{i,j} \leftarrow \mathbf{K}_{i,j} - \mathbf{L}_{i,k} * \mathbf{L}_{j,k} -$ 
           $\mathbf{L}_{i,k+1} * \mathbf{L}_{j,k+1} - \dots$ 
           $- \mathbf{L}_{i,k+r-1} * \mathbf{L}_{j,k+r-1}$ 
      END LOOP3
    ENDIF
  END Parallel LOOP2
END LOOP1
Begin barrier region
  {Finish any remaining columns of  $\mathbf{L}$ 
  (if  $n$  is not a multiple of  $r$ )}
End barrier region

```

Figure 3. Parallel PVBAND Choleski method.

the FORCE language is given in reference 11. On the other hand, microtasking and autotasking implementations use a less restrictive form of the barrier synchronization. In this case, the first active process that reaches a barrier region completes the work within that region and leaves the barrier after the work is completed. Other active processes that arrive at the barrier region must wait until the work is completed or, if the work has already been completed, they immediately skip the barrier region and continue execution. Each type of barrier implementation requires critical regions, that is, regions where variables in shared memory must be incremented by one (and only one) CPU at a time. The shared variable must be locked to all other CPU's during the time that it is being changed by a single CPU. The cost of locking and unlocking variables is a major part of the parallel overhead in the PVBAND implementations.

Parallel loops. The implementations of the parallel loops also differ substantially for the FORCE, microtasking, and autotasking versions. In the FORCE implementation, the parallel loop is distributed to the fixed number of tasks in a wrap-around fashion. That is, task 1 completes the SAXPY updates associated with loop indices $1, 1 + p, 1 + 2p$, etc., where p is the number of tasks. This approach has the advantage of evenly dividing the work in cases


```

c **** begin top of FORCE Barrier
  call lockon(barlck)
  if (ffnbar .lt. np - 1) then
    ffnbar = ffnbar + 1
    call lockoff(barlck)
    call lockon(barwit)
  endif
  if (ffnbar .eq. np - 1) then
c **** end top of FORCE Barrier

.
.
.
c **** execute code inside barrier

.
.
.
c **** begin bottom of FORCE Barrier
  endif
  if (ffnbar .eq. 0) then
    call lockoff(barlck)
  else
    ffnbar = ffnbar - 1
    call lockoff(barwit)
  endif
c **** end bottom of FORCE Barrier

```

Figure 4. FORTRAN code for a standard force barrier.

where the amount of work for each task is nearly the same. The disadvantage of a prescheduled assignment is that a large potential delay may be incurred on busy systems if all tasks are not active. In the microtasking and autotasking implementations, the parallel work is distributed only to active processes. This approach has the advantage of dynamic allocation of the computing resources in a multiuser environment. The disadvantage of this approach is that each CPU must determine its next loop iteration inside a critical region. A self-scheduled parallel loop is also available in FORCE which uses the same approach as microtasking and autotasking. However, the FORCE critical regions, which use macrotasking lock subroutines described in the next section, are much more expensive than the critical regions implemented using microtasking and autotasking.

Parallel Implementation Details

Details of the various parallel implementations of the PVBAND method are discussed next. These details show that autotasking and microtasking best exploit the parallel architecture of the CRAY computers, particularly in multiuser environments. The FORTRAN code used to implement the parallel constructs is not usually seen by algorithm developers since it is inserted automatically at compile time. However, to illustrate the differences in the imple-

mentation of the barrier region and parallel loop using FORCE, microtasking, and autotasking, the actual FORTRAN statements substituted for the FORCE statements and various compiler directives are examined. These details help to explain the sometimes large differences in performance that are observed in the results for the example problems using the same basic parallel algorithm but with different parallel implementations.

FORCE implementations. The FORCE language is described in detail in reference 6. Details of the use of FORCE will not be presented here, but the FORCE statements *Barrier*, *End barrier*, and *Presched DO* are discussed. In FORCE, critical regions are implemented through macrotasking lock subroutines. The locks are identified as shared integer variables. The lock subroutines do not make use of the fast hardware semaphore registers although there are macrotasking subroutines that do use the semaphore registers to implement certain types of critical regions (refs. 3 and 4). A barrier subroutine is also available in macrotasking but is not used by the current version of FORCE.

The FORCE statements *Barrier* and *End barrier* were used to implement the barrier region in the PVBAND method. The *Barrier* and *End barrier* statements are transformed by the FORCE preprocessor into the FORTRAN code shown in figure 4. The code listed in figure 4 uses subroutine calls to lock the shared variables *barlck* and *barwit*. These calls ensure that only one task at a time can increment the counting variable *ffnbar* and prevent tasks from exiting the barrier region before the work inside the barrier is completed. When the routine *lockon* is called, a task attempts to lock the argument variable. If the argument variable is already locked, the task incrementing the counting variable is disconnected from the process and placed in the *Blocked on a Lock* queue. When the argument variable becomes unlocked, the library scheduler will move the tasks from the *Blocked on a Lock* queue into the *Ready* queue. Tasks in the *Ready* queue may then be assigned to logical CPU's that are executed on physical CPU's as they become available. If the argument variable is not locked, the calling task locks the variable and continues execution. Estimates of lock overhead for both the CRAY-2 (ref. 3) and the CRAY Y-MP (ref. 4) are described below. The CRAY-2 requires 200 clock cycles when the variable to be locked is unlocked, whereas the CRAY Y-MP requires 400 clock cycles. If the variable is locked, the CRAY-2 overhead is 4000 clock cycles plus the time that a task must wait for the variable to become unlocked. The

```

c **** begin top of alternate FORCE
Barrier
  if (me .eq. 1) then
    ist = 2
10   continue
    do 20 i = ist, np
      if (lcard(i) .ne. k) then
        ist = i
        goto 10
      endif
20   continue
c **** end top of alternate FORCE
Barrier

execute code inside barrier

c **** begin bottom of alternate FORCE
Barrier
  lcard(1) = k
  else
30   continue
  lcard(me) = k
  if (lcard(1) .ne. k) then
    goto 30
  endif
endif
c **** end bottom of alternate FORCE
Barrier

```

Figure 5. FORTRAN code for an alternate FORCE barrier.

corresponding CRAY Y-MP overhead is 1500 clock cycles plus the wait time.

The FORCE barrier is implemented using two locks, *barlck* and *barwit*. The initial state, before each program call to the barrier, has *barlck* unlocked and *barwit* locked. At the beginning of the FORCE barrier, the first task to reach the *call lockon* statement locks the variable *barlck* and increments the counter *ffnbar*. That task then unlocks variable *barlck* and attempts to lock the second barrier shared variable *barwit*, but since *barwit* is initialized in the locked condition, that task must wait for *barwit* to be unlocked. As soon as *barlck* is unlocked, another task can lock it, increment the counting variable, unlock variable *barlck*, and wait for *barwit* to become unlocked. When the last task arrives at the barrier statement (i.e., $ffnbar = np - 1$, where np is the number of tasks), the variable *barlck* is left locked and the FORTRAN code between the *Barrier* and *End barrier* statements is executed while the remaining $np - 1$ tasks are still either suspended or trying to

lock variable *barwit*. When the single task finishes the sequential FORTRAN code, the counting variable *ffnbar* is decremented and *barwit* is unlocked. One of the remaining tasks now relocks *barwit* (which keeps the remaining tasks still trying to lock *barwit*) and continues execution by first decrementing the counting variable *ffnbar*, unlocking *barwit*, and continuing execution. The last task to lock *barwit* (i.e., when $ffnbar = 0$) unlocks *barlck* and continues execution past the barrier. This method ensures that all tasks arrive at a barrier region before the work inside the barrier work is started, and no task leaves the barrier region until the work inside the barrier region is completed.

The standard FORCE barrier described in the preceding paragraph is expensive on any parallel system where the cost of locking shared variables is high or where the specified number of tasks may not be concurrently executing. The cost of locking shared variables is eliminated by using an alternate FORCE barrier, shown in figure 5. This barrier was developed by Jones (ref. 12). In this barrier a shared array, *lcard*(), is initialized once outside of LOOP1 in figure 3 by using a standard FORCE barrier. The shared array is then used for the barrier region within LOOP1 in figure 3. Each task is associated with one array location (i.e., 1 through p) and writes only to that location. At the beginning of the barrier region, tasks 2 through p write the value of the LOOP1 iteration variable k to their respective locations in array *lcard*. Task 1 checks locations 2 through p in *lcard* and executes the code within the barrier region only after locations 2 through p in *lcard* are equal to k . Tasks 2 through p check location 1 of array *lcard*, continuing in a loop until task 1 completes the barrier region and writes the value of the loop iteration variable k to location 1 of *lcard*. Tasks are not automatically suspended while waiting for appropriate values in the *lcard* array since no calls to lock routines are used. However, excessive CPU time may accumulate when the number of tasks specified at run time is greater than the number actually running concurrently. For this reason, this barrier is used only on lightly loaded systems or in dedicated mode. This type of barrier synchronization is useful within a loop such as in the PVBAND method, where the outer-loop index k can be used for the test value of the *lcard* array. It is not a general-purpose barrier.

The parallel loop in the PVBAND method is implemented in FORCE using the *Presched DO* statement. This statement is translated into a FORTRAN DO loop by the FORCE preprocessor. For example, the FORCE statement

```
Presched DO 1 I=ISTART,IEND
```

```

c **** begin top of process control
structure
cmic$ process
cdir$ suppress (list of variables)
  if (utqbcs (-1)) then
    utqitr = utqifa(1,utqitr)
    if (utqitr .eq. -1) then
c **** end top of process control struc-
ture

c **** execute code inside control
structure

c **** begin bottom of process control
structure
cmic$ end process
  endif
cdir$ suppress (list of variables)
  call utqecs
  endif
c **** end bottom of process control
structure

```

Figure 6. Microtasking code for the *process* and *end process* directives.

is translated to the following FORTRAN DO loop:

```
DO 1 I=ISTART+ME-1,IEND,NP
```

The variable *NP* is the number of tasks specified by the user at run time, and the task identification variable *ME* is an integer between 1 and *NP*. This FORTRAN loop assigns each iteration of the DO loop to the specified number of tasks in a wrapped fashion.

Microtasking implementations. Microtasking is accomplished by inserting compiler directives in FORTRAN source code to identify control structures. The barrier regions in figure 3 are implemented using the *cmic\$ process* and *cmic\$ end process* directives, and the parallel loop is implemented using the *cmic\$ do global* directive. The code containing the microtasking directives is preprocessed so that the FORTRAN code implementing each control structure is inserted. In the following paragraphs, the code generated by the preprocessor for both control structures is discussed.

The *process* and *end process* control structure directives bound a barrier region that is to be completed by only one CPU before any additional CPU's

may proceed past the barrier region. In the micro-tasked algorithm, one CPU completes the *r* columns of *L* at each outer loop iteration, and only one CPU finishes the leftover work at the end. Only processes currently active must wait at the end of the *process* control structure. This approach differs significantly from the FORCE barrier described in the previous paragraphs where all tasks must reach the barrier before the sequential code can begin and all tasks must leave the barrier sequentially to begin executing the next section of code after the sequential code is completed. As soon as the sequential code is completed, all processes may proceed to the parallel loop, including any additional processes that may become active. The FORTRAN code inserted by the micro-tasking preprocessor for the *process* and *end process* directives is discussed next.

Figure 6 shows the FORTRAN code generated by *premult*, the microtasking preprocessor, for the *process* control structure. The preprocessor replaces the *process* directive with a comment and inserts the multitasking code. The *cdir\$ suppress* directive is inserted by the preprocessor at the beginning and end of the barrier region to force all variables referenced within the barrier region from registers into shared memory. This ensures that all CPU's have access to the most current values of the shared variables. The subroutine *utqbcs* is called by each CPU to determine if the control structure is eligible to be processed. The control structure is eligible for processing if the control structure has not been started or if it has been started but not yet completed. A shared variable is used to indicate which control structure is currently active. An additional shared variable is used to update the number of active processes within the control structure. The code within a *process* directive is executed by only one CPU, but all other CPU's must wait until the sequential code is completed. For example, if two of four CPU's are available, and hence two CPU's attempt to execute one of the barrier regions, the first CPU to reach the barrier region will begin executing the code. The second CPU will call routine *utqecs* which either suspends the process temporarily or just delays the second CPU until the single task is completed. As soon as the barrier region is completed, both CPU's proceed to the next section of code. If a third CPU becomes available and arrives at the same barrier region after the work within the barrier region is completed, routine *utqbcs* will return a value of *false* and that CPU will skip the entire barrier region and go on to the next section of code. The implementation of this scheme uses special shared and private variables, hardware semaphore registers to lock appropriate variables, and assembly coded

```

c **** begin top of do global control
structure
cmic$ do global
  utqtrips = comp@((lastr) - (k+6))
  if (utqbcsc (utqtrips) then
    if (utqtrips.lt.0) then
      utqitr = utqifa(1,utqitr)
      utqdummy = comp@(utqitr)
      utqoff = (lastr)+1
      j = utqoff + utqitr
      if (utqdummy .ge. 0) then
20      continue
c **** end top of do global control
structure

c **** execute code inside control
structure

c **** begin bottom of do global control
structure
  utqitr = utqifa(1,utqitr)
  utqdummy = comp@(utqitr)
  j = utqoff + utqitr
  if (utqdummy .ge. 0) goto
  20
endif
endif
call utqecs
endif
c **** end bottom of do global control
structure

```

Figure 7. Microtasking code for the *do global* directive.

routines such as *utqbcsc*, *utqifa*, and *utqecs* to efficiently carry out the necessary synchronization steps.

The *do global* directive is used to implement the parallel loop in figure 3. Each iteration of LOOP2, the multiple SAXPY updates, is assigned one at a time to each active process. This mode of parallel execution differs from the FORCE implementation that distributes the iterations of LOOP2 to each task in a predetermined sequence. The total number of loop iterations is computed from the limit variables in the FORTRAN DO loop. The particular loop index *j* executed by each CPU is determined by reading a shared variable within a critical region and then incrementing that variable.

Figure 7 shows the code generated by *premult* for the *do global* control structure. CPU's that reach the parallel DO loop will begin execution of the loop as long as routine *utqbcsc* returns a value of *true*. This

means that even CPU's that may become available after much of the work within the parallel region is completed may still be used in the parallel loop. Each CPU determines its current iteration value *utqitr* through calls to the routine *utqifa*. The routine *utqifa* maintains a shared variable used to determine iteration values. The shared variable does not appear in the FORTRAN code in figure 7. All CPU's that enter the parallel DO loop continue to execute different loop iterations until the loop counter *utqitr* gets to zero or to a positive value. Each CPU finishes its last available loop iteration and then calls routine *utqecs*, which delays all CPU's until all work in the parallel region is completed.

The code displayed in figures 6 and 7 was produced on the CRAY-2. The corresponding code generated on the CRAY Y-MP is different because of hardware differences and the greater use of intrinsic functions that are used to test semaphore registers and to read from and write to shared registers. These differences can cause noticeable changes in performance even though the same basic strategy is followed on both machines in implementing the barrier regions and parallel loops.

Autotasking implementations. The dependency phase *fpp* of the autotasking compilation was not able to detect the parallelism in the VBAND method. Therefore, directives that were functionally equivalent to microtasking directives were inserted in the FORTRAN subroutine used to carry out the PVBAND method. Two autotasking implementations for the PVBAND method were explored.

In the first autotasking implementation, LOOP2 in the PVBAND algorithm is preceded by the *cmic\$ do all* directive. The *do all* directive defines the start of a parallel region, and any code outside this region is to be executed only by the master process. The advantage of this approach is simplicity (i.e., only a single directive is required to parallelize the entire method), but the disadvantage is that each outer loop iteration requires additional overhead to start up a new parallel region.

In the second autotasking implementation, the entire subroutine is declared a single parallel region using the *cmic\$ parallel* and *cmic\$ end parallel* directives. This approach is identical to the microtasking implementation already discussed, but different directives are used. All variables within the subroutine are explicitly declared to be shared or private within the *parallel* directive. The autotasking directives *cmic\$ case* and *cmic\$ end case* perform the same function as the microtasking *process* and *end*

process directives, whereas the *cmic\$ do parallel* directive corresponds to the microtasking *do global* directive. All code outside the control structures, but inside the parallel region, is executed by all active processes. In experiments, the second autotasking method was found to be more efficient and was used for all autotasking results presented in this paper.

The autotasking translation phase, generated by *fmp*, produces code very similar to the code produced by *premult* using microtasking. The autotasked subroutine has additional code generated to handle the creation of parallel regions. In addition, separate code segments are created for the master and the slave processes. The code generated for the autotasking *do parallel* directive is the same as the code for the microtasking *do global* directive. Likewise, the code generated for the autotasking *case* directive is the same as the code for the microtasking *process* directive, except for some differences related to the *suppress* directive. In microtasking, the *suppress* directive, at the beginning of the *process* control structure, forces all variables that will be read inside the control structure to be written from registers to memory. At the end of the *process* control structure, all variables that have been modified are again written to main memory. In autotasking, only shared variables are subject to the *suppress* directive that precedes and follows the *case* directive.

CRAY-2 local-memory considerations. The local-memory version of the PVBAND method requires some additional considerations. The columns of **L**, completed inside the barrier region in the PVBAND method, are computed by only one CPU but are used by all other CPU's. Since the CRAY-2 and CRAY Y-MP are shared-memory machines, these columns are accessible to all CPU's. However, the fast local memory cache on the CRAY-2 is not shared between CPU's and so each CPU must copy the columns into local memory before beginning the parallel SAXPY updates. Two approaches were tested to minimize the extra cost of copying the columns of **L** into local memory. No noticeable difference was observed for the two approaches in the microtasked and autotasked implementations. However, for the FORCE implementations, a significant difference was observed.

The first approach was to complete all r columns inside a single barrier, or control structure. The CPU completing the r columns copies them into its local memory as soon as each column of **L** is completed. The remaining CPU's copy all six columns in one loop as soon as the single CPU finishes the barrier region. The second approach was to use r barrier or control structures, allowing the idle CPU's

to copy each column into local memory as soon as each column is completed. This approach reduces the time required at the end of the barrier to copy the columns into local memory, but at the expense of more barriers and an increased number of DO loops for the copying operation. Because of the high cost of FORCE barrier statements, the second approach was much slower than the first. In all the results presented in section 5 for the CRAY-2, the first approach was used for the FORCE implementations, whereas the second approach was used for the microtasking and autotasking implementations.

Amdahl's law. The degree of parallelism for any method is always limited by the number of sequential computations relative to the number of parallel computations. A much-used measure of the effect of sequential operations on the parallel efficiency of a given method is Amdahl's law, which states that the theoretical maximum speedup S possible for a parallel algorithm is given by

$$S = \frac{p}{1 + w_s(p - 1)} \quad (4)$$

where p is the number of processors and w_s is the percentage of time required by sequential computations. (See ref. 13 for a thorough discussion of several measures of the degree of parallelism.) For the PVBAND method, the *jki* portion at each outer loop iteration is the sequential part of the method, whereas all remaining computations are carried out in parallel in LOOP3 (fig. 3). Maximum theoretical speedup estimates for the PVBAND method for each of the three example structural analysis problems are given in table 2 by using equation (4). The percentages used for the computations in table 2 were obtained by inserting counting variables in the barrier region of code in the autotasked and microtasked implementations. The maximum speedup estimations assume that the sequential computations are carried out at the same rate as the multiple SAXPY updates in LOOP3 and also that no overhead exists for the parallel loop and the barrier regions. The actual speedups attained will always be less than the estimates in table 2, but values close to these estimates indicate good parallel implementations.

PVBAND examples. Table 3 gives a comparison of parallel and sequential timings for the High-Speed Civil Transport problem. The PVBAND implementations are denoted by appending the letters A, M, F, and J to the VBAND implementation symbols in table 3. The letters correspond to the autotasking, the microtasking, the FORCE using the

Table 2. Effect of Amdahl's Law on Maximum Theoretical Speedup for PVBAND Method

Number of processors, p	Maximum theoretical speedup for problem—		
	HSCT ^a	SRB ^b	3-D panel ^c
2	1.96	1.96	1.99
4	3.74	3.75	3.93
8	6.89	6.93	7.68
16	11.90	12.03	14.70

^aHSCT: High-Speed Civil Transport, sequential operations 2.3 percent of total operations for factorization.

^bSRB: Space Shuttle solid rocket booster, sequential operations 2.2 percent of total operations for factorization.

^c3-D panel: Cross-ply composite laminate, sequential operations 0.59 percent of total operations for factorization.

standard barrier, and the FORCE using the special-purpose barrier implementations of the PVBAND method, respectively. Runs were made on the CRAY-2 in dedicated mode comparing the parallel PVBAND implementations with the corresponding sequential VBAND implementations described previously. Parallel speedups are computed using the PVBAND wall clock times and the sequential VBAND wall clock times for the same method. Speedups are often computed by dividing the parallel CPU time by the parallel wall clock time, but such speedups do not accurately reflect the cost of parallel processing. A comparison of the CPU time for each PVBAND implementation with the CPU time for the sequential VBAND implementation of the same method shows that the total CPU time is always greater for the parallel implementations than for a single-processor implementation of the same algorithm. If the CPU time for the parallel implementations is used to compute parallel speedups, implementations with the largest CPU time may appear to have the greatest parallel efficiency. For example, in table 3 the parallel speedup for the LL6LF1 FORCE implementation would be 1.9 if computed by dividing the parallel CPU time by the parallel wall clock time. However, the actual wall clock time for this implementation is greater than the wall clock time for the sequential VBAND implementation.

The FORCE PVBAND implementations, which used the special alternate barrier described previously, were the fastest in dedicated mode and have the highest parallel speedups. The faster times are expected since no locking of variables is required for the special FORCE barriers and no tasks are

ever suspended while waiting for sequential work to be completed. However, the special barrier works effectively only in a dedicated environment. The FORCE implementations that used the standard barriers were the slowest of the PVBAND implementations. The LL6LF1 implementation, which uses one barrier for each column computed in the jki portion of the PVBAND method, is very inefficient on the CRAY-2. The wall clock time for this method was greater than the single-processor time of the corresponding VBAND implementation. CPU times were also much higher for the FORCE implementations on the CRAY-2 than for the microtasking and autotasking implementations. This difference reflects the high cost of barriers in the FORCE implementations. The differences between autotasking and microtasking are small, with autotasking slightly faster for these runs.

The parallel speedups shown in table 3 are affected by both the number of sequential computations and the overhead associated with the implementation of barrier regions. Table 4 shows the percentage of sequential operations for this problem. The LL1 methods have a lower number of sequential computations compared with the LL6 methods but they also require six times as many barrier regions. As a result, the parallel speedups in table 3 are, in some cases, higher for the LL6 methods than for the LL1 methods. This is especially true for the FORCE implementations that use the standard FORCE barrier. The results in table 3 also show that the use of temporary arrays for local memory versions and zero-checking reduces parallel speedups. However, both local memory and zero checking reduce total wall clock time and are therefore beneficial.

Table 3. PVBAND Method Timings for High-Speed Civil Transport Problem

[NASA Langley CRAY-2 computer in dedicated mode;
 16 146 equations; Maximum semibandwidth = 594;
 Average semibandwidth = 319]

Method ^a	CPU time (second)	Wall time (timef)	Speedup ^b
Single-processor VBAND Choleski method			
LL1	27.87	27.95	1.0
LL6	10.79	10.82	1.0
LL6L	8.53	8.55	1.0
LL6LZ	6.20	6.22	1.0
Parallel PVBAND method using autotasking			
LL1A	32.85	8.29	3.37
LL6A	12.33	3.13	3.46
LL6LA	9.91	2.67	3.20
LL6LZA	7.77	1.96	3.17
Parallel PVBAND method using microtasking			
LL1M	33.53	8.54	3.27
LL6M	12.90	3.67	2.95
LL6LM	10.06	2.89	2.96
LL6LZM	7.85	2.15	2.89
Parallel PVBAND method using standard FORCE barrier			
LL1F	50.86	17.18	1.63
LL6F	14.26	3.98	2.72
LL6LF1	54.52	28.52	
LL6LF2	13.37	4.25	2.01
LL6LZF	11.42	3.94	1.58
Parallel PVBAND method using special FORCE barrier ^c			
LL1J	30.26	7.64	3.66
LL6J	12.17	3.06	3.54
LL6LJ	9.56	2.41	3.55
LL6LZJ	7.15	1.79	3.47

^aVBAND method names appended by letters A, M, F, and J to denote autotasking, microtasking, FORCE with standard barrier, and FORCE with special alternate barrier implementations of PVBAND method, respectively.

LL6LF1 - uses six barrier statements for each outer loop iteration in *jki* portion of PVBAND method

LL6LF2 - uses only one barrier statement to finish all six columns in *jki* portion of PVBAND method

^bPVBAND speedups calculated using the PVBAND wall clock time and the corresponding wall clock time for the single-processor method.

^cSpecial FORCE barrier avoids use of critical regions that require calls to routines that lock the shared variables.

Solution of Triangular Systems

The solution of the triangular systems is a forward solution (eq. (2a)), followed by a backward solution (eq. (2b)). As in the factorization of \mathbf{K} , there

are several possible implementations of the triangular solutions. Since the factorization is more costly than the triangular solutions, the data structure used for the factorization often determines the implementations of the forward and backward solutions. In

Table 4. Number of Computations and Memory Requirements for Example Problems

Method	Factorization (total operations)	Sequential, percent of total	Memory (64-bit words)
High-Speed Civil Transport aircraft			
(16 146 equations; Max. semibandwidth = 594; Av. semibandwidth = 319)			
LL1	2 033 084 710	0.09	5 160 501
LL6	2 033 161 281	1.5	5 160 591
LL6LZ	1 345 866 285	2.3	5 160 591
Composite cross-ply laminate with hole			
(11 929 equations; Max. semibandwidth = 1597; Av. semibandwidth = 1081)			
LL6LZ	13 098 674 109	0.59	12 901 825
Space Shuttle solid rocket booster			
(54 870 equations; Max. semibandwidth = 558; Av. semibandwidth = 355)			
LL6LZ	5 427 998 155	2.2	19 522 323

the variable-band data structure, the lower triangular part of \mathbf{L} is stored by columns. As a result, the forward solution is carried out using SAXPY operations. This so-called column-sweep algorithm is shown in figure 8(a). For the backward solution, the columns of \mathbf{L} are now the rows of \mathbf{L}^T ; therefore, a different approach that uses inner products is used to ensure stride one memory accesses. Figure 8(b) illustrates this inner product algorithm.

```

LOOP1 j = 1 to n
  IF fj = 0 THEN skip LOOP2
  zj = fj/Lj,j
  lastr = j + len(j) - 1
  LOOP2 i = j + 1 to lastr
    fi = fi - Li,j * zj
  END LOOP2
END LOOP1

```

(a) Column-sweep forward solution, $\mathbf{Lz} = \mathbf{f}$.

```

LOOP1 j = n to 1
  lastr = j + len(j) - 1
  LOOP2 i = j + 1 to lastr
    zj = zj - Li,j * ui
  End LOOP2
  uj = zj/Lj,j
End LOOP1

```

(b) Inner product backward solution, $\mathbf{L}^T \mathbf{u} = \mathbf{z}$.

Figure 8. Forward-and-backward triangular solution algorithms.

The zero-checking option used to reduce operations in the factorization can also be applied to the forward solution, as shown in figure 8(a). For problems where the right-hand side vector has few

nonzero values, this strategy can significantly reduce the operation counts for the forward solution. The zero-checking strategy cannot be used effectively for the inner products in the backward solution. The loop-unrolling techniques described for the factorization were also applied to both the forward and backward solutions. The computation rates for the triangular solutions were improved significantly on the CRAY-2 as a result of loop unrolling, but on the CRAY Y-MP the improvement was very small. This difference is due to the positive effect of chaining on the CRAY Y-MP which caused the initial triangular solutions to be nearly two times faster than the CRAY-2 versions. Loop unrolling made up much of the difference on the CRAY-2, but the overhead of additional sequential computations required for the loop-unrolling version limited the amount of the improvement. A FORTRAN listing of the forward and backward solutions using loop unrolling and zero checking is given in the appendix.

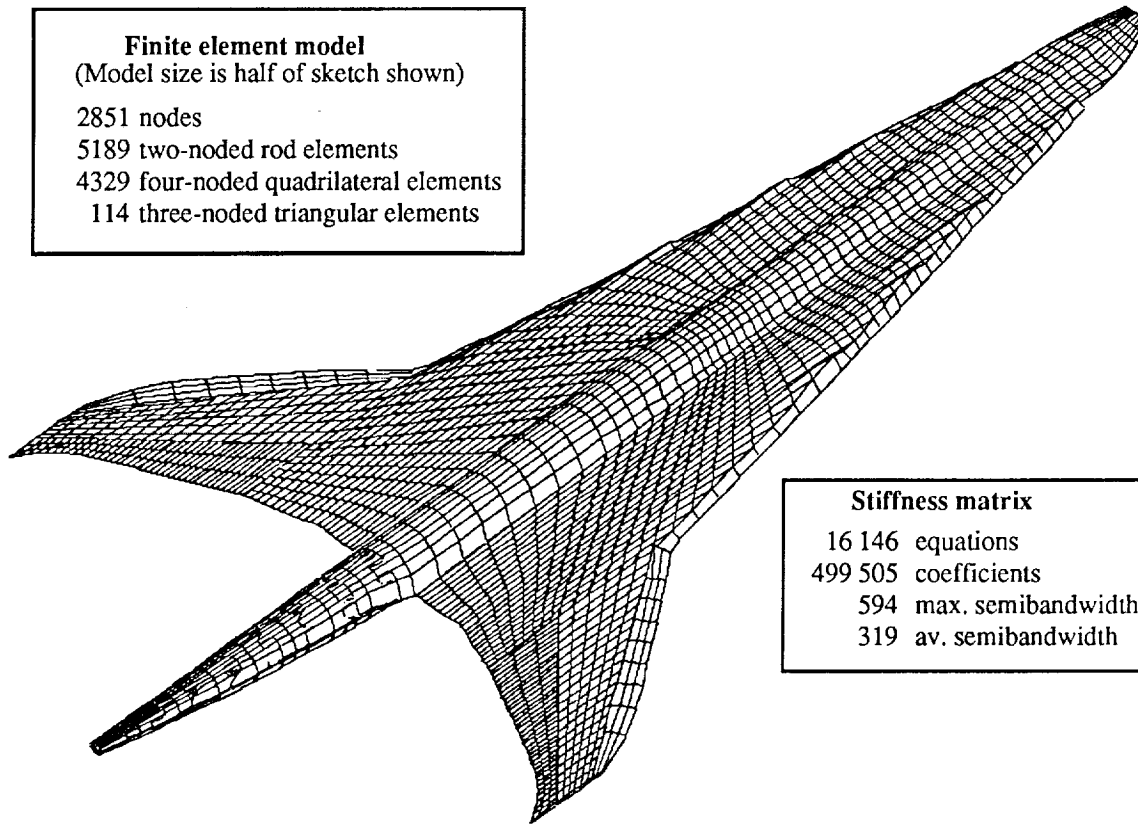
For the parallel implementations of the PVBAND method, the triangular solutions were carried out on a single processor. Times for the triangular solutions are given in section 5 for the three structural analysis problems.

Out-of-Core Extensions

An out-of-core version of both the VBAND and PVBAND methods is under development. The out-of-core version permits the solution of problems that are too large to fit in main memory or allows users to solve larger problems interactively on machines where interactive memory limits are imposed. The out-of-core method is designed to take advantage of the large memories available on today's

Finite element model
 (Model size is half of sketch shown)

- 2851 nodes
- 5189 two-noded rod elements
- 4329 four-noded quadrilateral elements
- 114 three-noded triangular elements



Stiffness matrix

- 16 146 equations
- 499 505 coefficients
- 594 max. semibandwidth
- 319 av. semibandwidth

Figure 9. Finite element model of High-Speed Civil Transport aircraft.

supercomputers. This is accomplished by using the *buffer in* and *buffer out* CRAY FORTRAN I/O statements and three buffers to store blocks of the factored matrix. The input matrix is stored in large blocks by records in an unformatted file. The I/O required is carried out using one of the three buffers in a rotating fashion, and computations required for the factorization proceed in the remaining buffers while I/O operations are underway. The data are read in once and written out after an entire buffer is finished, and no movement of data in memory is required. The overlapping of I/O and computations in this method results in factorization times for the out-of-core version that are very close to the factorization times for the in-core version of the factorization. The main difference in wall clock times for the in-core methods compared with the out-of-core methods occurs in the triangular solutions. The triangular solutions require two passes through the factored matrix, and the number of computations per coefficient is only two. As a result, the time required for the triangular solutions is increased dramatically for the out-of-core methods.

4. Structural Analysis Applications

In this paper, three representative structural analysis problems are used to compare the different parallel implementations of the variable-band Choleski method. The three problems represent full-scale problems that can all be stored in main memory on the CRAY-2 and CRAY Y-MP computers. The example structural analysis problems described in this work were carried out using the COmputational MEchanics Testbed (COMET), a large-scale structural analysis software system developed by the Computational Mechanics Branch at the LaRC (refs. 14 and 15). Although the structural response determination is the primary goal of structural analysis, the three problems considered herein are used primarily to assess the performance of the parallel/vector equation solvers. Hence, only a brief description of each structural problem is provided.

High-Speed Civil Transport Aircraft

Projected trends in travel to the Pacific Basin have led to a renewed national interest in the

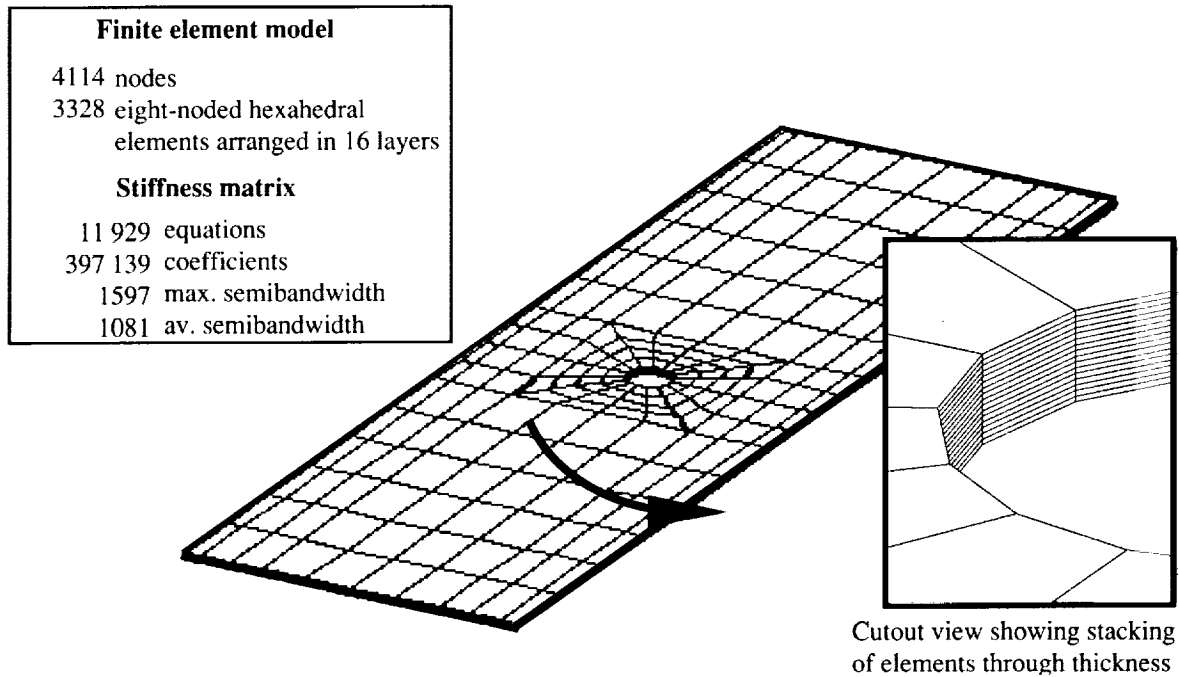


Figure 10. Finite element model of cross-ply laminate with a hole.

development of a High-Speed Civil Transport (HSCT) aircraft (refs. 16 and 17). The design of such a vehicle will require an integrated analysis approach involving both structures and aerodynamics. To accelerate the development of mathematical models of various structural configurations, a parameterized finite element model has been developed. The finite element model (fig. 9) used herein represents the symmetric half of the HSCT aircraft. The model involves 2851 nodes, 5189 two-noded rod elements, 4329 four-noded quadrilateral elements, and 114 three-noded triangular elements. The linear static response is determined for the case of a wingtip loading. The linear system for this problem has 16 146 equations with a maximum semibandwidth of 594 and an average semibandwidth of 319. The nodes in the finite element model were ordered using the reverse Cuthill-McKee algorithm implemented in the COMET software to minimize bandwidth.

Cross-Ply Composite Laminate With a Hole

A detailed stress analysis of composite structures is often required to accurately determine through-the-thickness (or interlaminar) stress distributions. Some sources of interlaminar stress gradients include free-edge effects, holes, and ply drop-offs (e.g., ta-

pered stiffener attachment flanges). To study these effects, three-dimensional finite element models are frequently used. To study the performance of these solvers for three-dimensional finite element models, the overall structural response of an 8-ply cross-ply composite laminate with a central circular hole was considered. The finite element model (fig. 10) used herein has 4114 nodes and 3328 eight-noded hexahedral solid elements arranged in 16 layers. This finite element model is adequate for overall response characteristics but must be refined in order to determine the interlaminar stress state accurately. The linear system for this problem has 11 929 equations with a maximum semibandwidth of 1597 and an average semibandwidth of 1081. The larger bandwidth of this problem is characteristic of three-dimensional models. The nodes in the finite element model were ordered using the reverse Cuthill-McKee algorithm implemented in the COMET software to minimize bandwidth.

Space Shuttle Solid Rocket Booster

A preliminary assessment of the Space Shuttle solid rocket booster (SRB) global shell response to selected prelaunch loads was presented in reference 18. The two-dimensional shell finite element model used in this study was translated into a format compatible

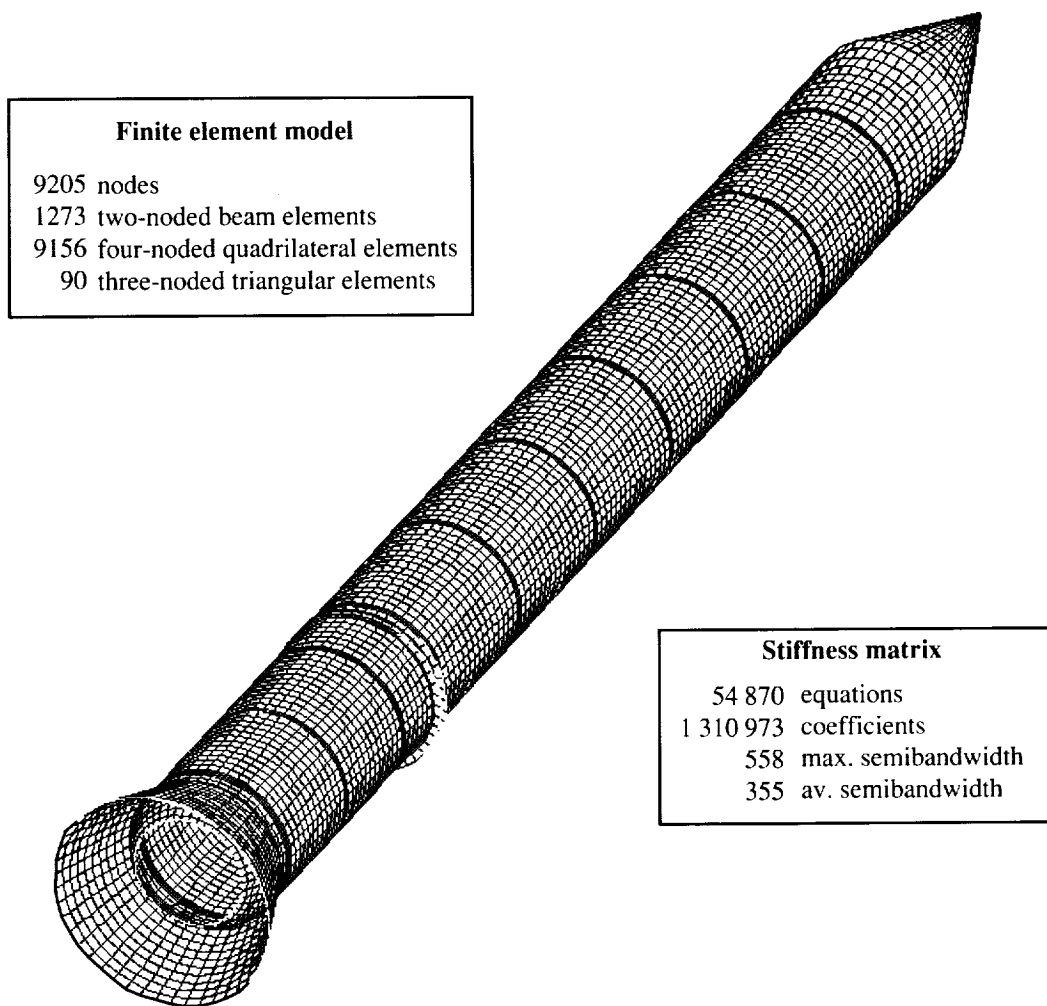


Figure 11. Finite element model of Space Shuttle solid rocket booster.

with the COMET software. The finite element model (fig. 11) involves 9205 nodes with 1273 two-noded beam elements, 90 three-noded triangular elements, and 9156 four-noded quadrilateral elements. The linear static response to the internal pressure loading in the SRB was obtained. The linear system for this problem has 54870 equations with a maximum semibandwidth of 558 and an average semibandwidth of 355. The nodes in the finite element model were ordered using the bandwidth minimization options in PATRAN software developed by PDA Engineering (ref. 19). The PATRAN bandwidth optimization reduced the bandwidth for this problem more than in previously reported results (refs. 8 and 9) which used the reverse Cuthill-McKee algorithm implemented in the COMET software. As a result, the number of


operations required for the matrix factorization was reduced approximately 25 percent.

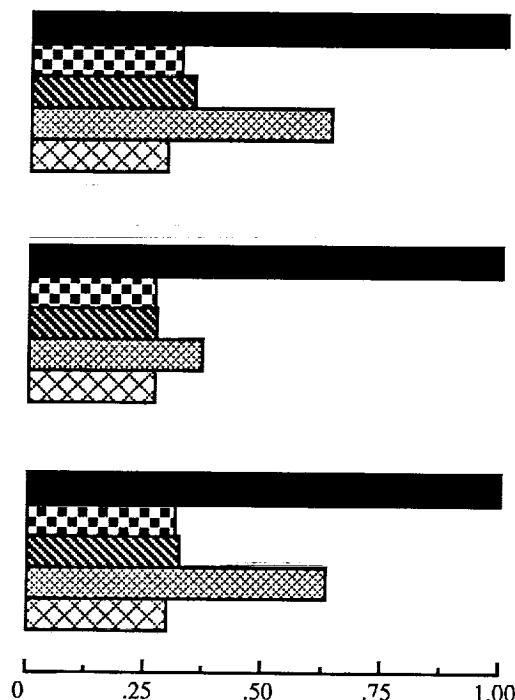
5. Numerical Results and Comparisons

In this section, timing results are presented for the matrix factorization stage of three structural analysis problems comparing the sequential VBAND Choleski method and the parallel PVBAND Choleski implementations. The experiments were performed on the CRAY-2 at the LaRC and on the CRAY Y-MP at the ARC. Results are presented from computer runs on both the CRAY-2 and CRAY Y-MP computers made first during dedicated single-user mode and second from runs made in

Method	CPU time (<i>second</i>)	Wall time (<i>timef</i>)	Rate, MFLOPS	Speedup
High-Speed Civil Transport aircraft (16 146 equations; Max. semibandwidth = 594; Av. semibandwidth = 319)				
VECTOR	6.20	6.22	216	
AUTO	7.77	1.96	687	3.17
MICRO	7.85	2.15	626	2.89
SFORCE	11.42	3.94	342	1.58
JFORCE	7.15	1.79	752	3.47
Composite cross-ply laminate with hole (11 929 equations; Max. semibandwidth = 1597; Av. semibandwidth = 1081)				
VECTOR	47.92	48.07	272	
AUTO	50.52	12.84	1020	3.74
MICRO	51.22	13.04	1004	3.69
SFORCE	55.34	17.57	746	2.74
JFORCE	51.27	12.95	1011	3.71
Space Shuttle solid rocket booster (54 780 equations; Max. semibandwidth = 558; Av. semibandwidth = 355)				
VECTOR	24.83	24.91	218	
AUTO	30.25	7.78	698	3.20
MICRO	31.07	8.01	678	3.11
SFORCE	43.87	15.74	345	1.58
JFORCE	29.20	7.40	734	3.37

(a) Comparison of vector and parallel implementations of Choleski factorization for three example problems.


 VECTOR - no parallelization
 AUTO - autotasking
 MICRO - microtasking
 SFORCE - macrotasking, standard FORCE barrier
 JFORCE - macrotasking, special FORCE barrier



(b) Comparison of wall clock times (normalized).

Figure 12. CRAY-2 factorization times in dedicated mode with a maximum of four CPU's.

normal multiuser environments. The runs made in dedicated mode show the maximum computation rates possible for the three problems using the PVBAND method and measure parallel overhead without additional delays caused by other programs executing at the same time. The runs made in multiuser mode measure the wall clock times for both the VBAND and PVBAND methods when other programs are executing. The results show that the PVBAND method is efficient on the CRAY computers and significantly reduces wall clock time in both multiuser and dedicated modes.

In the results that follow, the names VECTOR, AUTO, MICRO, SFORCE, and JFORCE are used to denote the vector implementation of the VBAND method and the autotasking, microtasking, and two FORCE implementations of the PVBAND method, respectively. SFORCE denotes the use of the FORCE *Barrier* statement, and JFORCE denotes the FORCE implementation that uses the alternate barrier described in section 3. All parallel methods compared in this section use loop unrolling to level 6


and the zero-checking option described in section 3. Table 4 shows the number of computations, percentage of sequential operations during factorization, and memory requirements for each of the example structural analysis problems.

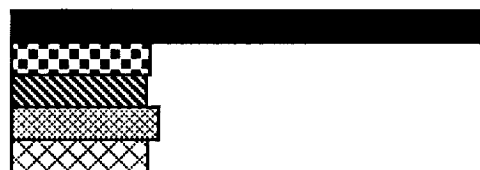
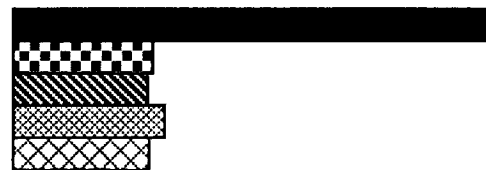
Timing Considerations

The measure of time for the comparison of algorithms is an important consideration, particularly on multiuser parallel computers like the CRAY computers which offer several timing options. For sequential algorithms, CPU time is the most often quoted measure of time and is usually measured on the CRAY computers by the FORTRAN function *second()*. For parallel runs, *second()* returns a cumulative CPU time, summed for all processes and as such provides no useful measure of parallel speedup. A different timing function, *tsecond()*, may be used to measure individual CPU time for a given task. The CPU time for a routine executed by p processes may be taken as the largest CPU time returned for any single process. This routine can be used as an effective measure of

Method	CPU time (<i>second</i>)	Wall time (<i>timef</i>)	Rate, MFLOPS	Speedup
High-Speed Civil Transport aircraft (16 146 equations; Max. semibandwidth = 594; Av. semibandwidth = 319)				
VECTOR	5.50	5.50	245	
AUTO	6.46	1.62	831	3.40
MICRO	6.29	1.57	857	3.50
SFORCE	6.98	1.75	769	3.14
JFORCE	6.33	1.58	852	3.48
Composite cross-ply laminate with hole (11 929 equations; Max. semibandwidth = 1597; Av. semibandwidth = 1081)				
VECTOR	46.33	46.34	283	
AUTO	48.71	12.18	1075	3.80
MICRO	48.11	12.03	1089	3.85
SFORCE	48.69	12.19	1075	3.80
JFORCE	49.14	12.29	1066	3.77
Space Shuttle solid rocket booster (54 780 equations; Max. semibandwidth = 558; Av. semibandwidth = 355)				
VECTOR	21.92	21.93	248	
AUTO	25.56	6.39	849	3.43
MICRO	24.83	6.21	874	3.53
SFORCE	27.19	6.80	798	3.23
JFORCE	25.10	6.28	864	3.49

(a) Comparison of vector and parallel implementations of Choleski factorization for three example problems.


 VECTOR - no parallelization
 AUTO - autotasking
 MICRO - microtasking
 SFORCE - macrotasking, standard FORCE barrier
 JFORCE - macrotasking, special FORCE barrier



0 .25 .50 .75 1.00

(b) Comparison of wall clock times (normalized).

Figure 13. CRAY Y-MP factorization times in dedicated mode with a maximum of four CPU's.

the load balance for a given algorithm executed on a fixed number of processors, as in FORCE implementations described in this paper. However, *tsecnd()* often does not provide an accurate measure of the total time required for parallel algorithms. Even on dedicated systems, the *tsecnd()* time is often substantially less than the wall clock time measured for the same process. This difference may lead to overly optimistic performance projections that cannot actually be obtained even in a dedicated environment. The most reliable measure of actual achievable computation rates for given parallel algorithms on the CRAY computers is obtained using wall clock time (function *timef*) in a dedicated, single-user environment. In this paper, only wall clock times are used to evaluate the performance of the parallel implementations. Additional results are given that show the times reported by *second()*, and they are referred to as CPU time. The CPU times show in general that parallel runs accumulate more total CPU time than the corresponding CPU time for a single-processor

run, and these times generally increase as the number of processors increases.


Dedicated Runs

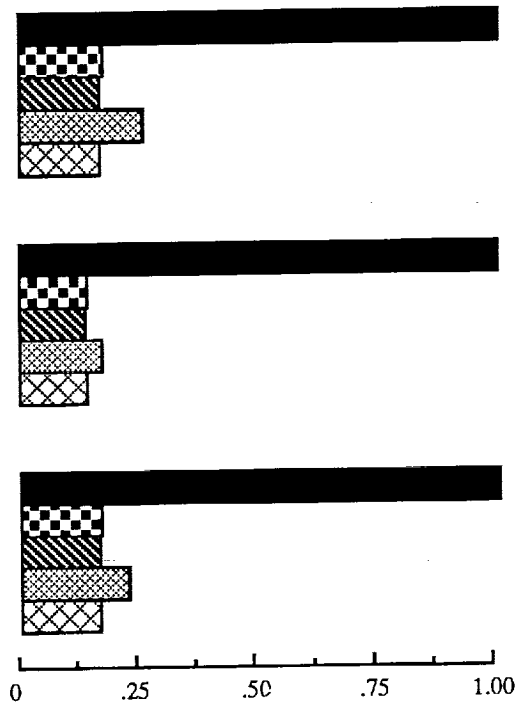
Dedicated, single-user runs were obtained for the three example problems to measure the maximum performance of the VBAND and PVBAND implementations. For each run, the wall clock time was used to compute the MFLOP rate and parallel speedups were measured relative to the wall clock time for the VBAND method VECTOR.

CRAY-2 results. Figure 12 shows CRAY-2 results for all three structural analysis problems using the VBAND and PVBAND implementations. The JFORCE method, which uses the special FORCE barrier with no calls to macrotasking lock subroutines, is the fastest PVBAND implementation on two of the three problems. The SFORCE method, which uses the standard FORCE barrier, is the slowest PVBAND implementation, demonstrating the high

Method	CPU time (second)	Wall time (timef)	Rate, MFLOPS	Speedup
High-Speed Civil Transport aircraft (16 146 equations; Max. semibandwidth = 594; Av. semibandwidth = 319)				
VECTOR	5.50	5.50	245	
AUTO	7.64	.96	1402	5.73
MICRO	7.39	.93	1447	5.91
SFORCE	10.93	1.42	948	3.87
JFORCE	7.43	.93	1447	5.91
Composite cross-ply laminate with hole (11 929 equations; Max. semibandwidth = 1597; Av. semibandwidth = 1081)				
VECTOR	46.33	46.34	283	
AUTO	51.32	6.43	2037	7.21
MICRO	50.55	6.34	2066	7.31
SFORCE	55.00	7.96	1646	5.82
JFORCE	52.11	6.52	2008	7.11
Space Shuttle solid rocket booster (54 780 equations; Max. semibandwidth = 558; Av. semibandwidth = 355)				
VECTOR	21.92	21.93	248	
AUTO	29.69	3.72	1459	5.90
MICRO	28.90	3.64	1491	6.02
SFORCE	39.44	4.95	1097	4.43
JFORCE	29.10	3.64	1491	6.02

(a) Comparison of vector and parallel implementations of Choleski factorization for three example problems.


 VECTOR - no parallelization
 AUTO - autotasking
 MICRO - microtasking
 SFORCE - macrotasking, standard FORCE barrier
 JFORCE - macrotasking, special FORCE barrier



(b) Comparison of wall clock times (normalized).

Figure 14. CRAY Y-MP factorization times in dedicated mode with a maximum of eight CPU's.

cost of the lock subroutines used by the standard FORCE implementation. Over 1 GIGAFLOP was achieved on the largest bandwidth problem, the composite cross-ply laminate, for all but the SFORCE method. This problem has the lowest percentage of sequential operations in the barrier region and the longest vector lengths in the parallel update section. On the CRAY-2, autotasking (method AUTO) is slightly faster than microtasking (method MICRO) for all three problems.

CRAY Y-MP results. Figures 13 and 14 give timing results from runs on the CRAY Y-MP using four and eight processors in a dedicated mode. The eight-processor CRAY Y-MP results in figure 14 show rates of over 1 GIGAFLOP for all three problems and over 2 GIGAFLOPS for the cross-ply laminate problem. Parallel speedups for each of the problems compare favorably with the theoretical maximums computed in table 2.

A comparison of the CRAY-2 results in figure 12 and the CRAY Y-MP results in figure 13 shows that

both the computation rates and parallel speedups are higher on the CRAY Y-MP than on the CRAY-2. These results may be due to the fact that the CRAY Y-MP has more hardware available than the CRAY-2 for multitasking purposes. There are nine clusters of shared registers for interprocessor communication and synchronization, each of which has 8 shared address, 8 shared scalar, and 32 semaphore registers. In contrast, the CRAY-2 has only eight semaphore registers that synchronize common memory references in multitasked jobs.

The CRAY Y-MP performance of the FORCE implementations relative to the autotasking and microtasking implementations is significantly improved over the CRAY-2 results. The FORCE times in figure 13 from CRAY Y-MP runs using four CPU's are very close to the autotasking and microtasking times. However, the FORCE times are noticeably slower than the autotasking and microtasking times as the number of processors is increased from four to eight (fig. 14).

On the CRAY Y-MP, the microtasked PVBAND implementation is slightly faster than the autotasked implementation. This result is reversed from the relative performance of autotasking and microtasking on the CRAY-2. The PVBAND implementations on the CRAY Y-MP were compiled using the CFT77 version 4.0 compiler. Earlier results that used a previous compiler had shown faster times for autotasking. The CRAY-2 compiler used for the results presented in this paper is CFT77 version 3.1.

The times in figures 12-14 show that the PVBAND method is efficient and achieves a high percentage of the maximum possible computation rate on both CRAY computers. The times for the Space Shuttle SRB problem are the lowest wall clock and CPU times yet achieved for this problem (refs. 8 and 9). The rates achieved in the VBAND and PVBAND implementations use only FORTRAN subroutines. Further improvements in the computation rates can be achieved by writing parts of the PVBAND implementations in assembly code. In addition, if the next group of r columns of L is completed as part of the parallel update region in the PVBAND method at each stage, or outer loop iteration, the barrier region can be eliminated. This change eliminates the barrier region and should improve the performance of the PVBAND method as the number of processors increases. These improvements are the subject of current and future studies.

Multiuser Runs

Although the dedicated runs indicate the maximum benefit of parallel processing to a single user, the typical computing environment on multiprocessor CRAY supercomputers is a multiuser mode with many users sharing the four to eight CPU's. Many users may be logged in at any given time and may be interactively executing jobs, editing files, or submitting batch runs to one or more of several job queues. In this environment, users rarely have control of all the CRAY CPU's for an entire job execution. The performance of multitasked jobs will differ across machines according to the system parameters defined by each computing center. In order to see the effect of multitasking in a busy environment on the CRAY-2 and the CRAY Y-MP, each problem was run multiple times using the VBAND and PVBAND implementations. Multiple runs were submitted using script files during normal operating hours. The two largest problems were run in batch queues that are normally run after the normal working hours along with other large batch runs. The results shown in figures 15-17 were taken from 10 runs of each problem. Both the CRAY-2 and CRAY Y-MP were running the CRAY

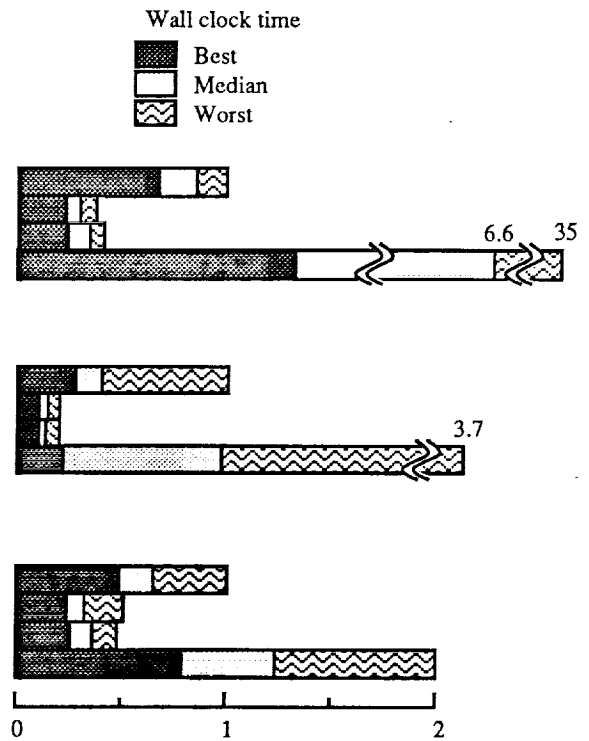
UNICOS 5.1.9 operating system at the time of this study. The CRAY UNICOS 6.0 operating system incorporates changes that should improve the performance of multitasked jobs in a batch environment.

CRAY-2 results. Figure 15 shows the best, median, and worst elapsed times for all three problems from runs on the CRAY-2. One factor that significantly affects the performance of the VBAND and PVBAND implementations in multiuser mode is the memory-swapping size limit. For programs below the memory-swapping size limit, the entire matrix is swapped out of main memory more often than for larger problems when tasks are interrupted. Larger problems are left in memory even though the CPU may be interrupted and dedicated to another user for a time period. This effect can be seen in figure 15 by comparing the CPU and wall clock times for the SRB problem, which is over the swapping size limit, with the HSCT and composite panel problems, which are under the swapping size limit. The best-case wall and CPU times for the SRB problem are nearly identical to the dedicated times for the VBAND method VECTOR. For the smaller problems, the best-case wall times were between two and four times longer than the dedicated wall times for method VECTOR, indicating that the cost of memory swapping significantly increased the wall clock times. The wall clock times for the autotasking and microtasking PVBAND implementations indicate that these parallel implementations significantly reduce wall clock time when compared with the wall clock time for the VBAND VECTOR implementation for all three problems. The FORCE implementations did not reduce CPU time in the multiuser mode, and in most cases they significantly increased wall clock times relative to the VBAND method. The very poor performance of FORCE on the CRAY-2 is the result of the interaction between the high cost of standard FORCE barriers and the scheduling of a fixed number of tasks in a busy environment. This apparently leads to more frequent swapping between memory and disks, thus greatly increasing the wall clock times. The multiuser times for the FORCE implementations on the CRAY-2 were as much as 35 times greater than the corresponding method VECTOR times in the worst case. The autotasking and microtasking implementation times were very close on all three problems, with the autotasking implementations giving the fastest best-case wall clock times on all three problems.

CRAY Y-MP results. On the ARC CRAY Y-MP, the maximum number of CPU's used for the single user's program is set by default to four. Users

Method	Best times: Wall / CPU	Median times: Wall / CPU	Worst times: Wall / CPU
High-Speed Civil Transport aircraft (16 146 equations; Max. semibandwidth = 594; Av. semibandwidth = 319)			
VECTOR	25.64 / 6.21	32.79 / 6.22	38.27 / 6.23
AUTO	8.92 / 7.06	11.51 / 6.97	14.09 / 6.99
MICRO	9.35 / 7.41	13.34 / 7.29	15.54 / 7.06
FORCE	51.17 / 13.88	253.95 / 15.10	1345.16 / 16.20
Composite cross-ply laminate with hole (11 929 equations; Max. semibandwidth = 1597; Av. semibandwidth = 1081)			
VECTOR	98.41 / 47.64	145.80 / 47.68	357.79 / 47.72
AUTO	38.79 / 49.35	51.01 / 49.15	71.16 / 48.95
MICRO	39.23 / 50.03	49.42 / 49.85	70.93 / 49.87
FORCE	79.42 / 55.17	350.31 / 56.95	1318.95 / 54.56
Space Shuttle solid rocket booster (54 780 equations; Max. semibandwidth = 558; Av. semibandwidth = 355)			
VECTOR	24.90 / 24.87	33.14 / 24.86	50.86 / 24.85
AUTO	12.38 / 29.17	16.64 / 28.28	25.74 / 27.88
MICRO	12.99 / 29.95	18.86 / 29.01	24.52 / 28.62
FORCE	40.17 / 51.98	62.54 / 50.07	100.71 / 54.78

(a) Wall clock and CPU times for Choleski factorization for 10 runs each on three example problems.

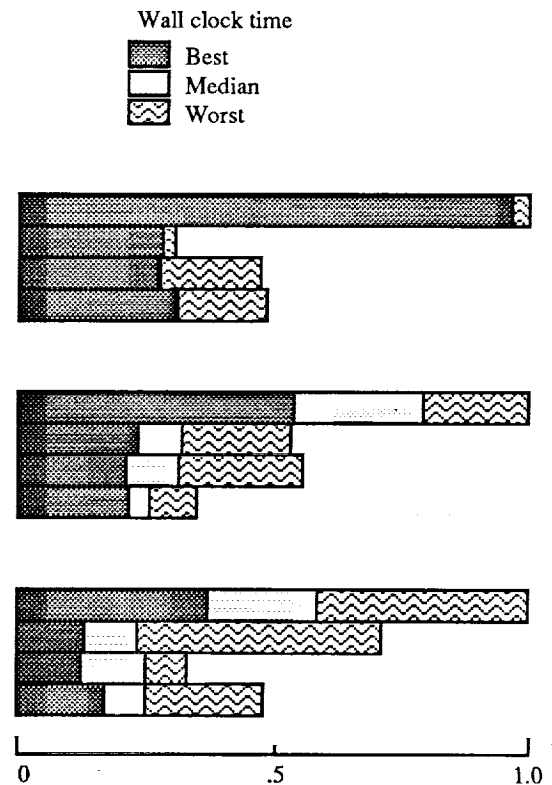


(b) Comparison of wall clock times (normalized).

Figure 15. CRAY-2 factorization times in multiuser mode with a maximum of four CPU's.

Method	Best times: Wall / CPU	Median times: Wall / CPU	Worst times: Wall / CPU
High-Speed Civil Transport aircraft (16 146 equations; Max. semibandwidth = 594; Av. semibandwidth = 319)			
VECTOR	5.60 / 5.58	5.62 / 5.60	5.84 / 5.59
AUTO	1.64 / 6.54	1.66 / 6.55	1.78 / 6.52
MICRO	1.59 / 6.33	1.61 / 6.35	2.76 / 6.28
FORCE	1.77 / 7.05	1.80 / 7.12	2.84 / 7.04
Composite cross-ply laminate with hole (11 929 equations; Max. semibandwidth = 1597; Av. semibandwidth = 1081)			
VECTOR	65.32 / 47.15	96.06 / 47.62	120.87 / 47.39
AUTO	28.35 / 48.92	38.88 / 49.13	64.76 / 48.78
MICRO	25.56 / 48.52	37.84 / 47.91	67.94 / 48.06
FORCE	26.10 / 50.17	31.74 / 49.94	42.71 / 49.84
Space Shuttle solid rocket booster (54 780 equations; Max. semibandwidth = 558; Av. semibandwidth = 355)			
VECTOR	22.15 / 22.10	34.49 / 22.45	58.79 / 22.36
AUTO	7.62 / 25.53	13.72 / 25.54	42.30 / 24.46
MICRO	7.38 / 25.08	14.99 / 24.86	19.76 / 24.40
FORCE	10.26 / 28.99	14.73 / 28.38	28.39 / 28.07

(a) Wall clock and CPU times for Choleski factorization for 10 runs each on three example problems.

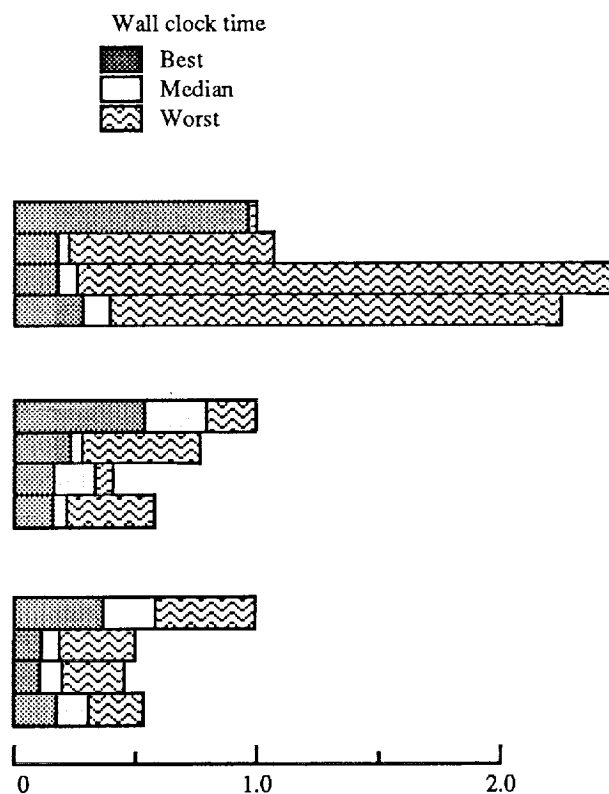


(b) Comparison of wall clock times (normalized).

Figure 16. CRAY Y-MP factorization times in multiuser mode with a maximum of four CPU's.

Method	Best times: Wall / CPU	Median times: Wall / CPU	Worst times: Wall / CPU
High-Speed Civil Transport aircraft (16 146 equations; Max. semibandwidth = 594; Av. semibandwidth = 319)			
VECTOR	5.60 / 5.58	5.62 / 5.60	5.84 / 5.59
AUTO	1.04 / 7.57	1.31 / 7.33	6.21 / 6.86
MICRO	1.03 / 7.31	1.49 / 7.04	14.42 / 6.60
FORCE	1.64 / 11.45	2.27 / 11.50	13.09 / 9.04
Composite cross-ply laminate with hole (11 929 equations; Max. semibandwidth = 1597; Av. semibandwidth = 1081)			
VECTOR	65.32 / 47.15	96.06 / 47.62	120.87 / 47.39
AUTO	28.46 / 50.00	34.62 / 49.61	93.09 / 49.18
MICRO	19.94 / 49.94	39.93 / 48.65	49.86 / 48.36
FORCE	19.50 / 54.01	26.55 / 53.81	70.46 / 52.99
Space Shuttle solid rocket booster (54 780 equations; Max. semibandwidth = 558; Av. semibandwidth = 355)			
VECTOR	22.15 / 22.10	34.49 / 22.45	58.79 / 22.36
AUTO	6.92 / 27.07	11.32 / 27.16	29.66 / 25.22
MICRO	6.46 / 26.87	11.86 / 26.87	26.93 / 25.02
FORCE	10.33 / 38.45	18.31 / 33.91	31.93 / 38.20

(a) Wall clock and CPU times for Choleski factorization for 10 runs each on three example problems.



(b) Comparison of wall clock times (normalized).

Figure 17. CRAY Y-MP factorization times in multiuser mode with a maximum of eight CPU's.

can request a maximum of eight CPU's at run time. Figure 16 gives results from multiuser runs for all three example problems using the default maximum number of four CPU's, and figure 17 gives results for the same problems using a maximum of eight CPU's. Very little improvement in wall clock time is seen in figure 17 compared with the results in figure 16. This result indicates the difficulty of obtaining all eight processors in a busy multiuser environment. The results in both figures 16 and 17 show a substantial reduction in wall clock time by all the PVBAND implementations. A major difference between the CRAY-2 and CRAY Y-MP results is that the CRAY Y-MP wall clock times are much better than the CRAY-2 wall clock times for the FORCE PVBAND implementations. This difference between the implementation of macrotasking on the two CRAY computers is reflected in a comparison of the wall clock times in figures 15, 16, and 17 for the FORCE implementations.

Triangular Solutions

Table 5 shows the CPU times for the solution of the triangular systems for the three structural anal-

ysis problems used in this study. The improvement in computation rate is demonstrated for the HSCT problem by comparing the basic triangular solution times LL1S with the triangular solutions using loop unrolling to level 6, i.e., LL6S. This improvement is largest on the CRAY-2, where the bottleneck due to the single path to memory causes single SAXPY or inner product computations to be much slower than the same operations on the CRAY Y-MP. The reduction in operations due to the zero-checking strategy LL6SZ is also shown in table 5 for the HSCT problem. For all three problems, the percentage of CPU time for the triangular solutions as compared with the factorization time is small. The potential benefit from parallelizing the triangular solutions is small because of the small percentage of total time required for the triangular solutions. For analyses requiring multiple triangular solutions, the benefit of a parallel triangular solution would be far greater.

6. Concluding Remarks

A Choleski method that effectively exploits key architectural features of the CRAY-2 and CRAY Y-MP computers has been described. This method is well suited for parallel processing, and several

Table 5. Forward-and-Backward Solution Times for Three Example Problems

Problem	Method ^a	CPU time (second)	Number of operations	Rate, MFLOPS	Factor time, ^b percent
NASA Langley CRAY-2 computer runs					
HSCT	LL1S	0.27	20 610 072	76	4.3
	LL6S	.19	20 610 072	107	3.1
	LL6SZ	.17	16 210 224	94	2.7
Panel	LL1S	0.45	51 583 442	114	0.9
	LL6S	.37	51 583 442	138	.8
	LL6SZ	.38	51 583 442	136	.8
SRB	LL1S	0.86	77 979 552	91	3.5
	LL6S	.71	77 979 552	110	2.9
	LL6SZ	.68	77 494 584	105	2.7
NASA Ames CRAY Y-MP computer runs					
HSCT	LL1S	0.14	20 610 072	147	2.5
	LL6S	.14	20 610 072	147	2.5
	LL6SZ	.12	16 210 224	94	2.2
Panel	LL1S	0.24	51 583 442	215	0.5
	LL6S	.26	51 583 442	198	.6
	LL6SZ	.25	51 583 442	206	.5
SRB	LL1S	0.50	77 979 552	156	2.3
	LL6S	.50	77 979 552	156	2.3
	LL6SZ	.49	71 494 584	146	2.2

^aLL1S: forward-backward solution, no loop unrolling.

LL6S: forward-backward solution, loop-unrolling level 6

LL6SZ: forward-backward solution, loop-unrolling level 6 with zero checking.

^bPercentage calculation uses CPU time for VECTOR method.

implementations of the PVBAND method have been presented comparing the use of macrotasking, autotasking, and microtasking. The differences between implementing the PVBAND method with a fixed number of tasks (i.e., as in using FORCE) and implementing the PVBAND method using processors as they become available at the time of program execution were discussed.

Three structural analysis example problems were used to compare the parallel implementations of the PVBAND Choleski method. The example problems were all generated using the COMET finite element software system and are representative of a wide class of large-scale problems for which the variable-band Choleski method is very efficient. The PVBAND implementations that use autotasking and microtasking were shown to be more effective on the CRAY multiprocessor supercomputers than the FORCE implementations that used macrotasking. Dedicated single-user runs on both the CRAY-2 and CRAY Y-MP computers demonstrated maximum computation rates of over 250 MFLOPS per

processor and maximum rates of over 1 GIGAFLOP on the CRAY-2 using four processors and over 2 GIGAFLOPS on the CRAY Y-MP using eight processors. Parallel speedups for the PVBAND implementations compared favorably with maximum theoretical speedups in all cases except for the FORCE implementations on the CRAY-2. Though computation rates and parallel speedups were slightly higher on the CRAY Y-MP than on the CRAY-2, the VBAND and PVBAND implementations are efficient on both computers.

The results presented for both dedicated and multiuser environments indicate that parallel processing is effective in both cases in reducing wall clock time. The cost of parallel processing runs will always be higher if based on total CPU (central processing unit) time, as indicated by increased CPU times for the PVBAND implementations compared with the VBAND implementations. The comparison of timing results for autotasking and microtasking implementations with the FORCE implementations demonstrates the importance of dynamically using available processors in a multiuser environment.

Appendix

VBAND and PVBAND FORTRAN Listings

In this appendix, FORTRAN listings of several subroutines used to implement the VBAND and PVBAND methods are given and briefly described. Figures A1 and A2 contain the listings for two subroutines that are used to convert a symmetric, sparse matrix data structure to the variable-band matrix data structure used for the VBAND and PVBAND Choleski methods. Figures A3 and A4 contain listings for the autotasked and FORCE implementations of the PVBAND method, respectively. The two PVBAND implementations illustrate the two approaches described in section 3 under the subsection "CRAY-2 local memory considerations." Figure A5 contains a listing of the forward-and-backward solutions used in this study. Each figure includes line numbers with comments to explain major sections and key variables within each subroutine. A brief description of each subroutine follows.

Subroutine *sp2vb*, shown in figure A1, accepts as input the sparse matrix pointer arrays and computes new pointer arrays for the variable-band data structure. The sparse matrix pointer arrays contain the number of nonzero coefficients in each column of the lower triangular matrix as well as an integer row index for each nonzero coefficient. The variable-band pointer arrays contain the lengths of each column in the lower triangular part of the variable-band matrix and the starting position of each column in a single-dimensioned array. In addition, the row index values from the sparse data structure are converted to offsets that give the location of each nonzero coefficient from the input sparse matrix within the variable-band matrix array. The column lengths for the variable-band matrix data structure are initially computed in lines 56–61 in figure A1. The initial lengths are increased where necessary in lines 67–69 to account for possible additional fill-in during factorization. A second adjustment is made to the column lengths in lines 77–92 to allow for loop unrolling. After adjustments to the column lengths, the starting position of each column within a single-dimensioned array is determined in lines 103–105. Finally, the row index pointers for each nonzero coefficient, stored in array *ia()*, are changed to location pointers in lines 113–118. These location pointers give the position of each nonzero coefficient within the single-dimensioned variable-band matrix array. Subroutine *sp2vb* also determines the amount of memory required for the variable-band matrix array and re-

turns that value along with the maximum bandwidth and average bandwidth as arguments.

Subroutine *convert*, shown in figure A2, is called after subroutine *sp2vb* to build the single-dimensioned array used to store the variable-band matrix. The array is initially filled with zeros in lines 29 and 30 in figure A2. Then, in lines 34–41, the nonzero coefficients from the sparse matrix storage array are assigned to the variable-band matrix array as determined by the locations stored in array *ia()*. Though initially many zeros are stored in the variable-band array, many are changed to nonzero values during matrix factorization. The use of two subroutines allows for efficient dynamic memory allocation of the variable-band matrix array after the exact length of the array is determined by subroutine *sp2vb*.

Subroutine *pvband*, shown in figure A3, is the autotasked version of method PVBAND used to obtain the results presented in this paper. The subroutine uses loop unrolling, local memory (on the CRAY-2), and zero checking as described in section 3 along with autotasking compiler directives to implement a fast, efficient parallel/vector version of the PVBAND method. The FORTRAN listing of the sequential vectorized version of the VBAND method is not given in this appendix, but it is easily obtained from the listing in figure A3 by deleting both the parallel directives and the sections of code used by additional processes to copy completed columns into local memory. For example, lines 53–61 copy a completed column of the variable-band matrix into local memory and are not required for the sequential FORTRAN version. Variable *idone* is used to facilitate the copying of columns into local memory in the autotasking implementation and is not required in the sequential FORTRAN version. In figure A3, lines 33–244 correspond to the barrier region in the PVBAND method described in section 3. Each of six columns, k through $k + 5$, are completed within an autotasking control structure (*case* directive). This sequential portion of the algorithm accounts for a very small percentage of the total computations. (See table 4.) Lines 253–284 comprise the parallel portion of the PVBAND method. The outer loop iterations of DO loop 61 (line 254) are completed in parallel, each performing a column update using the six-term SAXPY shown in lines 274–279. The zero-checking option (lines 257–271) skips the SAXPY update whenever all six scalar multipliers in DO loop 610 are all equal to zero.

The FORCE subroutine *Forcesub pvband*, shown in figure A4, illustrates the use of the FORCE language syntax to implement the PVBAND method. The variable *NPROC* in line 1 of figure A4 is the

fixed number of tasks, determined at run time, that executes the parallel subroutine. Variable ME, also in line 1, is the task identification variable and will have a unique value between 1 and NPROC. In the FORCE variable declaration portion ending with line 13, all variables used within subroutine *pband* are declared private and all argument variables are declared as shared or private in the calling program. The sequential portion of the PVBAND method in lines 40-159 is computed within a single FORCE barrier. The six columns computed within the barrier region by only one task are copied into the local memory of all remaining tasks in lines 162-179. The parallel update portion in lines 183-222 is implemented

by the FORCE *Presched DO* statement in line 190. The FORCE subroutine in figure A4 will run as is on any parallel computer for which FORCE is installed.

Figure A5 contains the FORTRAN listing of the forward and backward solutions used for the VBAND and PVBAND methods. The subroutine in figure A5 uses loop unrolling and zero checking. The main computation in the forward solution (lines 17-130) is the six-term SAXPY loop (lines 76-84). The main computations in the backward solution (lines 132-253) are the six inner product computations (lines 217-224). The remaining computations are sequential computations.

```

1:      subroutine sp2vb(spptr,splen,ia,n,nroll,vbptr,vblen,
2:      +                ibw,iavbw,ialth)
3:
4:      integer spptr(*),splen(*),ia(*),row,colm
5:      integer vbptr(*),vblen(*)
6:
7: c *** This routine is called to create pointer arrays which are
8: c *** subsequently used by routine convert to convert an input
9: c *** matrix stored in a sparse data structure to a variable-band
10: c *** matrix data structure. Major steps are: 1) Create arrays vblen and
11: c *** vbptr, 2) Change row index values stored for each lower triangular
12: c *** coefficient to location pointers for each coefficient.
13: c ***
14: c *** INPUT ARGUMENTS :
15: c *** spptr() - starting position for each column of sparse
16: c ***           matrix within a single-dimensioned array
17: c *** splen() - number of nonzero coefficients in each column of
18: c ***           sparse matrix
19: c *** ia()    - row index for each nonzero below the main diagonal
20: c ***           in input sparse matrix. The row index values are
21: c ***           arranged in the same order as the nonzero
22: c ***           coefficients; by columns.
23: c *** n      - number of equations in the sparse matrix
24: c *** nroll  - level of loop unrolling used in the factorization
25: c ***           routine (nroll > 1 requires adjustments to column
26: c ***           lengths)
27: c ***
28: c *** OUTPUT ARGUMENTS :
29: c *** ia()    - return values in this array contain an offset for
30: c ***           each nonzero coefficient in sparse matrix which
31: c ***           will be used to expand the sparse storage into
32: c ***           variable-band storage.
33: c *** vbptr() - starting position for each column of variable-band
34: c ***           matrix within a single-dimensioned array
35: c *** vblen() - length of each column of variable-band matrix;
36: c ***           includes diagonal and allows for zeros down to the
37: c ***           last nonzero coefficient in each column plus
38: c ***           possible adjustments for fill past the last nonzero
39: c ***           and loop-unrolling adjustments.
40: c *** ibw    - maximum semibandwidth (includes diagonal) after
41: c ***           all adjustments.
42: c *** iavbw  - average semibandwidth (total storage required
43: c ***           divided by number of equations)
44: c *** ialth  - total memory required for variable-band matrix
45: c ***           coefficient array, includes diagonal
46:

```

Figure A1. FORTRAN listing for subroutine *sp2bv*.

```

47: c *** Compute the lengths of each variable-band column
48: c *** from the main diagonal to the last nonzero coefficient
49: c *** in the lower triangular part of the matrix. The main
50: c *** diagonal is part of the column length.
51: c *** Loop 1 assumes that nonzero coefficients within each
52: c *** column of sparse input data structure are ordered by increasing
53: c *** row indexes. Pointer irx points to the last nonzero row index
54: c *** in each column.
55:
56:     vblen(n)=1
57:     do 1 colm=1,n-1
58:         irx=spptr(colm+1)-1
59:         row=ia(irx)
60:         vblen(colm)=row-colm+1
61: 1     continue
62:
63: c *** Adjust column lengths for possible fill during factorization;
64: c *** each column must extend at least to the last row
65: c *** of the previous column.
66:
67:     do 2 colm=2,n
68:         vblen(colm)=max(vblen(colm-1)-1,vblen(colm))
69: 2     continue
70:
71: c *** Adjust for loop unrolling if nroll > 1 ;
72: c *** groups of nroll columns must end in the same row.
73: c *** Zeros are effectively added to the ends of columns
74: c *** as required to enforce this condition by increasing
75: c *** the length of the column (vblen(colm)).
76:
77:     if (nroll.gt.1) then
78:         lastc=n-mod(n,nroll)
79:         do 30 colm=nroll,lastc,nroll
80:             lth=vblen(colm)
81:             do 30 i=1,nroll-1
82:                 vblen(colm-i)=lth+i
83: 30     continue
84:
85: c *** Fix up the end columns whenever n mod nroll <> 0
86: c *** The remainder columns are full.
87:
88:     iadd=2
89:     do 31 row=n-1,lastc+1,-1
90:         vblen(row)=iadd
91: 31     iadd=iadd+1
92:     end if

```

Figure A1. Continued.

```

93:
94: c *** Add up the column lengths and form vbptr array;
95: c *** also compute maximum semibandwidth (including diagonal).
96:
97:     ialth=0
98:     ibw=0
99:     do 40 colm=1,n
100:         ibw=max(ibw,vblen(colm))
101:     40     ialth=ialth+vblen(colm)
102:
103:     vbptr(1)=1
104:     do 41 colm=2,n
105:     41     vbptr(colm)=vbptr(colm-1)+vblen(colm-1)
106:     iavbw=ialth/n
107:
108: c *** Change the row index values from
109: c *** the sparse column storage in ia() to the
110: c *** locations where each nonzero coefficient is stored
111: c *** within the variable-band matrix single-dimensioned array.
112:
113:     do 51 colm=1,n-1
114:         do 51 k=spptr(colm),spptr(colm+1)-1
115:             row = ia(k)
116:             loc = vbptr(colm) + row - colm
117:             ia(k) = loc
118:     51     continue
119:
120:     return
121:     end

```

Figure A1. Concluded.

```

1:      subroutine convert(coefs,diag,ia,vbptr,ncoef,ialth,n,a)
2:      integer ia(*),row,colm
3:      integer vbptr(*)
4:      real a(*),coefs(*),diag(*)
5:
6: c *** This routine converts sparse data structure into variable-band
7: c *** data structure. Routine sp2vb must be called first to
8: c *** create input array vbptr() and modify array ia(*).
9: c ***
10: c *** INPUT ARGUMENTS:
11: c *** coefs() - single-dimensional array containing nonzero
12: c ***           coefficients of sparse matrix, arranged by columns,
13: c ***           lower triangular coefficients only
14: c *** diag() - main diagonal of sparse input matrix
15: c *** ia() - for each coefficient in array coefs() contains an offset
16: c ***           into variable-band matrix array, a()
17: c *** vbptr() - starting position of each column in the variable-band
18: c ***           matrix coefficient array a().
19: c *** ncoef - number of nonzero in array coefs()
20: c *** ialth - length of the variable-band array
21: c *** n - number of equations in sparse and variable-band matrices
22: c ***
23: c *** OUTPUT ARGUMENTS:
24: c *** a() - single-dimensional array containing the coefficients
25: c ***           of the variable-band output matrix including zeros
26: c ***           added within columns as appropriate.
27:
28: c *** Zero out space for vband storage of coefficients.
29:      do 10 i=1,ialth
30: 10      a(i)=0.0
31:
32: c *** Assign nonzero coefficients to locations in variable-band
33: c *** matrix array as designated by array ia().
34:      do 20 i=1,ncoef
35: 20      a(ia(i))=coefs(i)
36:
37: c *** Place main diagonal coefficients at beginning of each column
38: c *** in variable-band array.
39:      do 30 colm=1,n
40:          a(vbptr(colm))=diag(colm)
41: 30      continue
42:
43:      return
44:      end

```

Figure A2. FORTRAN listing for subroutine *convert*.


```

1:      subroutine pvband(a,alth,colptr,collth,n,nopsf)
2:      integer k,i,j,lastr,lth,t1,iadd
3:      integer ist1,ist2,ist3,ist4,ist5,ist6
4:      integer i1,i2,i3,i4,i5,i6
5:      integer n,alth,nopsf
6:      integer colptr(n),collth(n)
7:      real a(alth)
8:
9:      parameter (lmsize=9600)
10:     parameter (l6=lmsize/6)
11:     real m1(16),m2(16),m3(16),m4(16),m5(16),m6(16)
12:     common /lmem/ idone,m1,m2,m3,m4,m5,m6
13:
14: cdir$ regfile lmem
15:
16: cmic$ parallel shared(a,alth,colptr,collth,n,nopsf) private(k,i,j,
17: cmic$1  lastr,lth,t1,iadd,ist1,ist2,ist3,ist4,ist5,ist6,
18: cmic$2  i1,i2,i3,i4,i5,i6,idone,m1,m2,m3,m4,m5,m6,ict,itst)
19:
20: c *** Variable idone is a private variable and is used to
21: c *** determine which processors must copy multipliers into
22: c *** local memory. The processor which computes a multiplier
23: c *** already has that column in local memory and proceeds
24: c *** to the next code section, skipping the copy segment.
25:     idone=0
26:     k=1
27: c *** Counting variable nopsf is shared and must be zeroed prior
28: c *** to calling this routine. The variable is used to count
29: c *** only serial operations.
30:
31: 999  continue
32:
33: c *** Begin jki portion: Complete 6 columns of L; k through k+5.
34: c *** The 6 columns are used as multiplier columns in the
35: c *** update section.
36:
37: c *** Finish the kth multiplier.
38:     ist1=colptr(k)
39:     i1=ist1+1
40:
41: cmic$ case
42:     a(ist1)=1.0/sqrt(a(ist1))
43:     t1=1
44: cdir$ ivdep

```

Figure A3. FORTRAN listing for autotasked subroutine *pvband*.

```

45:      do 10 i=i1,i1+collth(k)-2
46:          a(i)=a(i)*a(ist1)
47:          m1(t1)=a(i)
48:          t1=t1+1
49:  10    continue
50:      idone=k
51: cmic$ end case
52:
53: c *** Everyone else copy the kth column into local memory.
54:      if (idone.lt.k) then
55:          t1=1
56:          do 111 i=i1,i1+collth(k)-2
57:              m1(t1)=a(i)
58:              t1=t1+1
59:  111    continue
60:      idone=k
61:      end if
62:
63: c *** Update, then finish column k+1.
64:      ist2=colptr(k+1)
65:      i2=ist2+1
66:
67: cmic$ case
68:      t1=1
69:      ict=0
70: cdir$ ivdep
71:      do 11 i=ist2,ist2+collth(k+1)-1
72:          a(i)=a(i)-m1(t1)*m1(t1+ict)
73:          ict=ict+1
74:  11    continue
75:
76: c *** Finish the (k+1)st multiplier.
77:      a(ist2)=1.0/sqrt(a(ist2))
78:      t1=2
79: cdir$ ivdep
80:      do 20 i=i2,i2+collth(k+1)-2
81:          a(i)=a(i)*a(ist2)
82:          m2(t1)=a(i)
83:          t1=t1+1
84:  20    continue
85:      idone=k+1
86: cmic$ end case

```

Figure A3. Continued.

```

87:
88: c *** Everyone else copy column k+1 into local memory.
89:     if (idone.lt.k+1) then
90:         t1=2
91:         do 211 i=i2,i2+collth(k+1)-2
92:             m2(t1)=a(i)
93:             t1=t1+1
94: 211     continue
95:         idone=k+1
96:     end if
97:
98: c *** Update, then finish column k+2.
99:     ist3=colptr(k+2)
100:    i3=ist3+1
101:
102: cmic$ case
103:     t1=2
104:     ict=0
105: cdir$ ivdep
106:     do 21 i=ist3,ist3+collth(k+2)-1
107:         a(i)=a(i)-m1(t1)*m1(t1+ict)-m2(t1)*m2(t1+ict)
108:         ict=ict+1
109: 21     continue
110:
111: c *** Finish the (k+2)nd multiplier.
112:     a(ist3)=1.0/sqrt(a(ist3))
113:     t1=3
114: cdir$ ivdep
115:     do 30 i=i3,i3+collth(k+2)-2
116:         a(i)=a(i)*a(ist3)
117:         m3(t1)=a(i)
118:         t1=t1+1
119: 30     continue
120:     idone=k+2
121: cmic$ end case
122:
123: c *** Everyone else copy the column k+2 into local memory.
124:     if (idone.lt.k+2) then
125:         t1=3
126:         do 311 i=i3,i3+collth(k+2)-2
127:             m3(t1)=a(i)
128:             t1=t1+1
129: 311     continue
130:         idone=k+2
131:     endif
132:
133: c *** Update, then finish column k+3.

```

Figure A3. Continued.

```

134:      ist4=colptr(k+3)
135:      i4=ist4+1
136:
137: cmic$ case
138:      t1=3
139:      ict=0
140: cdir$ ivdep
141:      do 31 i=ist4,ist4+collth(k+3)-1
142:          a(i)=a(i)-m1(t1)*m1(t1+ict)-m2(t1)*m2(t1+ict)-m3(t1)*m3(t1+ict)
143:          ict=ict+1
144: 31      continue
145:
146: c *** Finish the (k+3)rd multiplier.
147:      a(ist4)=1.0/sqrt(a(ist4))
148:      t1=4
149: cdir$ ivdep
150:      do 40 i=i4,i4+collth(k+3)-2
151:          a(i)=a(i)*a(ist4)
152:          m4(t1)=a(i)
153:          t1=t1+1
154: 40      continue
155:          idone=k+3
156: cmic$ end case
157:
158: c *** Everyone else copy the column k+3 into local memory.
159:      if (idone.lt.k+3) then
160:          t1=4
161:          do 411 i=i4,i4+collth(k+3)-2
162:              m4(t1)=a(i)
163:              t1=t1+1
164: 411      continue
165:          idone=k+3
166:      endif
167:
168: c *** Update, then finish column k+4.
169:      ist5=colptr(k+4)
170:      i5=ist5+1
171:
172: cmic$ case
173:      t1=4
174:      ict=0
175: cdir$ ivdep
176:      do 41 i=ist5,ist5+collth(k+4)-1
177:          a(i)=a(i)-m1(t1)*m1(t1+ict)-m2(t1)*m2(t1+ict)-m3(t1)*m3(t1+ict)
178:      +          -m4(t1)*m4(t1+ict)
179:          ict=ict+1
180: 41      continue
181:

```

Figure A3. Continued.

```

182: c *** Finish the (k+4)th multiplier.
183:     a(ist5)=1.0/sqrt(a(ist5))
184:     t1=5
185: cdir$ ivdep
186:     do 50 i=i5,i5+collth(k+4)-2
187:         a(i)=a(i)*a(ist5)
188:         m5(t1)=a(i)
189:         t1=t1+1
190: 50     continue
191:     idone=k+4
192: cmic$ end case
193:
194: c *** Everyone else copy the column k+4 into local memory.
195:     if (idone.lt.k+4) then
196:         t1=5
197:         do 511 i=i5,i5+collth(k+4)-2
198:             m5(t1)=a(i)
199:             t1=t1+1
200: 511     continue
201:         idone=k+4
202:     endif
203:
204: c *** Update, then finish column k+5.
205:     ist6=colptr(k+5)
206:     i6=ist6
207:
208: cmic$ case
209:     t1=5
210:     ict=0
211: cdir$ ivdep
212:     do 51 i=ist6,ist6+collth(k+5)-1
213:         a(i)=a(i)-m1(t1)*m1(t1+ict)-m2(t1)*m2(t1+ict)-m3(t1)*m3(t1+ict)
214:     +     -m4(t1)*m4(t1+ict)-m5(t1)*m5(t1+ict)
215:         ict=ict+1
216: 51     continue
217:
218: c *** Finish the (k+5)th multiplier.
219:     a(ist6)=1.0/sqrt(a(ist6))
220:     t1=6
221: cdir$ ivdep
222:     do 60 i=ist6+1,ist6+collth(k+5)-1
223:         a(i)=a(i)*a(ist6)
224:         m6(t1)=a(i)
225:         t1=t1+1
226: 60     continue

```

Figure A3. Continued.

```

227:         idone=k+5
228:
229: c *** Count all floating point vector operations for finishing
230: c *** this group of 6 multiplier columns.
231:         nopsf=nopsf+collth(k+5)*36 + 55
232: cmic$ end case
233:
234: c *** Everyone else copy column k+5 into local memory.
235:         if (idone.lt.k+5) then
236:             t1=6
237:             do 6111 i=ist6+1,ist6+collth(k+5)-1
238:                 m6(t1)=a(i)
239:                 t1=t1+1
240: 6111         continue
241:             idone=k+5
242:         endif
243:
244: c *** End jki portion: 6 columns of L completed.
245:
246: c *** Begin kji portion: update columns k+6 thru lastr
247: c *** of matrix using the 6 completed multiplier columns.
248:
249: c *** Variable lastr corresponds to the last row in column k
250: c *** that is stored in the variable-band format.
251:         lastr=k+collth(k)-1
252:
253: cmic$ do parallel
254:         do 61 j=k+6,lastr
255:             t1=j-k
256:             itst=t1
257: c *** Begin full zero checking
258:             if (m1(itst).eq.0.0) then
259:                 if (m2(itst).eq.0.0) then
260:                     if (m3(itst).eq.0.0) then
261:                         if (m4(itst).eq.0.0) then
262:                             if (m5(itst).eq.0.0) then
263:                                 if (m6(itst).eq.0.0) then
264:                                     goto 611
265:                                 endif
266:                             endif
267:                         endif
268:                     endif
269:                 endif
270:             endif
271: c *** End zero checking.
272:         lth=lastr-j

```

Figure A3. Continued.

```

273: cdir$ ivdep
274:         do 610 i=colptr(j),colptr(j)+lth
275:             a(i)=a(i)-m1(itst)*m1(t1)-m2(itst)*m2(t1)
276:         +             -m3(itst)*m3(t1)-m4(itst)*m4(t1)
277:         +             -m5(itst)*m5(t1)-m6(itst)*m6(t1)
278:             t1=t1+1
279: 610     continue
280:
281: 611     continue
282: 61     continue
283:
284: c *** End kji portion: increment k and begin next kji portion
285: c *** if k is less than n-6.
286:
287:         k=k+6
288:         if (k.lt.n-6) goto 999
289:
290: c *** Finish any remaining columns of L. There are up to 7
291: c *** columns remaining. The last 7 columns are explicitly
292: c *** computed without loop overhead, etc.
293:
294: cmic$ case
295:         goto (100,110,120,130,140,150,160),(n-k+1)
296:
297: 160     continue
298: c *** 6 columns left to update
299:         ist1=colptr(k)
300:         a(ist1)=1.0/sqrt(a(ist1))
301:         a(ist1+1)=a(ist1+1)*a(ist1)
302:         a(ist1+2)=a(ist1+2)*a(ist1)
303:         a(ist1+3)=a(ist1+3)*a(ist1)
304:         a(ist1+4)=a(ist1+4)*a(ist1)
305:         a(ist1+5)=a(ist1+5)*a(ist1)
306:         a(ist1+6)=a(ist1+6)*a(ist1)
307:         ist2=colptr(k+1)
308:         a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
309:         a(ist2+1)=a(ist2+1)-a(ist1+1)*a(ist1+2)
310:         a(ist2+2)=a(ist2+2)-a(ist1+1)*a(ist1+3)
311:         a(ist2+3)=a(ist2+3)-a(ist1+1)*a(ist1+4)
312:         a(ist2+4)=a(ist2+4)-a(ist1+1)*a(ist1+5)
313:         a(ist2+5)=a(ist2+5)-a(ist1+1)*a(ist1+6)
314:         ist3=colptr(k+2)
315:         a(ist3)=a(ist3)-a(ist1+2)*a(ist1+2)
316:         a(ist3+1)=a(ist3+1)-a(ist1+2)*a(ist1+3)
317:         a(ist3+2)=a(ist3+2)-a(ist1+2)*a(ist1+4)
318:         a(ist3+3)=a(ist3+3)-a(ist1+2)*a(ist1+5)
319:         a(ist3+4)=a(ist3+4)-a(ist1+2)*a(ist1+6)

```

Figure A3. Continued.

```

320:     ist4=colptr(k+3)
321:     a(ist4)=a(ist4)-a(ist1+3)*a(ist1+3)
322:     a(ist4+1)=a(ist4+1)-a(ist1+3)*a(ist1+4)
323:     a(ist4+2)=a(ist4+2)-a(ist1+3)*a(ist1+5)
324:     a(ist4+3)=a(ist4+3)-a(ist1+3)*a(ist1+6)
325:     ist5=colptr(k+4)
326:     a(ist5)=a(ist5)-a(ist1+4)*a(ist1+4)
327:     a(ist5+1)=a(ist5+1)-a(ist1+4)*a(ist1+5)
328:     a(ist5+2)=a(ist5+2)-a(ist1+4)*a(ist1+6)
329:     ist6=colptr(k+5)
330:     a(ist6)=a(ist6)-a(ist1+5)*a(ist1+5)
331:     a(ist6+1)=a(ist6+1)-a(ist1+5)*a(ist1+6)
332:     a(colptr(k+6))=a(colptr(k+6))-a(ist1+6)*a(ist1+6)
333:     k=k+1
334:     nopsf=nopsf+49
335:
336: 150  continue
337: c *** 5 columns left to update
338:     ist1=colptr(k)
339:     a(ist1)=1.0/sqrt(a(ist1))
340:     a(ist1+1)=a(ist1+1)*a(ist1)
341:     a(ist1+2)=a(ist1+2)*a(ist1)
342:     a(ist1+3)=a(ist1+3)*a(ist1)
343:     a(ist1+4)=a(ist1+4)*a(ist1)
344:     a(ist1+5)=a(ist1+5)*a(ist1)
345:     ist2=colptr(k+1)
346:     a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
347:     a(ist2+1)=a(ist2+1)-a(ist1+1)*a(ist1+2)
348:     a(ist2+2)=a(ist2+2)-a(ist1+1)*a(ist1+3)
349:     a(ist2+3)=a(ist2+3)-a(ist1+1)*a(ist1+4)
350:     a(ist2+4)=a(ist2+4)-a(ist1+1)*a(ist1+5)
351:     ist3=colptr(k+2)
352:     a(ist3)=a(ist3)-a(ist1+2)*a(ist1+2)
353:     a(ist3+1)=a(ist3+1)-a(ist1+2)*a(ist1+3)
354:     a(ist3+2)=a(ist3+2)-a(ist1+2)*a(ist1+4)
355:     a(ist3+3)=a(ist3+3)-a(ist1+2)*a(ist1+5)
356:     ist4=colptr(k+3)
357:     a(ist4)=a(ist4)-a(ist1+3)*a(ist1+3)
358:     a(ist4+1)=a(ist4+1)-a(ist1+3)*a(ist1+4)
359:     a(ist4+2)=a(ist4+2)-a(ist1+3)*a(ist1+5)
360:     ist5=colptr(k+4)
361:     a(ist5)=a(ist5)-a(ist1+4)*a(ist1+4)
362:     a(ist5+1)=a(ist5+1)-a(ist1+4)*a(ist1+5)
363:     a(colptr(k+5))=a(colptr(k+5))-a(ist1+5)*a(ist1+5)

```

Figure A3. Continued.


```

364:      k=k+1
365:      nopsf=nopsf+36
366:
367:  140  continue
368: c ***  4 columns left to update
369:      ist1=colptr(k)
370:      a(ist1)=1.0/sqrt(a(ist1))
371:      a(ist1+1)=a(ist1+1)*a(ist1)
372:      a(ist1+2)=a(ist1+2)*a(ist1)
373:      a(ist1+3)=a(ist1+3)*a(ist1)
374:      a(ist1+4)=a(ist1+4)*a(ist1)
375:      ist2=colptr(k+1)
376:      a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
377:      a(ist2+1)=a(ist2+1)-a(ist1+1)*a(ist1+2)
378:      a(ist2+2)=a(ist2+2)-a(ist1+1)*a(ist1+3)
379:      a(ist2+3)=a(ist2+3)-a(ist1+1)*a(ist1+4)
380:      ist3=colptr(k+2)
381:      a(ist3)=a(ist3)-a(ist1+2)*a(ist1+2)
382:      a(ist3+1)=a(ist3+1)-a(ist1+2)*a(ist1+3)
383:      a(ist3+2)=a(ist3+2)-a(ist1+2)*a(ist1+4)
384:      ist4=colptr(k+3)
385:      a(ist4)=a(ist4)-a(ist1+3)*a(ist1+3)
386:      a(ist4+1)=a(ist4+1)-a(ist1+3)*a(ist1+4)
387:      a(colptr(k+4))=a(colptr(k+4))-a(ist1+4)*a(ist1+4)
388:      k=k+1
389:      nopsf=nopsf+25
390:
391:  130  continue
392: c ***  3 columns left to update
393:      ist1=colptr(k)
394:      a(ist1)=1.0/sqrt(a(ist1))
395:      a(ist1+1)=a(ist1+1)*a(ist1)
396:      a(ist1+2)=a(ist1+2)*a(ist1)
397:      a(ist1+3)=a(ist1+3)*a(ist1)
398:      ist2=colptr(k+1)
399:      a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
400:      a(ist2+1)=a(ist2+1)-a(ist1+1)*a(ist1+2)
401:      a(ist2+2)=a(ist2+2)-a(ist1+1)*a(ist1+3)
402:      ist3=colptr(k+2)
403:      a(ist3)=a(ist3)-a(ist1+2)*a(ist1+2)
404:      a(ist3+1)=a(ist3+1)-a(ist1+2)*a(ist1+3)
405:      a(colptr(k+3))=a(colptr(k+3))-a(ist1+3)*a(ist1+3)
406:      k=k+1
407:      nopsf=nopsf+16
408:

```

Figure A3. Continued.

```

409: 120  continue
410: c *** 2 columns left to update
411:      ist1=colptr(k)
412:      a(ist1)=1.0/sqrt(a(ist1))
413:      a(ist1+1)=a(ist1+1)*a(ist1)
414:      a(ist1+2)=a(ist1+2)*a(ist1)
415:      ist2=colptr(k+1)
416:      a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
417:      a(ist2+1)=a(ist2+1)-a(ist1+1)*a(ist1+2)
418:      a(colptr(k+2))=a(colptr(k+2))-a(ist1+2)*a(ist1+2)
419:      k=k+1
420:      nopsf=nopsf+9
421:
422: 110  continue
423: c *** 1 column left to update
424:      ist1=colptr(k)
425:      a(ist1)=1.0/sqrt(a(ist1))
426:      a(ist1+1)=a(ist1+1)*a(ist1)
427:      ist2=colptr(k+1)
428:      a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
429:      nopsf=nopsf+4
430:
431: 100  continue
432: c *** 0 columns left to update
433:      a(colptr(n))=1.0/sqrt(a(colptr(n)))
434:      nopsf=nopsf+1
435: cmic$ end case
436: cmic$ end parallel
437:
438:      return
439:      end

```

Figure A3. Concluded.

```

1:      Forcesub pvband(a,alth,colptr,collth,n,nopsf) of NPROC ident ME
2:      Private integer k,i,j,lastr,lth,t1,iadd
3:      Private integer ist1,ist2,ist3,ist4,ist5,ist6
4:      Private integer i1,i2,i3,i4,i5,i6,ict,itst
5:      integer n,alth,nopsf
6:      integer colptr(n),collth(n)
7:      real a(alth)
8:
9:      parameter (lmsize=9600)
10:     parameter (l6=lmsize/6)
11:     Private real m1(16),m2(16),m3(16),m4(16),m5(16),m6(16)
12:     common /lmem/ idone,m1,m2,m3,m4,m5,m6
13:     End declarations
14:
15: cdir$ regfile lmem
16:
17:
18: c *** Variable idone is a private variable and is used to
19: c *** determine which processors must copy multipliers into
20: c *** local memory. The processor which computes a multiplier
21: c *** already has that column in local memory and proceeds
22: c *** to the next code section, skipping the copy segment.
23:     idone=0
24:     k=1
25:     nopsf=0
26: c *** Counting variable nopsf is private and is used to count
27: c *** the number of operations used by each process.
28:
29: 999  continue
30:
31: c *** Begin jki portion: Complete 6 columns of L; k through k+5.
32: c *** The 6 columns are used as multiplier columns in the
33: c *** update section.
34:
35: c *** Finish the kth multiplier.
36:     ist1=colptr(k)
37:     i1=ist1+1
38:
39:     Barrier
40:     a(ist1)=1.0/sqrt(a(ist1))
41:     t1=1
42: cdir$ ivdep
43:     do 10 i=i1,i1+collth(k)-2
44:         a(i)=a(i)*a(ist1)
45:         m1(t1)=a(i)
46:         t1=t1+1
47: 10  continue

```

Figure A4. FORTRAN listing for FORCE subroutine *pvband*.

```

48:
49: c *** Update, then finish column k+1.
50:     ist2=colptr(k+1)
51:     i2=ist2+1
52:     t1=1
53:     ict=0
54: cdir$ ivdep
55:     do 11 i=ist2,ist2+collth(k+1)-1
56:         a(i)=a(i)-m1(t1)*m1(t1+ict)
57:         ict=ict+1
58: 11     continue
59:
60: c *** Finish the (k+1)st multiplier.
61:     a(ist2)=1.0/sqrt(a(ist2))
62:     t1=2
63: cdir$ ivdep
64:     do 20 i=i2,i2+collth(k+1)-2
65:         a(i)=a(i)*a(ist2)
66:         m2(t1)=a(i)
67:         t1=t1+1
68: 20     continue
69:
70: c *** Update, then finish column k+2.
71:     ist3=colptr(k+2)
72:     i3=ist3+1
73:     t1=2
74:     ict=0
75: cdir$ ivdep
76:     do 21 i=ist3,ist3+collth(k+2)-1
77:         a(i)=a(i)-m1(t1)*m1(t1+ict)-m2(t1)*m2(t1+ict)
78:         ict=ict+1
79: 21     continue
80:
81: c *** Finish the (k+2)nd multiplier.
82:     a(ist3)=1.0/sqrt(a(ist3))
83:     t1=3
84: cdir$ ivdep
85:     do 30 i=i3,i3+collth(k+2)-2
86:         a(i)=a(i)*a(ist3)
87:         m3(t1)=a(i)
88:         t1=t1+1
89: 30     continue
90:
91: c *** Update, then finish column k+3.
92:     ist4=colptr(k+3)
93:     i4=ist4+1
94:     t1=3
95:     ict=0

```

Figure A4. Continued.

```

96: cdir$ ivdep
97:     do 31 i=ist4,ist4+collth(k+3)-1
98:         a(i)=a(i)-m1(t1)*m1(t1+ict)-m2(t1)*m2(t1+ict)-m3(t1)*m3(t1+ict)
99:         ict=ict+1
100: 31     continue
101:
102: c *** Finish the (k+3)rd multiplier.
103:     a(ist4)=1.0/sqrt(a(ist4))
104:     t1=4
105: cdir\$ ivdep
106:     do 40 i=i4,i4+collth(k+3)-2
107:         a(i)=a(i)*a(ist4)
108:         m4(t1)=a(i)
109:         t1=t1+1
110: 40     continue
111:
112: c *** Update, then finish column k+4.
113:     ist5=colptr(k+4)
114:     i5=ist5+1
115:     t1=4
116:     ict=0
117: cdir\$ ivdep
118:     do 41 i=ist5,ist5+collth(k+4)-1
119:         a(i)=a(i)-m1(t1)*m1(t1+ict)-m2(t1)*m2(t1+ict)-m3(t1)*m3(t1+ict)
120:         +           -m4(t1)*m4(t1+ict)
121:         ict=ict+1
122: 41     continue
123:
124: c *** Finish the (k+4)th multiplier.
125:     a(ist5)=1.0/sqrt(a(ist5))
126:     t1=5
127: cdir\$ ivdep
128:     do 50 i=i5,i5+collth(k+4)-2
129:         a(i)=a(i)*a(ist5)
130:         m5(t1)=a(i)
131:         t1=t1+1
132: 50     continue
133:
134: c *** Update, then finish column k+5.
135:     ist6=colptr(k+5)
136:     i6=ist6
137:     t1=5
138:     ict=0
139: cdir\$ ivdep
140:     do 51 i=ist6,ist6+collth(k+5)-1
141:         a(i)=a(i)-m1(t1)*m1(t1+ict)-m2(t1)*m2(t1+ict)-m3(t1)*m3(t1+ict)

```

Figure A4. Continued.

```

142:      +          -m4(t1)*m4(t1+ict)-m5(t1)*m5(t1+ict)
143:      ict=ict+1
144: 51      continue
145:
146: c *** Finish the (k+5)th multiplier.
147:      a(ist6)=1.0/sqrt(a(ist6))
148:      t1=6
149: cdir\$ ivdep
150:      do 60 i=ist6+1,ist6+collth(k+5)-1
151:          a(i)=a(i)*a(ist6)
152:          m6(t1)=a(i)
153:          t1=t1+1
154: 60      continue
155:      idone=k+5
156:
157: c *** Count all floating point vector operations for finishing
158: c *** this group of 6 multiplier columns.
159:      nopsf=nopsf+collth(k+5)*36 + 55
160:      End barrier
161:
162: c *** Everyone else copy columns k thru k+5 into local memory.
163:      if (idone.lt.k+5) then
164:          i1=colptr(k)+5
165:          i2=colptr(k+1)+4
166:          i3=colptr(k+2)+3
167:          i4=colptr(k+3)+2
168:          i5=colptr(k+4)+1
169:          i6=colptr(k+5)
170:          do 6111 ict=1,collth(k+5)-1
171:              m1(5+ict)=a(i1+ict)
172:              m2(5+ict)=a(i2+ict)
173:              m3(5+ict)=a(i3+ict)
174:              m4(5+ict)=a(i4+ict)
175:              m5(5+ict)=a(i5+ict)
176:              m6(5+ict)=a(i6+ict)
177: 6111      continue
178:          idone=k+5
179:      endif
180:
181: c *** End jki portion: 6 columns of L completed.
182:
183: c *** Begin kji portion: update columns k+6 thru lastr
184: c *** of matrix using the 6 completed multiplier columns.
185:
186: c *** Variable lastr corresponds to the last row in column k
187: c *** that is stored in the variable-band format.
188:      lastr=k+collth(k)-1

```

Figure A4. Continued.

```

189:
190:     Presched do 61 j=k+6,lastr
191:         t1=j-k
192:         itst=t1
193: c ***     Begin full zero checking
194:         if (m1(itst).eq.0.0) then
195:             if (m2(itst).eq.0.0) then
196:                 if (m3(itst).eq.0.0) then
197:                     if (m4(itst).eq.0.0) then
198:                         if (m5(itst).eq.0.0) then
199:                             if (m6(itst).eq.0.0) then
200:                                 goto 611
201:                             endif
202:                         endif
203:                     endif
204:                 endif
205:             endif
206:         endif
207: c ***     End zero checking.
208:         lth=lastr-j
209: cdir\$ ivdep
210:         do 610 i=colptr(j),colptr(j)+lth
211:             a(i)=a(i)-m1(itst)*m1(t1)-m2(itst)*m2(t1)
212:             +           -m3(itst)*m3(t1)-m4(itst)*m4(t1)
213:             +           -m5(itst)*m5(t1)-m6(itst)*m6(t1)
214:             t1=t1+1
215:         610     continue
216: c ***     Count floating point vector operations in kji update loop
217:         nopsf=nopsf+12*(lth+1).
218:
219:         611     continue
220:         61     End presched do
221:
222: c ***     End kji portion: increment k and begin next jki portion
223: c ***     if k is less than n-6.
224:
225:         k=k+6
226:         if (k.lt.n-6) goto 999
227:
228: c ***     Finish any remaining columns of L. There are up to 7
229: c ***     columns remaining. The last 7 columns are explicitly
230: c ***     computed without loop overhead, etc.
231:
232:         Barrier
233:         goto (100,110,120,130,140,150,160),(n-k+1)
234:

```

Figure A4. Continued.

```

235: 160  continue
236: c *** 6 columns left to update
237:     ist1=colptr(k)
238:     a(ist1)=1.0/sqrt(a(ist1))
239:     a(ist1+1)=a(ist1+1)*a(ist1)
240:     a(ist1+2)=a(ist1+2)*a(ist1)
241:     a(ist1+3)=a(ist1+3)*a(ist1)
242:     a(ist1+4)=a(ist1+4)*a(ist1)
243:     a(ist1+5)=a(ist1+5)*a(ist1)
244:     a(ist1+6)=a(ist1+6)*a(ist1)
245:     ist2=colptr(k+1)
246:     a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
247:     a(ist2+1)=a(ist2+1)-a(ist1+1)*a(ist1+2)
248:     a(ist2+2)=a(ist2+2)-a(ist1+1)*a(ist1+3)
249:     a(ist2+3)=a(ist2+3)-a(ist1+1)*a(ist1+4)
250:     a(ist2+4)=a(ist2+4)-a(ist1+1)*a(ist1+5)
251:     a(ist2+5)=a(ist2+5)-a(ist1+1)*a(ist1+6)
252:     ist3=colptr(k+2)
253:     a(ist3)=a(ist3)-a(ist1+2)*a(ist1+2)
254:     a(ist3+1)=a(ist3+1)-a(ist1+2)*a(ist1+3)
255:     a(ist3+2)=a(ist3+2)-a(ist1+2)*a(ist1+4)
256:     a(ist3+3)=a(ist3+3)-a(ist1+2)*a(ist1+5)
257:     a(ist3+4)=a(ist3+4)-a(ist1+2)*a(ist1+6)
258:     ist4=colptr(k+3)
259:     a(ist4)=a(ist4)-a(ist1+3)*a(ist1+3)
260:     a(ist4+1)=a(ist4+1)-a(ist1+3)*a(ist1+4)
261:     a(ist4+2)=a(ist4+2)-a(ist1+3)*a(ist1+5)
262:     a(ist4+3)=a(ist4+3)-a(ist1+3)*a(ist1+6)
263:     ist5=colptr(k+4)
264:     a(ist5)=a(ist5)-a(ist1+4)*a(ist1+4)
265:     a(ist5+1)=a(ist5+1)-a(ist1+4)*a(ist1+5)
266:     a(ist5+2)=a(ist5+2)-a(ist1+4)*a(ist1+6)
267:     ist6=colptr(k+5)
268:     a(ist6)=a(ist6)-a(ist1+5)*a(ist1+5)
269:     a(ist6+1)=a(ist6+1)-a(ist1+5)*a(ist1+6)
270:     a(colptr(k+6))=a(colptr(k+6))-a(ist1+6)*a(ist1+6)
271:     k=k+1
272:     nopsf=nopsf+49
273:
274: 150  continue
275: c *** 5 columns left to update
276:     ist1=colptr(k)
277:     a(ist1)=1.0/sqrt(a(ist1))
278:     a(ist1+1)=a(ist1+1)*a(ist1)
279:     a(ist1+2)=a(ist1+2)*a(ist1)

```

Figure A4. Continued.


```

280:      a(ist1+3)=a(ist1+3)*a(ist1)
281:      a(ist1+4)=a(ist1+4)*a(ist1)
282:      a(ist1+5)=a(ist1+5)*a(ist1)
283:      ist2=colptr(k+1)
284:      a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
285:      a(ist2+1)=a(ist2+1)-a(ist1+1)*a(ist1+2)
286:      a(ist2+2)=a(ist2+2)-a(ist1+1)*a(ist1+3)
287:      a(ist2+3)=a(ist2+3)-a(ist1+1)*a(ist1+4)
288:      a(ist2+4)=a(ist2+4)-a(ist1+1)*a(ist1+5)
289:      ist3=colptr(k+2)
290:      a(ist3)=a(ist3)-a(ist1+2)*a(ist1+2)
291:      a(ist3+1)=a(ist3+1)-a(ist1+2)*a(ist1+3)
292:      a(ist3+2)=a(ist3+2)-a(ist1+2)*a(ist1+4)
293:      a(ist3+3)=a(ist3+3)-a(ist1+2)*a(ist1+5)
294:      ist4=colptr(k+3)
295:      a(ist4)=a(ist4)-a(ist1+3)*a(ist1+3)
296:      a(ist4+1)=a(ist4+1)-a(ist1+3)*a(ist1+4)
297:      a(ist4+2)=a(ist4+2)-a(ist1+3)*a(ist1+5)
298:      ist5=colptr(k+4)
299:      a(ist5)=a(ist5)-a(ist1+4)*a(ist1+4)
300:      a(ist5+1)=a(ist5+1)-a(ist1+4)*a(ist1+5)
301:      a(colptr(k+5))=a(colptr(k+5))-a(ist1+5)*a(ist1+5)
302:      k=k+1
303:      nopsf=nopsf+36
304:
305: 140  continue
306: c *** 4 columns left to update
307:      ist1=colptr(k)
308:      a(ist1)=1.0/sqrt(a(ist1))
309:      a(ist1+1)=a(ist1+1)*a(ist1)
310:      a(ist1+2)=a(ist1+2)*a(ist1)
311:      a(ist1+3)=a(ist1+3)*a(ist1)
312:      a(ist1+4)=a(ist1+4)*a(ist1)
313:      ist2=colptr(k+1)
314:      a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
315:      a(ist2+1)=a(ist2+1)-a(ist1+1)*a(ist1+2)
316:      a(ist2+2)=a(ist2+2)-a(ist1+1)*a(ist1+3)
317:      a(ist2+3)=a(ist2+3)-a(ist1+1)*a(ist1+4)
318:      ist3=colptr(k+2)
319:      a(ist3)=a(ist3)-a(ist1+2)*a(ist1+2)
320:      a(ist3+1)=a(ist3+1)-a(ist1+2)*a(ist1+3)
321:      a(ist3+2)=a(ist3+2)-a(ist1+2)*a(ist1+4)
322:      ist4=colptr(k+3)
323:      a(ist4)=a(ist4)-a(ist1+3)*a(ist1+3)
324:      a(ist4+1)=a(ist4+1)-a(ist1+3)*a(ist1+4)
325:      a(colptr(k+4))=a(colptr(k+4))-a(ist1+4)*a(ist1+4)
326:      k=k+1
327:      nopsf=nopsf+25
328:

```

Figure A4. Continued.

```

329: 130  continue
330: c *** 3 columns left to update
331:     ist1=colptr(k)
332:     a(ist1)=1.0/sqrt(a(ist1))
333:     a(ist1+1)=a(ist1+1)*a(ist1)
334:     a(ist1+2)=a(ist1+2)*a(ist1)
335:     a(ist1+3)=a(ist1+3)*a(ist1)
336:     ist2=colptr(k+1)
337:     a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
338:     a(ist2+1)=a(ist2+1)-a(ist1+1)*a(ist1+2)
339:     a(ist2+2)=a(ist2+2)-a(ist1+1)*a(ist1+3)
340:     ist3=colptr(k+2)
341:     a(ist3)=a(ist3)-a(ist1+2)*a(ist1+2)
342:     a(ist3+1)=a(ist3+1)-a(ist1+2)*a(ist1+3)
343:     a(colptr(k+3))=a(colptr(k+3))-a(ist1+3)*a(ist1+3)
344:     k=k+1
345:     nopsf=nopsf+16
346:
347: 120  continue
348: c *** 2 columns left to update
349:     ist1=colptr(k)
350:     a(ist1)=1.0/sqrt(a(ist1))
351:     a(ist1+1)=a(ist1+1)*a(ist1)
352:     a(ist1+2)=a(ist1+2)*a(ist1)
353:     ist2=colptr(k+1)
354:     a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
355:     a(ist2+1)=a(ist2+1)-a(ist1+1)*a(ist1+2)
356:     a(colptr(k+2))=a(colptr(k+2))-a(ist1+2)*a(ist1+2)
357:     k=k+1
358:     nopsf=nopsf+9
359:
360: 110  continue
361: c *** 1 column left to update
362:     ist1=colptr(k)
363:     a(ist1)=1.0/sqrt(a(ist1))
364:     a(ist1+1)=a(ist1+1)*a(ist1)
365:     ist2=colptr(k+1)
366:     a(ist2)=a(ist2)-a(ist1+1)*a(ist1+1)
367:     nopsf=nopsf+4
368:
369: 100  continue
370: c *** 0 columns left to update
371:     a(colptr(n))=1.0/sqrt(a(colptr(n)))
372:     nopsf=nopsf+1
373:     End barrier
374:
375:     return
376:     end

```

Figure A4. Concluded.

```

1:      subroutine vbsolve(a,alth,colptr,collth,n,b,nopss)
2:      integer alth,n,nopss
3:      integer i,j,ict,lastr,left
4:      integer t1,t2,t3,t4,t5,t6,t7
5:      integer ist1,ist2,ist3,ist4,ist5,ist6
6:      integer colptr(n),collth(n)
7:      real a(alth),b(n),t
8:
9: c *** This routine performs forward-and-backward substitution
10: c *** of triangular factored matrices. It assumes a variable-band
11: c *** data structure with adjustments for loop unrolling to level 6.
12: c *** NOTE: This solver assumes that the diagonal elements of
13: c *** the triangular systems are the reciprocals of the actual
14: c *** diagonal elements
15:      nopss=0
16:
17: c *** Solve Ly=b (column sweep)
18:
19:      left=mod(n,6)
20:      last=n-left
21:      do 1 i=1,last,6
22:
23:          ist1=colptr(i)
24:          ist2=colptr(i+1)
25:          ist3=colptr(i+2)
26:          ist4=colptr(i+3)
27:          ist5=colptr(i+4)
28:          ist6=colptr(i+5)
29: c *** Begin zero-checking option.
30:          if (b(i).eq.0.0) then
31:              if (b(i+1).eq.0.0) then
32:                  if (b(i+2).eq.0.0) then
33:                      if (b(i+3).eq.0.0) then
34:                          if (b(i+4).eq.0.0) then
35:                              if (b(i+5).eq.0.0) then
36:                                  goto 11
37:                              endif
38:                          endif
39:                      endif
40:                  endif
41:              endif
42:          endif
43: c *** End zero-checking option.
44:          b(i)=b(i)*a(ist1)
45:          b(i+1)=(b(i+1)-a(ist1+1)*b(i)
46:      +          )*a(ist2)
47:          b(i+2)=(b(i+2)-a(ist1+2)*b(i)

```

Figure A5. FORTRAN listing for subroutine *vbsolve*.

```

48:      +          -a(ist2+1)*b(i+1)
49:      +          )*a(ist3)
50:      b(i+3)=(b(i+3)-a(ist1+3)*b(i)
51:      +          -a(ist2+2)*b(i+1)
52:      +          -a(ist3+1)*b(i+2)
53:      +          )*a(ist4)
54:      b(i+4)=(b(i+4)-a(ist1+4)*b(i)
55:      +          -a(ist2+3)*b(i+1)
56:      +          -a(ist3+2)*b(i+2)
57:      +          -a(ist4+1)*b(i+3)
58:      +          )*a(ist5)
59:      b(i+5)=(b(i+5)-a(ist1+5)*b(i)
60:      +          -a(ist2+4)*b(i+1)
61:      +          -a(ist3+3)*b(i+2)
62:      +          -a(ist4+2)*b(i+3)
63:      +          -a(ist5+1)*b(i+4)
64:      +          )*a(ist6)
65:
66:      nopss=nopss+36
67:      ist1=ist1+6
68:      ist2=ist2+5
69:      ist3=ist3+4
70:      ist4=ist4+3
71:      ist5=ist5+2
72:      ist6=ist6+1
73:      lth=collth(i)-6
74:      jrow=i+6
75: cdir\$ ivdep
76:      do 10 ict=0,lth-1
77:      b(jrow)=b(jrow)-a(ist1+ict)*b(i)
78:      +          -a(ist2+ict)*b(i+1)
79:      +          -a(ist3+ict)*b(i+2)
80:      +          -a(ist4+ict)*b(i+3)
81:      +          -a(ist5+ict)*b(i+4)
82:      +          -a(ist6+ict)*b(i+5)
83:      jrow=jrow+1
84: 10      continue
85:      nopss=nopss+12*lth
86: 11      continue
87: 1      continue
88:
89:      goto (100,101,102,103,104,105),(left+1)
90:
91: 105     continue
92:      i=n-4
93:      ist1=colptr(i)

```

Figure A5. Continued.

```

94:      b(i)=b(i)*a(ist1)
95:      b(i+1)=b(i+1)-a(ist1+1)*b(i)
96:      b(i+2)=b(i+2)-a(ist1+2)*b(i)
97:      b(i+3)=b(i+3)-a(ist1+3)*b(i)
98:      b(i+4)=b(i+4)-a(ist1+4)*b(i)
99:      nopss=nopss+9
100:
101: 104  continue
102:      i=n-3
103:      ist1=colptr(i)
104:      b(i)=b(i)*a(ist1)
105:      b(i+1)=b(i+1)-a(ist1+1)*b(i)
106:      b(i+2)=b(i+2)-a(ist1+2)*b(i)
107:      b(i+3)=b(i+3)-a(ist1+3)*b(i)
108:      nopss=nopss+7
109:
110: 103  continue
111:      i=n-2
112:      ist1=colptr(i)
113:      b(i)=b(i)*a(ist1)
114:      b(i+1)=b(i+1)-a(ist1+1)*b(i)
115:      b(i+2)=b(i+2)-a(ist1+2)*b(i)
116:      nopss=nopss+5
117:
118: 102  continue
119:      i=n-1
120:      ist1=colptr(i)
121:      b(i)=b(i)*a(ist1)
122:      b(i+1)=b(i+1)-a(ist1+1)*b(i)
123:      nopss=nopss+3
124:
125: 101  continue
126:      ist1=colptr(n)
127:      b(n)=b(n)*a(ist1)
128:      nopss=nopss+1
129:
130: 100  continue
131:
132: c *** Solve L(t)x=y
133:
134:      goto (200,201,202,203,204,205) left+1
135:
136: 205  continue
137:      ist1=colptr(n)
138:      ist2=colptr(n-1)
139:      ist3=colptr(n-2)

```

Figure A5. Continued.

```

140:     ist4=colptr(n-3)
141:     ist5=colptr(n-4)
142:     b(n)=b(n)*a(ist1)
143:     b(n-1)=(b(n-1)-a(ist2+1)*b(n)
144: +         )*a(ist2)
145:     b(n-2)=(b(n-2)-a(ist3+2)*b(n)
146: +         -a(ist3+1)*b(n-1)
147: +         )*a(ist3)
148:     b(n-3)=(b(n-3)-a(ist4+3)*b(n)
149: +         -a(ist4+2)*b(n-1)
150: +         -a(ist4+1)*b(n-2)
151: +         )*a(ist4)
152:     b(n-4)=(b(n-4)-a(ist5+4)*b(n)
153: +         -a(ist5+3)*b(n-1)
154: +         -a(ist5+2)*b(n-2)
155: +         -a(ist5+1)*b(n-3)
156: +         )*a(ist5)
157:     nopss=nopss+25
158:     goto 200
159:
160: 204  continue
161:     ist1=colptr(n)
162:     ist2=colptr(n-1)
163:     ist3=colptr(n-2)
164:     ist4=colptr(n-3)
165:     b(n)=b(n)*a(ist1)
166:     b(n-1)=(b(n-1)-a(ist2+1)*b(n)
167: +         )*a(ist2)
168:     b(n-2)=(b(n-2)-a(ist3+2)*b(n)
169: +         -a(ist3+1)*b(n-1)
170: +         )*a(ist3)
171:     b(n-3)=(b(n-3)-a(ist4+3)*b(n)
172: +         -a(ist4+2)*b(n-1)
173: +         -a(ist4+1)*b(n-2)
174: +         )*a(ist4)
175:     nopss=nopss+16
176:     goto 200
177:
178: 203  continue
179:     ist1=colptr(n)
180:     ist2=colptr(n-1)
181:     ist3=colptr(n-2)
182:     b(n)=b(n)*a(ist1)
183:     b(n-1)=(b(n-1)-a(ist2+1)*b(n)
184: +         )*a(ist2)
185:     b(n-2)=(b(n-2)-a(ist3+2)*b(n)
186: +         -a(ist3+1)*b(n-1)
187: +         )*a(ist3)

```

Figure A5. Continued.

```

188:      nopss=nopss+9
189:      goto 200
190:
191: 202  continue
192:      ist1=colptr(n)
193:      ist2=colptr(n-1)
194:      b(n)=b(n)*a(ist1)
195:      b(n-1)=(b(n-1)-a(ist2+1)*b(n)
196:      +      )*a(ist2)
197:      nopss=nopss+4
198:      goto 200
199:
200: 201  continue
201:      ist1=colptr(n)
202:      b(n)=b(n)*a(ist1)
203:      nopss=nopss+1
204:
205: 200  continue
206:
207:      do 2 i=last,1,-6
208:          lth=collth(i)-1
209:          ist1=colptr(i-5)+5
210:          ist2=colptr(i-4)+4
211:          ist3=colptr(i-3)+3
212:          ist4=colptr(i-2)+2
213:          ist5=colptr(i-1)+1
214:          ist6=colptr(i)
215:          if (lth.eq.0) goto 21
216: cdir\$ ivdep
217:          do 20 ict=1,lth
218:              b(i)=b(i)-a(ist6+ict)*b(i+ict)
219:              b(i-1)=b(i-1)-a(ist5+ict)*b(i+ict)
220:              b(i-2)=b(i-2)-a(ist4+ict)*b(i+ict)
221:              b(i-3)=b(i-3)-a(ist3+ict)*b(i+ict)
222:              b(i-4)=b(i-4)-a(ist2+ict)*b(i+ict)
223:              b(i-5)=b(i-5)-a(ist1+ict)*b(i+ict)
224: 20      continue
225:          nopss=nopss+12*lth
226: 21      continue
227:          b(i)=b(i)*a(ist6)
228:          b(i-1)=(b(i-1)-a(ist5)*b(i)
229:          +      )*a(ist5-1)
230:          b(i-2)=(b(i-2)-a(ist4)*b(i)
231:          +      -a(ist4-1)*b(i-1)
232:          +      )*a(ist4-2)
233:          b(i-3)=(b(i-3)-a(ist3)*b(i)

```

Figure A5. Continued.

```

234:      +          -a(ist3-1)*b(i-1)
235:      +          -a(ist3-2)*b(i-2)
236:      +          )*a(ist3-3)
237:      b(i-4)=(b(i-4)-a(ist2)*b(i
238:      +          -a(ist2-1)*b(i-1)
239:      +          -a(ist2-2)*b(i-2)
240:      +          -a(ist2-3)*b(i-3)
241:      +          )*a(ist2-4)
242:      b(i-5)=(b(i-5)-a(ist1)*b(i
243:      +          -a(ist1-1)*b(i-1)
244:      +          -a(ist1-2)*b(i-2)
245:      +          -a(ist1-3)*b(i-3)
246:      +          -a(ist1-4)*b(i-4)
247:      +          )*a(ist1-5)
248:      nopss=nopss+36
249:  2    continue
250:
251:      return
252:      end

```

Figure A5. Concluded.

References

1. Ortega, James M.: *Matrix Theory—A Second Course*. Plenum Press, c.1987.
2. Everstine, Gordon C.: *The BANDIT Computer Program for the Reduction of Matrix Bandwidth for NASTRAN*. Rep. 3827, Naval Ship Research & Development Center, Mar. 1972.
3. Cray Research, Inc.: *Cray Multitasking Programmer's Reference Manual*. Central Scientific Computing Complex Document CR-18a, SN-2026C, c.1989.
4. Cray Research, Inc.: *Cray Y-MP, Cray X-MP EA and Cray X-MP Multitasking Programmer's Manual*. Central Scientific Computing Complex Document CR-33, SR-0222 F-01, c.1989.
5. Jordan, Harry F.: Programming Language Concepts for Multiprocessors. *Parallel Comput.*, vol. 8, no. 1-3, Oct. 1988, pp. 31-40.
6. Jordan, Harry F.; Benten, Muhammad S.; Arenstorff, Norbert S.; and Ramanan, Aruna V.: *Force User's Manual—A Portable, Parallel FORTRAN*. NASA CR-4265, 1990.
7. Cray Research, Inc.: *Cray UNICOS Autotasking User Guide*. Central Scientific Computing Complex Document CR-17, SN-2088 CFT77 3.0, c.1989.
8. Poole, Eugene L.; and Overman, Andrea L.: *The Solution of Linear Systems of Equations With a Structural Analysis Code on the NAS CRAY-2*. NASA CR-4159, 1988.
9. Agarwal, Tarun K.; Storaasli, Olaf O.; and Nguyen, Duc T.: A Parallel-Vector Algorithm for Rapid Structural Analysis on High-Performance Computers. *A Collection of Technical Papers, Part 2—AIAA/ASME/ASCE/AHS/ASC 31st Structures, Structural Dynamics and Materials Conference*, Apr. 1990, pp. 662-672. (Available as AIAA-90-1149-CP.)
10. Ortega, J. M.: The ijk Forms of Factorization Methods I. Vector Computers. *Parallel Comput.*, vol. 7, no. 2, June 1988, pp. 135-147.
11. Arenstorff, Norbert S.; and Jordan, Harry F.: Comparing Barrier Algorithms. *Parallel Comput.*, vol. 12, no. 1, Oct. 1989, pp. 157-170.
12. Jones, Mark T.; and Patrick, Merrell L.: *The Use of Lanczo's Method To Solve the Large Generalized Symmetric Eigenvalue Problem in Parallel*. NASA CR-182072, ICASE Rep. No. 90-48, 1990.
13. Ortega, James M.: *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum Press, c.1988.
14. Knight, N. F., Jr.; Gillian, R. E.; McCleary, S. L.; Lotts, C. G.; Poole, E. L.; Overman, A. L.; and Macy, S. C.: CSM Testbed Development and Large-Scale Structural Applications. *Science and Engineering on Cray Supercomputers - Proceedings of the Fourth International Symposium*, Cray Research, Inc., 1988, pp. 359-387. (Available as NASA TM-4072, 1989.)
15. Knight, Norman F., Jr.; McCleary, Susan L.; Macy, Steven C.; and Aminpour, Mohammad A.: *Large-Scale Structural Analysis: The Structural Analyst, the CSM Testbed, and the NAS System*. NASA TM-100643, 1989.
16. Robins, A. Warner; Dollyhigh, Samuel M.; Beissner, Fred L., Jr.; Geiselhart, Karl; Martin, Glenn L.; Shields, E. W.; Swanson, E. E.; Coen, Peter G.; and Morris, Shelby J., Jr.: *Concept Development of a Mach 3.0 High-Speed Civil Transport*. NASA TM-4058, 1988.
17. Kao, Pi-Jen; Wrenn, Gregory A.; and Giles, Gary L.: Comparison of Equivalent Plate and Finite Element Analysis of a Realistic Aircraft Structural Configuration. AIAA-90-3293, Sept. 1990.
18. Knight, Norman F., Jr.; Gillian, Ronnie E.; and Nemeth, Michael P.: *Preliminary 2-D Shell Analysis of the Space Shuttle Solid Rocket Boosters*. NASA TM-100515, 1988.
19. *PATRAN Plus User Manual—Volume II*, Release 2.4. Publ. No. 2191024, PDA Engineering, c.1990.

