3/31/91 Open O

ornl

ORNL/TM-11960

OAK RIDGE
NATIONAL
LABORATORY

MARTIN MARIETTA

# Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers

Esmond G. Ng
Barry W. Peyton

MANAGED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

ORNL/TM--11960

DE92 004167

Engineering Physics and Mathematics Division

Mathematical Sciences Section

# BLOCK SPARSE CHOLESKY ALGORITHMS ON ADVANCED UNIPROCESSOR COMPUTERS

Esmond G. Ng
Barry W. Peyton

Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

DATE PUBLISHED - DECEMBER 1991

Research was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

## Contents

# BLOCK SPARSE CHOLESKY ALGORITHMS ON ADVANCED UNIPROCESSOR COMPUTERS

Esmond G. Ng
Barry W. Peyton

**Abstract**

As with many other linear algebra algorithms, devising a portable implementation of sparse Cholesky factorization that performs well on the broad range of computer architectures currently available is a formidable challenge. Even after limiting our attention to machines with only one processor, as we have done in this report, there are still several interesting issues to consider. For dense matrices, it is well known that block factorization algorithms are the best means of achieving this goal. We take this approach for sparse factorization as well.

This paper has two primary goals. First, we examine two sparse Cholesky factorization algorithms, the multifrontal method and a blocked left-looking sparse Cholesky method, in a systematic and consistent fashion, both to illustrate the strengths of the blocking techniques in general and to obtain a fair evaluation of the two approaches. Second, we assess the impact of various implementation techniques on time and storage efficiency, paying particularly close attention to the work-storage requirement of the two methods and their variants.

## 1. Introduction

Many scientific and engineering applications require the solution of large sparse symmetric positive definite systems of linear equations. *Direct* methods use Cholesky factorization followed by forward and backward triangular solutions to solve such systems. For any $n \times n$ symmetric positive definite matrix $\boldsymbol{A}$, its Cholesky factor $\boldsymbol{L}$ is the lower triangular matrix with positive diagonal such that $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$. When $\boldsymbol{A}$ is sparse, it will generally suffer some fill during the computation of $\boldsymbol{L}$; that is, some of the zero elements in $\boldsymbol{A}$ will become nonzero elements in $\boldsymbol{L}$. In order to reduce time and storage requirements, only the nonzero positions of $\boldsymbol{L}$ are stored and operated on during *sparse* Cholesky factorization. Techniques for accomplishing this task and for reducing fill have been studied extensively (see [12,19] for details). In this paper we restrict our attention to the numerical factorization phase. We assume that the preprocessing steps, such as reordering to reduce fill and symbolic factorization to set up the compact data structure for $\boldsymbol{L}$, have been performed. Details on the preprocessing can be found in [12,19].

As with many other linear algebra algorithms, devising a portable implementation of sparse Cholesky factorization that performs well on the broad range of computer architectures currently available is a formidable challenge. Even after limiting our attention to machines with only one processor, as we have done herein, there are still several interesting issues to consider. In this paper we will investigate sparse Cholesky algorithms designed to run efficiently on vector supercomputers (e.g., the Cray Y-MP) and on powerful scientific workstations (e.g., the IBM RS/6000, the DEC 5000, and the Stardent P3000). To achieve high performance on such machines, the algorithms must be able to exploit vector processors and/or pipelined functional units. Moreover, with the dramatic increases in processor speed during the past few years, rapid memory access has become a very important factor in determining performance levels on several of these machines. To be efficient, algorithms must reuse data in fast memory (e.g., cache) as much as possible. Consequently, a highly localized and regular memory-access pattern is ideal for many of today's fastest machines.

It is well known that block factorization algorithms are the best means of achieving this goal. Perhaps the best-known example of a software package based on this approach is LAPACK, a software package for performing dense linear algebra computations on advanced computer architectures including shared-memory multiprocessor systems [2]. Each block algorithm in LAPACK is built around some computationally intensive variant of a matrix-matrix (BLAS3) or matrix-vector (BLAS2) multiplication kernel subroutine, which can be optimized for each computing platform on which the package is run.

The sparse block Cholesky algorithms discussed in this paper take essentially the same approach; we do not, however, include multiprocessors nor do we tune the kernels for efficiency on specific machines. We investigate two algorithms:

1. The multifrontal method [15,24], which is based on the right-looking formulation of the Cholesky factorization algorithm.

2. A left-looking block algorithm that has, until recently (this report and [29]),

received little attention in the literature.

Both methods will use the same kernel subroutines to do all the numerical work required during the factorization. The differences are limited to such issues as:

- indirect addressing and other integer operations related to the *structural* aspects of sparse factorization,
- the ability to reuse data in cache,
- the amount of data movement,
- the memory-access pattern, and
- the work-storage requirement.

In general, variations in the efficiency of the block algorithms and their variants are not very large. However, our tests indicate significant differences in the amount of work storage and expensive data movement required.

This paper has two primary goals. First, we will look at the two block Cholesky factorization algorithms in a systematic and consistent fashion, both to illustrate the strengths of the blocking techniques in sparse matrix computations in general and to obtain a fair evaluation of the two basic approaches. Second, we will assess the value of various implementation techniques on time and storage efficiency, paying particularly close attention to the work-storage requirement of the two methods and their variants.

Rothberg and Gupta [29] have studied these algorithms independently. They consider the caching issue in more detail and implement a more complicated and effective loop-unrolling scheme than we do. However, they do not compare the work-storage requirements of the various algorithms as we do. We have introduced enhancements to the multifrontal algorithm that greatly reduce the amount of stack storage and data movement overhead required by that algorithm. Also, we consider the performance of these algorithms on a vector supercomputer and a high-performance workstation with vector hardware.

The paper is organized as follows. Section 2 contains notation and other background material needed to present the algorithms, including a discussion of previous work on block sparse Cholesky algorithms. Section 3 describes the left-looking block Cholesky algorithm and some of its key features. Presented in Section 4 are implementation details and enhancements for both the left-looking block algorithm and the multifrontal algorithm. Section 5 contains the results of our performance tests on several of the machines mentioned earlier in this section. Finally, concluding remarks and speculations on future work appear in Section 6.

## 2. Background material

### 2.1. Column-based Cholesky factorization methods

Cholesky factorization of a symmetric positive definite matrix $\boldsymbol{A}$ can be described as a triple nested loop around the single statement

$$a_{ij} = a_{ij} - (a_{ik}a_{kj})/a_{kk}.$$

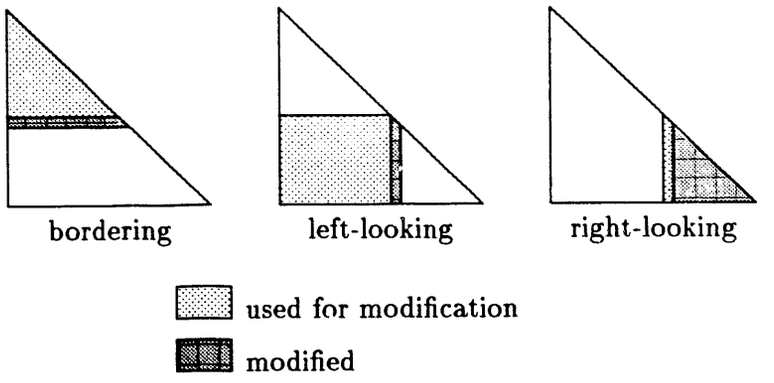


Figure 1: Three forms of Cholesky factorization.

By varying the order in which the loop indices $i$, $j$, and $k$ are nested, we obtain three different formulations of Cholesky factorization, each with a different memory access pattern.

1. *Bordering Cholesky*: Taking $i$ in the outer loop, successive rows of $\boldsymbol{L}$ are computed one by one, with the inner loops solving a triangular system for each new row in terms of the previously computed rows (see Figure 1).

2. *Left-looking Cholesky*: Taking $j$ in the outer loop, successive columns of $\boldsymbol{L}$ are computed one by one, with the inner loops computing a matrix-vector product that gives the effect of previously computed columns on the column currently being computed.

3. *Right-looking Cholesky*: Taking $k$ in the outer loop, successive columns of $\boldsymbol{L}$ are computed one by one, with the inner loops applying the current column as a rank-1 update to the remaining partially-reduced submatrix.

The various versions of Cholesky factorization can be used to take better advantage of particular architectural features of a given machine (cache, virtual memory, vectorization, etc.) [11]. For more details concerning these three versions of Cholesky factorization, consult George and Liu [19, pp. 18–21].

The bordering method requires a row-oriented data structure for storing the nonzeros of $\boldsymbol{L}$. Liu [25] has devised a compact row-oriented data structure for this purpose, but currently the technique has not been successfully adapted to run efficiently on modern workstations and vector supercomputers. Consequently, our report will focus on block versions of the left-looking and right-looking algorithms (also known as column-Cholesky and submatrix-Cholesky, respectively). Both the left-looking and right-looking algorithms naturally require a column-oriented data structure, which is easy to construct [31]. Thus, we restrict our attention to column-oriented implementations of the left-looking and right-looking algorithms.

We need the following definitions to write down the algorithms. Let $\boldsymbol{M}$ be an $n$ by $n$ matrix and denote the $j$-th column of $\boldsymbol{M}$ by $\boldsymbol{M}_{*,j}$. The sparsity structure of column $j$ in the *lower triangular* part of $\boldsymbol{M}$ (excluding the diagonal entry) is denoted by $Struct(\boldsymbol{M}_{*,j})$. That is,

$$Struct(\boldsymbol{M}_{*,j}) := \{s > j : \boldsymbol{M}_{s,j} \neq 0\}.$$

Column-oriented Cholesky factorization algorithms can be expressed in terms of the following two subtasks:

1. $cmod(j,k)$ : modification of column $j$ by a multiple of column $k$, $k < j$,
2. $cdiv(j)$ : division of column $j$ by a scalar.

Of course, sparsity in columns $j$ and $k$ is exploited when $\boldsymbol{A}$ and $\boldsymbol{L}$ are sparse. Using these basic operations, Figures 2 and 3 give high-level descriptions of the basic left-looking and right-looking sparse Cholesky algorithms, respectively. (We will refer to these two algorithms as left-looking and right-looking `col-col`.)

---

**for** $j = 1$ **to** $n$ **do**
  **for** $k$ such that $\boldsymbol{L}_{j,k} \neq 0$ **do**
    $cmod(j,k)$
  $cdiv(j)$

Figure 2: Left-looking sparse Cholesky factorization algorithm (left-looking `col-col`).

---

**for** $k = 1$ **to** $n$ **do**
  $cdiv(k)$
  **for** $j$ such that $\boldsymbol{L}_{j,k} \neq 0$ **do**
    $cmod(j,k)$

Figure 3: Right-looking sparse Cholesky factorization algorithm (right-looking `col-col`).

---

Left-looking sparse Cholesky is the simpler of the two algorithms to implement, and it appears in several well known commercially available sparse matrix packages [8,16]. For implementation details, the reader should consult George and Liu [19]. Straightforward implementations of the right-looking approach are generally quite inefficient because matching the updating column $k$'s sparsity pattern with that of each column $j$ in the updated submatrix requires expensive searching through the row indices in $Struct(\boldsymbol{L}_{*,k})$ and $Struct(\boldsymbol{L}_{*,j})$, $j \in Struct(\boldsymbol{L}_{*,k})$. Consequently, we will not pursue

such an implementation in this paper. However, Rothberg and Gupta [29] have recently reported that a block version of this approach is reasonably competitive, because for practical problems the blocking greatly reduces the amount of index matching needed. Note also that a straightforward implementation of the right-looking approach forms the basis for a distributed-memory parallel factorization algorithm known as the fan-out method [6,18,33]. In this paper, we will study a left-looking block algorithm and also the *multifrontal* algorithm [15,24], which can be viewed as an efficient implementation of right-looking sparse Cholesky factorization as we shall see in Section 2.3.

### 2.2. Supernodes and elimination trees

Efficient implementations of both the multifrontal algorithm and left-looking block algorithms require that columns of the Cholesky factor $\boldsymbol{L}$ sharing the same sparsity structure be grouped together into so-called *supernodes*. More formally, the set of contiguous columns[1] $j$, $j+1$, ..., $j+t$ constitutes a supernode if $Struct(\boldsymbol{L}_{*,k}) = Struct(\boldsymbol{L}_{*,k+1}) \cup \{k+1\}$ for $j \leq k \leq j+t-1$. A set of supernodes for an example matrix is shown in Figure 4. Note that the columns of a supernode $\{j, j+1, \ldots, j+t\}$ have a dense diagonal block and have *identical* column structure below row $j+t$. Note also that columns in the same supernode can be treated as a unit for both computation and storage. (See, for example, [26] for further details.)

The multifrontal method makes explicit use of the *elimination tree* associated with $\boldsymbol{L}$. For each column $\boldsymbol{L}_{*,j}$ having off-diagonal nonzero elements, we define the *parent* of $j$ to be the row index of the first off-diagonal nonzero in that column. For example, the parent of node 9 is node 19 for the matrix in Figure 4. It is easy to see that the parents of the columns define a tree structure, which is called the elimination tree of $\boldsymbol{L}$. Associated with any supernode partition is a *supernodal elimination tree*, which is obtained from the elimination tree essentially by collapsing the nodes (columns) in each supernode into a single node (block column). This can be done because the nodes in each supernode form a chain in the elimination tree. Figure 5 displays the elimination tree for the matrix in Figure 4. The supernodal elimination tree for the partition in Figure 4 is also shown in Figure 5, superimposed on the underlying elimination tree.

### 2.3. Supernode-based Cholesky algorithms: previous work

Figures 2 and 3 contain high-level descriptions of sparse Cholesky algorithms whose innermost loop updates a single column $j$ with a multiple of a single column $k$. The next two subsections briefly describe two well known sparse-Cholesky algorithms that exploit the shared sparsity structure within supernodes to improve performance. The first is the left-looking `sup-col` algorithm, whose atomic operation is updating the target column $j$ with every column in a supernode (a BLAS2 operation). The other is the more widely known multifrontal method.

[1] It is convenient to denote a column $L_{*,j}$ belonging to a supernode by its column index $j$. It should be clear by context when $j$ is being used in this manner.

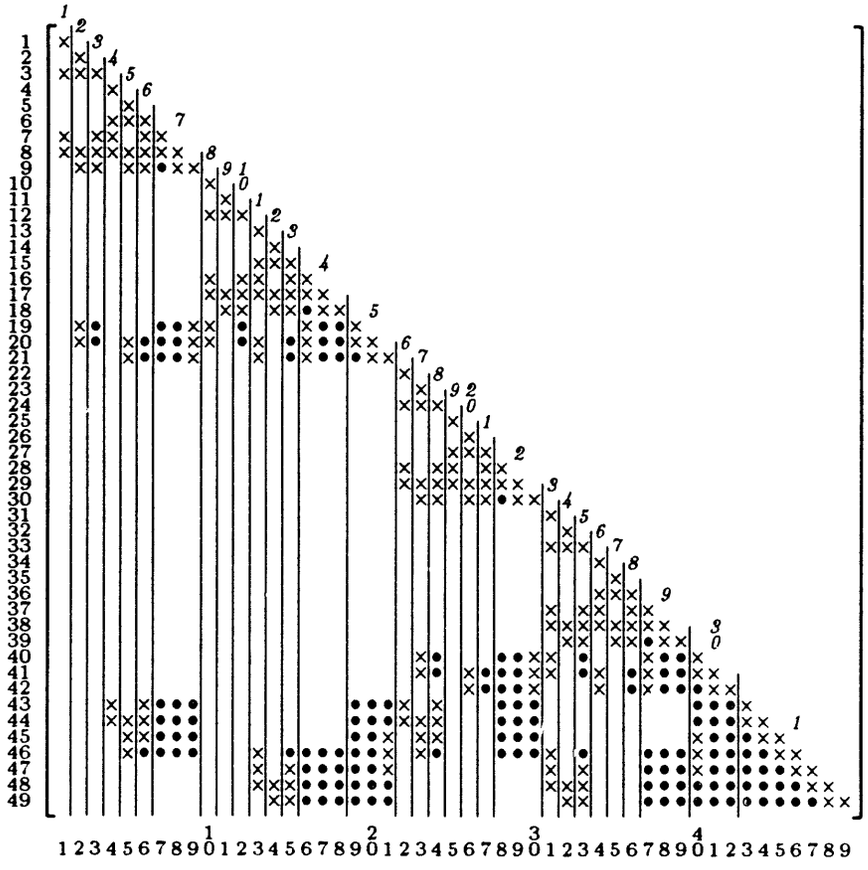

Figure 4: Supernodes for $7 \times 7$ nine-point grid problem ordered by nested dissection. ($\times$ and $\bullet$ refer to nonzeros in $\boldsymbol{A}$ and fill in $\boldsymbol{L}$, respectively. Numbers over diagonal entries label supernodes.)

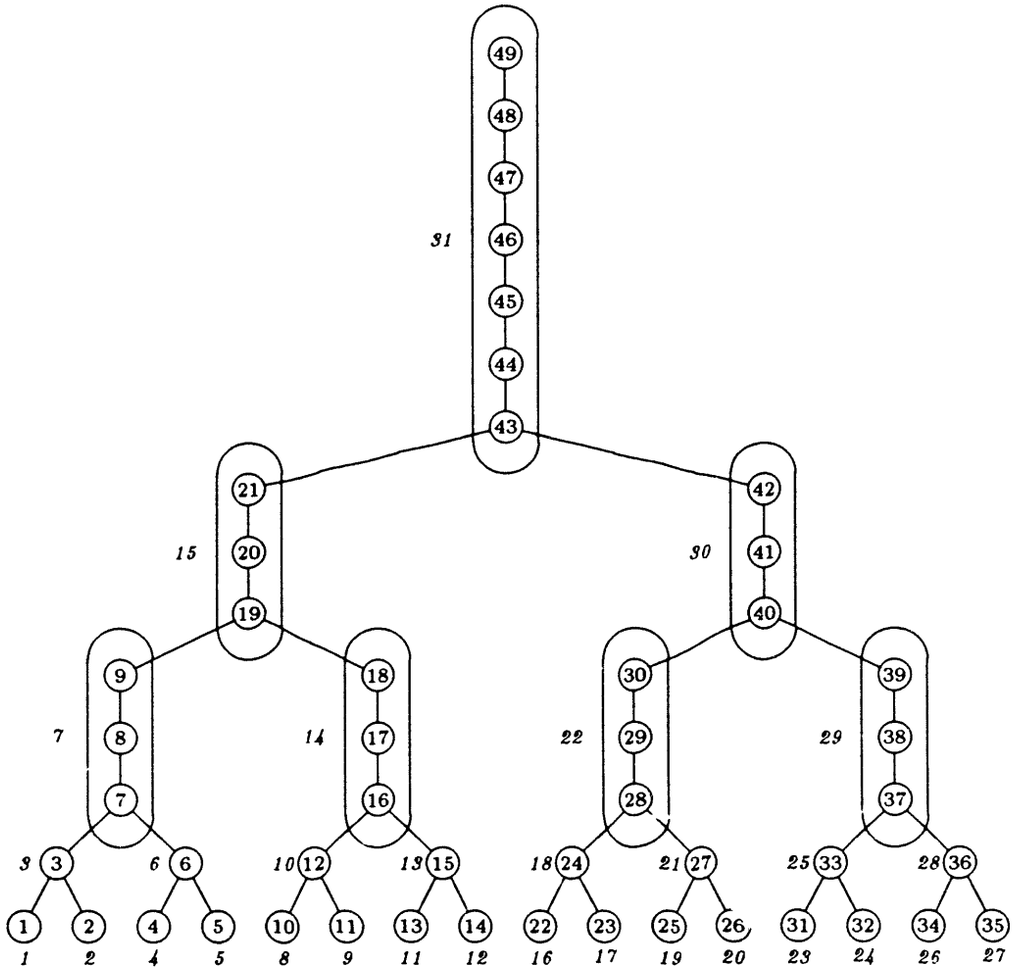


Figure 5: Elimination tree (and supernode elimination tree) for the matrix shown in Figure 4. Ovals enclose supernodes that contain more than one node. Nodes not enclosed by an oval are singleton supernodes. Italicized numbers label supernodes.

### 2.3.1. Left-looking sup-col Cholesky factorization

The basic idea behind the left-looking **sup-col** Cholesky algorithm is very simple. Let $K = \{p, p+1, \ldots, p+q\}$ be a supernode[2] in $\boldsymbol{L}$ and consider the computation of $\boldsymbol{L}_{*,j}$ for some $j > p+q$. It follows from the definition of supernodes that column $\boldsymbol{A}_{*,j}$ will be modified by *no* columns of $K$ or *every* column of $K$. Previous studies [7,26,27,28] have demonstrated that this observation has important ramifications for the performance of left-looking sparse Cholesky factorization. Loosely speaking, when used to update the target column $\boldsymbol{L}_{*,j}$, the columns in a supernode $K$ can now be treated as a single unit (or *block column*) in the computation. Since the columns in a supernode share the same sparsity structure below the dense diagonal block, modification of a particular column $j > p+q$ by these columns can be accumulated in a work vector using *dense* vector operations, and then applied to the target column using a single *sparse* vector operation that employs indirect addressing. Moreover, the use of *loop unrolling* in the accumulation, as described in [10], reduces memory traffic.

In Figure 6, we present the left-looking **sup-col** Cholesky factorization algorithm.

---

$\boldsymbol{t} \leftarrow 0$
**for** $J = 1$ **to** $N$ **do**
  Scatter $J$'s relative indices into *indmap*.
  **for** $j \in J$ (in order) **do**
    **for** $K$ such that $\boldsymbol{L}_{j,K} \neq 0$ **do**
      $\boldsymbol{t} \leftarrow cmod(j, K)$
      Assemble $\boldsymbol{t}$ into $\boldsymbol{L}_{*,j}$ using *indmap*
        while simultaneously setting $\boldsymbol{t}$ to zero.
    $cmod(j, J)$
    $cdiv(j)$

Figure 6: Left-looking **sup-col** Cholesky factorization algorithm.

---

The reader will find a more detailed implementation of the algorithm presented in [26]. In order to keep the notation simple, $K$ is to be interpreted in one of two different senses, depending on the context in which it appears. In one context (e.g., line 4 of Figure 6), $K$ is interpreted as the set of columns in the supernode, i.e., $K = \{p, p+1, \ldots, p+q\}$. In the second line of the algorithm, the supernodes are treated as an ordered set of loop indices $1, 2, \ldots, K, \ldots, N$, where $K < J$ if and only if $p < p'$, where $p$ and $p'$ are the first columns of $K$ and $J$, respectively. This dual-purpose notation is also illustrated in Figure 4, where the supernode labels are written over the diagonal entries, yet we can still write $30 = \{40, 41, 42\}$, for example. We denote both the last supernode and the number of supernodes by $N$.

[2]Throughout the remainder of the report the numbers designating a supernode will be italicized and the letters denoting a supernode will be capitalized.

Suppose $K = \{p, p+1, \ldots, p+q\}$. Whenever $j > p+q$ and $l_{j,p+q} \neq 0$, the task $cmod(j, K)$ consists of the operations $cmod(j, k)$, where $k = p, p+1, \ldots, p+q$. When $j \in K$, $cmod(j, K)$ consists of the operations $cmod(j, k)$, for $k = p, p+1, \ldots, j-1$. We let $\boldsymbol{L}_{j,K}$ denote the 1 by $|K|$ submatrix in $\boldsymbol{L}$ induced by row $j$ and the columns in $K$.

The indirect addressing scheme used by the algorithm is as follows. The indices in each list $Struct(\boldsymbol{L}_{*,j})$ are sorted in ascending order during the preprocessing stage (i.e., symbolic factorization). For each row index $i \in Struct(\boldsymbol{L}_{*,j})$, the corresponding *relative index* is the position $\ell$ of $i$ relative to the bottom of the list. For example, $\ell = 0$ for the last index in the list, $\ell = 1$ for the next-to-the-last one, and so forth. For each supernode $J = \{p, p+1, \ldots, p+q\}$, define

$$Struct(\boldsymbol{L}_{*,J}) = \{p\} \cup Struct(\boldsymbol{L}_{*,p}).$$

The relative position $\ell$ of each row index $i \in Struct(\boldsymbol{L}_{*,J})$ is stored in an $n$-vector *indmap* as follows: $indmap[i] \leftarrow \ell$. First the update $cmod(j, K)$ is accumulated in a work vector $\boldsymbol{t}$ whose length is the number of nonzero entries in the update. That is, the update is computed and stored as a *dense* vector $\boldsymbol{t}$. Then the algorithm assembles (scatter-adds) $\boldsymbol{t}$ into factor storage, using $indmap[i]$ to map each active row index $i \in Struct(\boldsymbol{L}_{*,K})$ to the appropriate location in $\boldsymbol{L}_{*,j}$ to which the corresponding component of $\boldsymbol{t}$ is added. The notion of relative indices apparently was first proposed by Schreiber [30].

### 2.3.2. Multifrontal Cholesky factorization

The multifrontal method, introduced by Duff and Reid in [15], is well documented in the literature. With much of its work performed within dense frontal matrices, this method has proven to be extremely effective on vector supercomputers [1,3,7,9]. Moreover, the multifrontal method is naturally expressed and implemented as a block method, and several of the advantages it derives from block matrix operations have already been explored in the literature: e.g., its ability to reuse data in fast memory [1,27] and its ability to perform well on machines with virtual memory and paging [22]. Implementation of the multifrontal method is more complicated and involves more subtleties than does any of the left-looking Cholesky variants. For the purposes of this report, it is adequate to restrict our presentation to an informal outline of the method. For a detailed survey of the multifrontal method and the techniques required for an efficient implementation, the reader should consult Liu [24]. The following paragraphs discuss the informal statement of the algorithm, found in Figure 7.

The outer loop of the supernodal multifrontal algorithm processes the supernodes $1, 2, \ldots, N$, where the supernodes have been renumbered by a postorder traversal of the supernodal elimination tree. After moving the required columns of $\boldsymbol{A}_{*,J}$ into the leading columns of $J$'s dense *frontal matrix* $\boldsymbol{F}_J$, the algorithm pops from the *update matrix stack* an update matrix $\boldsymbol{U}_K$ for each child $K$ of $J$ in the supernodal elimination tree, and assembles these accumulated update columns into $\boldsymbol{F}_J$. (The postordering enables the use of a simple and efficient stack for the update matrices.) The update matrix $\boldsymbol{U}_K$ is a dense matrix containing *all* updates destined for ancestors of $K$ from columns in the subtree of the supernodal elimination tree rooted at $K$. The assembly

Zero out the update matrix stack.
**for** $J = 1$ **to** $N$ (in postorder) **do**
  Move $\boldsymbol{A}_{*,J}$ into $\boldsymbol{F}_J$.
  **for** $K \in children(J)$ on top of the stack **do**
    Pop $\boldsymbol{U}_K$ from stack.
    While zeroing out $\boldsymbol{U}_K$, assemble $\boldsymbol{U}_K$ into $\boldsymbol{F}_J$.
  Within $\boldsymbol{F}_J$,
    compute the columns of $\boldsymbol{L}_{*,J}$ ($cdiv(J)$),
    and compute all update columns from $\boldsymbol{L}_{*,J}$
    (i.e., $cmod(k, J)$, where $k \in Struct(L_{*,J}) - J$).
  While zeroing out the vacated locations occupied by $\boldsymbol{F}_J$,
    move the new factor columns from $\boldsymbol{F}_J$ to $\boldsymbol{L}_{*,J}$.
  While zeroing out any vacated locations occupied by $\boldsymbol{F}_J$,
    move $\boldsymbol{U}_J$ to the top of the stack.

Figure 7: Supernodal multifrontal Cholesky factorization algorithm.

operation adds each entry of $\boldsymbol{U}_K$ to the corresponding entry of $\boldsymbol{F}_J$. These are sparse operations requiring indirect indexing because an update matrix generally modifies a *proper subset* of the entries in the target frontal matrix. These are the only sparse operations required by the multifrontal method.

Now with all the necessary data accumulated in $\boldsymbol{F}_J$, the next step in the main loop applies dense left-looking Cholesky factorization to the first $|J|$ columns in $\boldsymbol{F}_J$ (which we will call a $cdiv(J)$ operation) to compute the block column $\boldsymbol{L}_{*,J}$, and then accumulates in the trailing columns of $\boldsymbol{F}_J$ all column updates $cmod(k, j)$, where $j \in J$ and $k \in Struct(\boldsymbol{L}_{*,J}) - J$. At this point, the leading $|J|$ columns of $\boldsymbol{F}_J$ contain the columns of $\boldsymbol{L}_{*,J}$, and the other columns have accumulated every update column for ancestors of $J$ contributed by $J$ and its descendants in the supernodal elimination tree. The algorithm then moves the newly-computed columns to the appropriate location in the data structure for $L$, moves the update matrix $\boldsymbol{U}_J$ down onto the top of the stack, and proceeds with the next step of the major loop.

Three issues will occupy our attention when we take up the multifrontal algorithm again in Section 4. First, since all updates from the columns in $\boldsymbol{L}_{*,J}$ are computed immediately after the new factor columns are computed, the multifrontal method provides the opportunity for optimal reuse of columns loaded in cache. Second, the costs of data movement overhead are potentially significant. We are referring here to the movement of matrix columns between each frontal matrix and $\boldsymbol{L}$'s data structure, and the movement of each update matrix from the location in work storage where it was computed to its storage-saving location at the top of the stack. This issue is of particular concern on machines with cache, where moving large amounts of data in this manner will cause expensive cache misses not incurred by the left-looking algorithms. Third, we will be concerned with the amount of storage required for the stack of update

matrices, an issue that has received considerable attention in past studies [3,21,24].

## 3. Left-looking sup-sup Cholesky factorization

The idea behind the left-looking **sup-sup** Cholesky factorization algorithm is simple: The $cmod(j, K)$ operation is blocked one level higher, creating a supernode-to-supernode block-column updating operation $cmod(J, K)$ around which the new algorithm is constructed. The $cmod(J, K)$ operation performs $cmod(j, K)$ for every column $j \in J$ updated by the columns of $K$ (a BLAS3 operation). The idea of constructing a sparse-Cholesky algorithm around this operation is not new. Ashcraft and the second author wrote a left-looking **sup-sup** sparse Cholesky factorization code, which was mentioned in [7], but was not presented there. The indexing scheme they used, however, was unnecessarily complex. Though efficient, it had the side-effect of destroying the row indices of the nonzeros in $L$ so that they had to be recomputed later for use during the triangular solution phase or any future factorizations of matrices with the same structure. For these reasons, Ashcraft and the second author ultimately concluded that their implementation was unacceptable. Ashcraft recently sketched out a high-level version of the algorithm in a report on a different topic [4]. He has also created a single-parameter hybrid sparse Cholesky algorithm that performs a left-looking **sup-sup** factorization when the parameter takes on one extreme value, and performs a supernodal multifrontal factorization when it takes on the opposite extreme value [5]. The left-looking **sup-sup** approach was proposed again by the authors [26] as a promising candidate for parallelization on shared-memory multiprocessors. Parts of this work are steps toward completing the goals stated in the conclusion of that report. Recently and independently, Rothberg and Gupta have examined the caching behavior of three block Cholesky factorization algorithms, including the multifrontal and left-looking **sup-sup** methods [29].

The following paragraphs discuss the left-looking **sup-sup** Cholesky factorization algorithm in Figure 8 and its more basic implementation issues. One new item of notation is introduced; we let $\boldsymbol{L}_{J,K}$ denote the $|J|$ by $|K|$ submatrix in $\boldsymbol{L}$ induced by the members of $J$ and the members of $K$.

The bulk of the work is performed within the $cmod(J, K)$ and $cdiv(J)$ operations. The underlying matrix-matrix multiplication subroutine that performs most of the work in the implementation is used by the block multifrontal code as well, enabling a fair comparison of the two approaches. As in the left-looking **sup-col** approach, the update columns are accumulated in work storage. Naturally, far more work storage is required to accumulate the $cmod(J, K)$ updates than is required to accumulate the $cmod(j, K)$ updates, which consists of a single dense column no larger than the column of $\boldsymbol{L}$ with the most nonzero entries. This storage overhead will receive further attention in Sections 4 and 5.

Another distinction between the left-looking **sup-col** and **sup-sup** algorithms is that the **sup-sup** algorithm must compute the number of columns of $J$ to be updated by the columns of $K$, which it does by searching for all row indices $i \in J \cap Struct(\boldsymbol{L}_{*,K})$ in $K$'s sorted index list.

The algorithm handles indirect addressing in much the same way that the **sup-col**

$\boldsymbol{T} \leftarrow 0$
**for** $J = 1$ **to** $N$ **do**
    Scatter $J$'s relative indices into *indmap*.
    **for** $K$ such that $\boldsymbol{L}_{J,K} \neq 0$ **do**
        Compute the number of columns of $J$ to be updated by the columns of $K$.
        $\boldsymbol{T} \leftarrow cmod(J, K)$
        Gather $K$'s indices relative to $J$'s structure from *indmap* into *relind*.
        Using *relind*, assemble $\boldsymbol{T}$ into $\boldsymbol{L}_{*,J}$,
            while simultaneously restoring $\boldsymbol{T}$ to zero.
    $cdiv(J)$

Figure 8: Left-looking **sup-sup** Cholesky factorization algorithm.

algorithm in Figure 6 does, with one key difference which generally improves its efficiency. (See Section 2.3.1 for other details about the indexing scheme.) The **sup-sup** algorithm gathers the indices of $K$ *relative to* $J$ from *indmap* into a temporary vector *relind*: each active row index $i \in Struct(\boldsymbol{L}_{*,K})$, is replaced by *indmap*$[i]$ in the integer vector *relind*. This single gather operation provides the indexing information for assembling the entire block update into factor storage (i.e., the storage that will contain $\boldsymbol{L}_{*,J}$). The **sup-col** algorithm essentially has to repeat this gather operation each time it assembles a $cmod(j, K)$ update ($j \in J$) into factor storage.

## 4. Implementation details and options

Section 5 reports performance statistics for implementations of the multifrontal and the left-looking **sup-sup** Cholesky factorization algorithms on several powerful uniprocessor computing systems. Our Fortran codes have not been tuned for performance on any *specific machine* except for our choice of the level of loop-unrolling. To run efficiently on some of these machines, however, our implementations cannot afford to ignore other architectural considerations altogether. Unless they make effective use of data (i.e., columns of the matrix) once they have been loaded into cache, their performance will be severely penalized by an excessive number of cache misses. Thus our implementations must be designed with this goal in mind. Our codes require the cache size on each machine to reuse cached data effectively (see Section 4.1). The cache size and the level of loop-unrolling are the *only* machine-dependent parameters in our codes. Other implementation options and enhancements, which are entirely independent of the computer architecture, are also discussed in this section.

### 4.1. Reuse of data in cache

Consider the computation of a $cmod(J, K)$ update during the left-looking **sup-sup** Cholesky factorization. Suppose the operation updates $q$ columns of $J$ with the columns

of $K$. The number of columns updated may be as few as 1 or as many as $|J|$. We can compute $cmod(J, K)$ as a sequence of `sup-col` updates $cmod(j, K)$ for the $q$ columns $j \in J$. If the columns of $K$, which happen to be stored contiguously in main memory, fit into cache memory, then the first $cmod(j, K)$ loads the columns of $K$ into cache, while the following $q - 1$ *cmod*'s will have extremely fast access to this data because it is already in cache.

Quite often, however, the columns of a supernode do not fit into the 32K or 64K caches used on current workstations. This can dramatically increase the number of cache misses associated with the final $q - 1$ *cmod*'s, as the columns of $K$ overwrite one another as they are repeatedly read into cache. To avoid this problem, the algorithm partitions large supernodes into "panels" of contiguous columns that fit into the cache, as Rothberg and Gupta have done in their studies [27,28,29]. If $K$ has been partitioned into two panels, then the $cmod(J, K)$ update is performed by applying the *cmod*'s from the first panel to the $q$ target columns of $J$, then applying the *cmod*'s from the second panel to the $q$ target columns of $J$. We use essentially the same strategy to increase the reuse of data in cache by our multifrontal codes. This simple strategy has proven effective for the problems, machines, and factorization methods used in our tests. Extremely large problems, however, may require more complicated techniques that involve both horizontal and vertical partitioning, and perhaps even sweeping changes in the data structure used to store $\boldsymbol{L}$. The reader should consult Rothberg and Gupta [29] for a thorough discussion of these and many other issues associated with improving reuse of data in cache by both the multifrontal and the left-looking `sup-sup` sparse Cholesky algorithms.

### 4.2. Traversing row-structure sets

The left-looking `col-col` algorithm needs access to the row-structure sets $\mathcal{R}_j = \{k : \boldsymbol{L}_{j,k} \neq 0\}$ (see Figure 2). These row-structure sets must be computed from or traversed within the strictly column-oriented data structure used by the algorithm. By far the most commonly used method is to maintain the row-structure sets as linked lists within a single integer $n$-vector. Every column belongs to one and only one row-structure list at any given time during the course of the factorization. After a column update is completed, the column is placed in the list belonging to the next column it will modify. Details of this approach can be found in George and Liu [19, pp. 152–155] and in Ng and Peyton [26]. The same technique applies to the row-structure sets for the left-looking `sup-col` and `sup-sup` algorithms.

There is another way to determine the row-structure sets in the left-looking `col-col` algorithm, which relies on the fact that each row-structure set $\mathcal{R}_j$ is a pruned subtree of the elimination tree [20,30]. Consequently, if the elimination tree is made available to the factorization algorithm, each member of $\mathcal{R}_j$ can be visited by performing a depth-first traversal of the appropriate pruned subtree. Implementation details can be found in Schreiber [30]. Again, the same technique applies to the row-structure sets for the left-looking `sup-col` and `sup-sup` algorithms. This approach is particularly attractive in a parallel implementation of the left-looking factorization algorithms for shared-memory multiprocessor systems since it eliminates the need for critical sections

when manipulating the row-structure sets. We are currently pursuing this idea.

For the **sup-col** algorithm, our tests indicate that the total factorization time using the tree-traversal technique is slightly larger than that using the linked-list approach. However, for the **sup-sup** algorithm, the difference in total factorization time using the two approaches is negligible because the total time required to traverse the row-structure sets is extremely small in this algorithm for both techniques. Because overall factorization times differ by so little when the two approaches are compared, we have not included timing results for the more complicated of the two (the tree-traversal method) in Section 5. The important point to note is that either approach can be used in the **sup-sup** algorithm, and moreover, we believe that the tree-walking technique may ultimately be preferable in a parallel implementation for shared-memory multiprocessors [26].

### 4.3. Enhancements to the multifrontal method

The size of the stack of update matrices in the multifrontal method is a major issue associated with this method. A large stack obviously requires greater storage; perhaps not so obvious is that a large stack usually creates a great deal of overhead data movement that can erode efficiency. We have implemented two variants of the multifrontal algorithm. The first is a straightforward implementation of the algorithm in Figure 7. One standard enhancement has been incorporated into our basic multifrontal code. Using a technique introduced by Liu [21], we have reordered the children of each parent in the supernodal elimination tree to minimize the storage requirement for the stack. This section describes the techniques incorporated into our enhanced version of the multifrontal method.

We are aware of multifrontal implementations [32] that compute the new factor columns $\boldsymbol{L}_{*,J}$ in factor storage rather than in $\boldsymbol{F}_J$, and then compute only the update matrix $\boldsymbol{U}_J$ within the frontal matrix $\boldsymbol{F}_J$. This simple change reduces the size of the frontal matrix and eliminates the need to move matrix columns back and forth between factor storage and the frontal matrix. We have implemented this technique, and also further pursued the idea of reducing stack storage and limiting data movement by incorporating updates into factor storage as early as possible. More specifically, we have incorporated the following two techniques into our enhanced code.

First, let $P$ be the parent of $J$ in the supernodal elimination tree. We say that $J$ is *dense relative to* $P$ if

$$Struct(\boldsymbol{L}_{*,P}) \subseteq Struct(\boldsymbol{L}_{*,J}).$$

If $J$ is dense relative to $P$, then the update $cmod(P, J)$, which would normally fill the leading columns of $\boldsymbol{U}_J$, can be applied directly to $\boldsymbol{L}_{*,P}$, the columns of $P$ in factor storage. This shrinks the size of the update matrix $\boldsymbol{U}_J$, and thus reduces data movement when $\boldsymbol{U}_J$ is ultimately moved to its final position at the top of the stack. Since this condition usually holds for the root supernode and one or more of its children, both of which usually have very large frontal matrices, this simple enhancement can save a lot of storage.

The second technique pushes this idea a bit further. Consider the update matrix $\boldsymbol{U}_J$, and again consider $J$'s parent $P$. For the multifrontal method, the relative indices

of each child with respect to its parent have been computed in advance. The relative indices used to assemble $\boldsymbol{U}_J$ into $\boldsymbol{F}_P$ can also be used to assemble the columns of $\boldsymbol{U}_J$ destined for $\boldsymbol{L}_{*,P}$ directly into factor storage. But there is no reason to limit this technique to the parent just because only the indices relative to $P$ are available. If $J$ happens to update its grandparent supernode $P'$, then $J$'s indices relative to $P'$ can be obtained by gathering the appropriate indices of $P$ (relative to $P'$) into an integer work vector *relind*, and then using them to assemble the appropriate columns of $\boldsymbol{U}_J$ into factor storage (i.e., $\boldsymbol{L}_{*,P'}$). If $J$ happens to update its great-grandparent $P''$, then the process can be repeated with the old indices in *relind* (relative to $P'$) used to gather some of the indices of $P'$ (relative to $P''$) into *relind*, giving us the indices of $J$ relative to $P''$. The enhanced algorithm continues this process until it encounters the root of the supernodal elimination tree or an ancestor of $J$ that is *not* updated by the columns of $J$. Each assembly into factor storage reduces the amount of storage required for the reduced version of $\boldsymbol{U}_J$ and the amount of time required to move it to the top of the stack. The only overhead computation required, the sequence of integer gather operations, is negligible compared to the savings in data movement, and this technique is surprisingly effective at reducing the stack storage requirement, as we shall see in Section 5.

Lastly, one commonly used stack-reduction technique is the *extension in place* of the update matrix for the child on top of the stack into the parent's new frontal matrix, which is initially set to zero. Liu [21] points out that this technique is used in the Harwell MA27 code, and Ashcraft [3] reports that overlapping the new frontal matrix with the topmost update matrix in this fashion saves a surprising 15–27% in stack storage for his test problems. We have incorporated it into our enhanced multifrontal code.

### 4.4. Refinements for left-looking `sup-col` and sup-sup Cholesky

Three refinements have been incorporated into our implementations of the left-looking `sup-col` and `sup-sup` Cholesky factorization algorithms, several of which concern the incorporation of update columns that are dense relative to the target column directly into factor storage. First, whenever $K$ has only one column, the `sup-col` (`sup-sup`) code accumulates the column modification $cmod(j, K)$ ($cmod(J, K)$) directly into factor storage, avoiding use of the real work vector $\boldsymbol{t}$ ($\boldsymbol{T}$) altogether. This is extremely simple to implement, avoids some useless data movement, and is valuable for problems with many singleton supernodes. Second, all column modifications where the source and target columns come from the same supernode are performed as dense updates incorporated directly into factor storage using no indirect indexing. That is, they are performed as a dense update would be performed. Third, whenever the length of update columns from $K$ matches the length of a target column(s) from $J$, it is also handled as a dense update. No indirect indexing is used, and the update is accumulated directly into factor storage.

Two minor refinements are incorporated into the left-looking `sup-sup` Cholesky algorithm only. Unlike the `sup-col` algorithm, the block algorithm explicitly computes and records relative indices as they are needed. By taking the difference between the

first and the last of these indices and checking the difference against the length of the target, the algorithm is now capable of checking for *all remaining* dense updates, thereby avoiding some data movement and indirect addressing normally associated with these operations. Finally, note that the size of the block of work storage $\boldsymbol{T}$ needed by the algorithm is the size of the largest block update $cmod(J, K)$ generated by the algorithm that is *not dense* relative to its target factor columns. In practice the children of the root supernode are usually dense relative to their parent, and moreover the largest block update is often found among the block updates they generate for the root. Consequently, the practice of accumulating dense block updates directly into factor storage often reduces the amount of work storage needed for the algorithm.

## 5. Performance results

In this section we compare the performance of various sparse Cholesky factorization algorithms discussed in this paper, which include

- left-looking `col-col` Cholesky,
- left-looking `sup-col` Cholesky,
- left-looking `sup-sup` Cholesky,
- a basic multifrontal method, and
- an enhanced multifrontal method.

All algorithms were coded in Fortran and all floating-point operations were performed in double precision, except on the Cray Y-MP. The code for left-looking `col-col` Cholesky was taken from SPARSPAK [8]. All codes were compiled with optimization turned on and were run on a vector supercomputer and a number of high-performance scientific workstations. It should be noted that identical code was run on each machine, except for the level of loop-unrolling used in the block update routines.

The machines used in the experiments include

- an IBM RS/6000 model 530,
- a DEC 5000,
- a Stardent P3000, and
- one processor of a Cray Y-MP.

Each of the workstations has 64 kilobytes of cache memory. The cache on the IBM RS/6000 is 4-way set-associative, while those on the DEC 5000 and Stardent P3000 are direct-mapped. The cache line size on the IBM RS/6000 is 128 bytes, compared to 4 bytes on the DEC 5000 and Stardent P3000. The IBM RS/6000 and DEC 5000 have 16 megabytes of main memory, while the Stardent P3000 has 32 megabytes. Since we restricted our tests to problems that fit into the main memory, there was no paging, and hence differences in memory size had no effect on performance. Both the DEC 5000

and Stardent P3000 use the same central processing unit (MIPS 3000), but they have different floating-point coprocessors. The Stardent P3000 moreover has special vector floating-point hardware that can be enabled or disabled during code compilation. The DEC 5000 and Stardent P3000 (with vectorization disabled) are similar in so many respects that we expect similar performance on these machines.

The vector supercomputer we used, the Cray Y-MP, has no memory hierarchy and has enough main memory for the largest of our test problems. It is also worth noting that this machine, as a rule, performs floating-point arithmetic far more efficiently than integer arithmetic, in contrast to the workstations where integer and floating-point performance is better balanced.

As we pointed out in previous sections, loop unrolling was employed in our implementation of the $cmod(j, K)$ and $cmod(J, K)$ block update operations. The optimal level of loop unrolling varies from machine to machine. In our experiments, we tried level-$p$ loop unrolling, for $p = 1$, 2, 4 and 8. To limit the amount of data presented in our tables, we report data for only the level of loop unrolling that performed best on the specific machine under consideration. The best level was $p = 4$ for the DEC 5000 and Cray Y-MP, and $p = 8$ for the IBM RS/6000 and Stardent P3000.

Almost all the test problems were taken from the Harwell-Boeing Test Collection [13], which is widely used in testing and evaluating sparse matrix algorithms. The problems we selected and some of their characteristics are provided in Tables 1 and 2, respectively. To ensure that no paging occurred, only the small to medium size problems were run on the workstations. All problems were run on the Cray Y-MP.

| problem | brief description |
|---|---|
| BCSSTK13 | Stiffness matrix – fluid flow generalized eigenvalues |
| BCSSTK14 | Stiffness matrix – roof of Omni Coliseum, Atlanta |
| BCSSTK15 | Stiffness matrix – module of an offshore platform |
| BCSSTK16 | Stiffness matrix – Corp. of Engineers dam |
| BCSSTK17 | Stiffness matrix – elevated pressure vessel |
| BCSSTK18 | Stiffness matrix – R.E.Ginna nuclear power station |
| BCSSTK23 | Stiffness matrix – portion of a 3D globally triangular bldg |
| BCSSTK24 | Stiffness matrix – winter sports arena |
| BCSSTK25 | Stiffness matrix – 76 story skyscraper |
| BCSSTK29 | Stiffness matrix – buckling model of the 767 rear bulkhead |
| BCSSTK30 | Stiffness matrix – off-shore generator platform (MSC NASTRAN) |
| BCSSTK31 | Stiffness matrix – automobile component (MSC NASTRAN) |
| BCSSTK32 | Stiffness matrix – automobile chassis (MSC NASTRAN) |
| BCSSTK33 | Stiffness matrix – pin boss (auto steering component), solid elements |
| NASA1824 | Structure from NASA Langley, 1824 degrees of freedom |
| NASA2910 | Structure from NASA Langley, 2910 degrees of freedom |
| NASA4704 | Structure from NASA Langley, 4704 degrees of freedom |

Table 1: List of test problems.

| problem | $n$ | $\|A\|$ | $\|L\|$ | $\mu(L)$ | $N$ | flops |
|---|---|---|---|---|---|---|
| BCSSTK13 | 2,003 | 83,883 | 271,671 | 28,621 | 599 | 58,550,598 |
| BCSSTK14 | 1,806 | 63,454 | 112,267 | 17,508 | 503 | 9,793,431 |
| BCSSTK15 | 3,948 | 117,816 | 651,222 | 61,614 | 1,295 | 165,035,094 |
| BCSSTK16 | 4,884 | 290,378 | 741,178 | 50,365 | 691 | 149,100,948 |
| BCSSTK17 | 10,974 | 428,650 | 1,005,859 | 94,225 | 2,595 | 144,269,031 |
| BCSSTK18 | 11,948 | 149,090 | 662,725 | 116,807 | 7,438 | 140,907,823 |
| BCSSTK23 | 3,134 | 45,178 | 420,311 | 49,018 | 1,522 | 119,155,247 |
| BCSSTK24 | 3,562 | 159,910 | 278,922 | 22,331 | 414 | 32,429,194 |
| BCSSTK25 | 15,439 | 252,241 | 1,416,568 | 205,513 | 7,288 | 283,732,315 |
| BCSSTK29 | 13,992 | 619,488 | 1,694,796 | 174,770 | 3,231 | 393,045,158 |
| BCSSTK30 | 28,924 | 2,043,492 | 3,843,435 | 229,670 | 3,689 | 928,323,809 |
| BCSSTK31 | 35,588 | 1,181,416 | 5,308,247 | 330,896 | 8,304 | 2,550,954,465 |
| BCSSTK32 | 44,609 | 2,014,701 | 5,246,353 | 374,507 | 6,927 | 1,108,686,016 |
| BCSSTK33 | 8,738 | 591,904 | 2,546,802 | 124,532 | 1,201 | 1,203,491,786 |
| NASA1824 | 1,824 | 39,208 | 73,699 | 12,587 | 527 | 5,160,949 |
| NASA2910 | 2,910 | 174,296 | 204,403 | 25,170 | 599 | 21,068,943 |
| NASA4704 | 4,704 | 104,756 | 281,472 | 35,339 | 1,245 | 35,003,786 |

Table 2: Characteristics of test problems.

Legend:

$n$: number of equations,

$|A|$: number of nonzeros in $A$,

$|L|$: number of nonzeros in $L$, including the diagonal,

$\mu(L)$: number of row subscripts required to represent the supernodal structure of $L$,

$N$: number of fundamental supernodes in $L$,

flops: number of floating-point operations required to compute $L$.

The tables presented in the following subsections contain the times required to run the factorization algorithms on several different machines. All execution times are in seconds. For machines that have cache memory, the notation `method(`$s$`)` is used, where `method` is either **sup-sup** or **mf** (multifrontal). When $s = 0$, supernodes are not subdivided into panels; when $s > 0$, large supernodes are subdivided into panels that fit into the $s$-kilobyte cache available on that machine. For example, on all the workstations $s = 64$ when the supernodes are subdivided. It is worth noting that all the test problems have many supernodes small enough to fit into cache, and both the multifrontal and left-looking **sup-sup** algorithms fully "reuse" the columns of such supernodes once they are loaded into cache, regardless of whether or not the larger supernodes have been subdivided to fit into cache.

### 5.1. IBM RS/6000

| problem | col-col | sup-col | sup-sup | | basic mf | | enhanced mf | |
|---|---|---|---|---|---|---|---|---|
| | | | (0) | (64) | (0) | (64) | (0) | (64) |
| BCSSTK13 | 7.33 | 3.59 | 3.22 | 3.04 | 3.58 | 3.39 | 3.32 | 3.10 |
| BCSSTK14 | 1.32 | .69 | .61 | .61 | .71 | .69 | .65 | .65 |
| BCSSTK15 | 20.40 | 9.68 | 8.77 | 8.08 | 9.49 | 8.78 | 8.98 | 8.32 |
| BCSSTK16 | 18.61 | 8.94 | 7.93 | 7.47 | 8.59 | 8.15 | 8.01 | 7.52 |
| BCSSTK18 | 17.86 | 9.30 | 8.58 | 8.07 | 9.39 | 9.08 | 8.99 | 8.47 |
| BCSSTK23 | 14.71 | 7.13 | 6.57 | 6.00 | 7.21 | 6.67 | 6.78 | 6.26 |
| BCSSTK24 | 4.28 | 2.03 | 1.76 | 1.72 | 1.92 | 1.88 | 1.78 | 1.74 |
| NASA1824 | .74 | .41 | .36 | .36 | .40 | .40 | .36 | .36 |
| NASA2910 | 2.81 | 1.46 | 1.24 | 1.23 | 1.36 | 1.36 | 1.26 | 1.25 |
| NASA4704 | 4.56 | 2.29 | 2.01 | 1.94 | 2.17 | 2.10 | 2.03 | 1.96 |

Table 3: Factorization times in seconds on IBM RS/6000.

Table 3 contains the execution times (in seconds) required by the various factorization methods on an IBM RS/6000 model 530. We make the following observations from these results.

First, we see that **sup-col** consistently reduces factorization times by roughly a factor of 2 over **col-col**. Part of this large improvement is due to reductions in memory traffic and indirect addressing, which are in turn due respectively to the loop unrolling and the dense matrix-vector multiplication used to implement the $cmod(j, K)$ operation. However, the improvement of **sup-col** over **col-col** observed on this machine is considerably larger than that observed on the other workstations, which obtain the same reductions in memory traffic and indirect indexing. We believe that the large cache line size (128 bytes) on the IBM RS/6000 is largely responsible for this phenomenon. The memory-access pattern of the **col-col** algorithm is far more disordered and contains far fewer stride one vector reads and writes than that of the **sup-col** algorithm. As a result, the **sup-col** algorithm is far more likely to use most or all of the floating-point numbers in a line as it is loaded into cache. Consequently, it often uses several (up to $16 = 128/8$) double precision numbers at the cost of a single cache miss.

We see that **sup-sup(0)** usually improves performance over **sup-col** by 10–15%. This improvement is partly due to further reductions in the cost of indirect indexing and the integer overhead associated with the row-structure lists. It is likely however that most of the improvement is due to reuse of data in the cache when the supernodes are small enough.

By partitioning supernodes whose columns overflow the cache into panels of contiguous columns that fit into cache and moreover organizing the matrix-matrix multiplication operations to operate on these panels, data in cache is reused more effectively, and thus the amount of data moved to and from main memory is reduced. This leads to another 6–10% improvement in factorization times for **sup-sup(64)** when it is used on the medium- and large-sized problems in the test set. Smaller increases are obtained for the small problems because most or all of their supernodes already fit into cache. This improvement is quite modest compared to that observed on the other workstations. We further explore this issue in the next subsection.

As expected, subdividing supernodes into panels that fit into cache improves the performance of both the basic and enhanced multifrontal methods in much the same manner that it improves the performance of the **sup-sup** algorithm. Enhanced **mf** performs significantly better than basic **mf**, probably because the former method typically required much less data movement. Our implementation of enhanced **mf** is slightly less efficient than **sup-sup** because the former still requires more data movement than the latter, despite our efforts to minimize such movement. Where applicable, these observations hold true on the other machines as well.

One of the most widely used implementations of the multifrontal method is the MA27 routine in the Harwell library [14]. To verify that our implementations of this method are adequate for fair comparisons, we have compared their performance with that of the MA27 routine in Table 4. Since loop-unrolling and techniques for exploit-

| problem | **MA27** | basic **mf(0)** **level=1** | enhanced **mf(64)** **level=8** |
|---|---|---|---|
| BCSSTK13 | 5.88 | 4.98 | 3.10 |
| BCSSTK14 | 1.31 | .90 | .65 |
| BCSSTK15 | 15.38 | 13.44 | 8.32 |
| BCSSTK16 | 15.02 | 12.22 | 7.52 |
| BCSSTK18 | 14.95 | 12.76 | 8.47 |
| BCSSTK23 | 11.17 | 10.15 | 6.26 |
| BCSSTK24 | 3.74 | 2.65 | 1.74 |
| NASA1824 | 0.74 | .50 | .36 |
| NASA2910 | 2.88 | 1.81 | 1.25 |
| NASA4704 | 3.83 | 2.96 | 1.96 |

Table 4: Comparing 3 multifrontal methods: factorization times in seconds on IBM RS/6000 (basic **mf** does not use loop unrolling and enhanced **mf** uses level-8 loop unrolling).

ing cache memory have not been incorporated into MA27, the fairest comparison is between the first two columns of the table. The second column contains the times re-

quired by basic **mf** with no loop-unrolling and with no subdivision of the supernodes to improve cache usage. While this code outperforms MA27, the comparison is not really fair because of the additional cost of MA27's extremely flexible method for inputting the matrix entries. In any case, it is clear that our code is quite competitive. The last column demonstrates the value of the enhancements incorporated into our best implementation of the multifrontal method.

### 5.2. DEC 5000

| problem | col-col | sup-col | sup-sup | | basic mf | | enhanced mf | |
|---|---|---|---|---|---|---|---|---|
| | | | (0) | (64) | (0) | (64) | (0) | (64) |
| BCSSTK13 | 24.33 | 18.77 | 15.02 | 10.51 | 17.45 | 12.92 | 15.45 | 10.90 |
| BCSSTK14 | 3.50 | 2.53 | 1.84 | 1.84 | 2.40 | 2.40 | 1.96 | 1.97 |
| BCSSTK15 | 70.84 | 52.60 | 44.36 | 28.49 | 48.86 | 33.06 | 45.29 | 29.31 |
| BCSSTK16 | 61.10 | 47.13 | 36.29 | 25.90 | 40.85 | 30.41 | 36.70 | 26.27 |
| BCSSTK18 | 60.38 | 46.47 | 39.44 | 27.30 | 46.75 | 34.38 | 41.65 | 29.29 |
| BCSSTK23 | 50.86 | 38.80 | 34.30 | 21.53 | 38.75 | 25.86 | 35.36 | 22.49 |
| BCSSTK24 | 12.41 | 9.18 | 6.39 | 5.67 | 7.45 | 6.73 | 6.47 | 5.75 |
| NASA1824 | 1.87 | 1.36 | 1.04 | 1.04 | 1.32 | 1.32 | 1.07 | 1.07 |
| NASA2910 | 7.96 | 6.01 | 4.05 | 3.89 | 4.99 | 4.84 | 4.16 | 3.99 |
| NASA4704 | 14.01 | 10.60 | 7.69 | 6.28 | 8.89 | 7.47 | 7.82 | 6.41 |

Table 5: Factorization times in seconds on DEC 5000.

Table 5 contains factorization times for the various factorization methods on a DEC 5000. In contrast to the IBM RS/6000, the reduction in factorization time of **sup-col** over **col-col** is only 30–38%. (All percentages used in comparisons are relative to the smaller of the two times.) Due to the 4-byte cache line size on the DEC 5000, the contrasting memory access patterns of the **col-col** and **sup-col** algorithms do not incur, respectively, nearly as severe a penalty or nearly as great a performance boost as those noted earlier on the IBM workstation.

However, the improvement of the **sup-sup** algorithm over the **sup-col** algorithm is much larger on this machine. Largely due to the 4-byte cache line and the larger penalty associated with each cache miss (2 misses per floating-point number), the **sup-sup** algorithm generally obtains very large performance improvements over the **sup-col** algorithm, whose capacity to reuse data in cache is quite limited for the larger test problems. The **sup-sup(0)** algorithm improves performance over the **sup-col** algorithm by 13–30% for the larger problems and 31–48% for the smaller problems. The **sup-sup(64)** algorithm improves performance over the **sup-sup(0)** algorithm by 40–59% for the larger problems and 0–22% for the smaller problems. The cumulative improvement is 70–85% for the larger problems and 4–48% for the smaller problems. A more detailed look at the the effect of cache size and organization on the performance of both the **sup-sup** and multifrontal algorithms can be found in Rothberg and Gupta [29].

### 5.3. Stardent P3000 (without vectorization)

As mentioned earlier in this section, the Stardent P3000 and DEC 5000 have identical central processing units but different floating-point coprocessors. Thus, when vectorization is not used on the Stardent P3000, we expect performance to be quite similar on these two machines. The results in Tables 5 and 6 indicate that that is indeed the case,

| problem | col-col | sup-col | sup-sup | | basic mf | | enhanced mf | |
|---|---|---|---|---|---|---|---|---|
| | | | (0) | (64) | (0) | (64) | (0) | (64) |
| BCSSTK13 | 28.75 | 24.80 | 19.09 | 12.62 | 20.97 | 14.43 | 19.49 | 12.96 |
| BCSSTK14 | 3.82 | 3.17 | 2.15 | 2.15 | 2.53 | 2.54 | 2.28 | 2.29 |
| BCSSTK15 | 84.70 | 69.33 | 56.50 | 34.20 | 60.08 | 37.64 | 57.42 | 35.00 |
| BCSSTK16 | 72.25 | 62.84 | 45.67 | 31.09 | 49.11 | 34.45 | 46.26 | 31.33 |
| BCSSTK18 | 72.00 | 61.75 | 50.09 | 32.83 | 55.50 | 37.96 | 52.18 | 34.58 |
| BCSSTK23 | 60.66 | 51.08 | 44.01 | 25.85 | 47.65 | 29.34 | 45.12 | 26.81 |
| BCSSTK24 | 14.11 | 11.84 | 7.71 | 6.70 | 8.40 | 7.39 | 7.74 | 6.73 |
| NASA1824 | 2.03 | 1.69 | 1.21 | 1.21 | 1.39 | 1.39 | 1.23 | 1.24 |
| NASA2910 | 8.94 | 7.64 | 4.82 | 4.58 | 5.44 | 5.21 | 4.86 | 4.64 |
| NASA4704 | 16.34 | 13.87 | 9.45 | 7.46 | 10.25 | 8.26 | 9.54 | 7.55 |

Table 6: Factorization times in seconds on Stardent P3000 (without vectorization).

with one exception. For reasons we don't understand, loop-unrolling is considerably less effective on this machine than it is on the DEC 5000. With the exception of the `col-col` to `sup-col` comparison, the various methods compare with each other very much as they did on the DEC 5000.

### 5.4. Stardent P3000 (with vectorization)

| problem | col-col | sup-col | sup-sup | | basic mf | | enhanced mf | |
|---|---|---|---|---|---|---|---|---|
| | | | (0) | (64) | (0) | (64) | (0) | (64) |
| BCSSTK13 | 20.04 | 4.98 | 4.65 | 4.68 | 5.01 | 5.01 | 4.73 | 4.76 |
| BCSSTK14 | 3.78 | 1.18 | 1.13 | 1.13 | 1.26 | 1.26 | 1.20 | 1.21 |
| BCSSTK15 | 55.37 | 13.07 | 12.18 | 12.13 | 13.01 | 12.93 | 12.49 | 12.45 |
| BCSSTK16 | 51.09 | 12.68 | 11.48 | 11.51 | 11.98 | 12.05 | 11.42 | 11.49 |
| BCSSTK18 | 48.46 | 13.59 | 12.80 | 12.81 | 13.31 | 13.33 | 13.23 | 13.22 |
| BCSSTK23 | 39.75 | 9.62 | 9.03 | 9.01 | 9.74 | 9.65 | 9.29 | 9.26 |
| BCSSTK24 | 11.92 | 3.11 | 2.93 | 2.92 | 3.06 | 3.07 | 2.92 | 2.93 |
| NASA1824 | 2.10 | .74 | .73 | .73 | .75 | .75 | .72 | .71 |
| NASA2910 | 7.90 | 2.33 | 2.20 | 2.21 | 2.31 | 2.32 | 2.20 | 2.21 |
| NASA4704 | 12.76 | 3.58 | 3.36 | 3.37 | 3.49 | 3.49 | 3.36 | 3.37 |

Table 7: Factorization times in seconds on Stardent P3000 (with vectorization).

The Stardent P3000 has floating-point vector hardware, which can be enabled or disabled when the code is compiled. Table 7 contains factorization times for the various factorization methods with *vectorization turned on.* An important observation is that

subdividing the supernodes into panels that fit into the 64K cache has virtually no effect on performance. To avoid the complication of resolving cache misses during a vector operation, the vector hardware bypasses the cache altogether, and instead reads data directly from main memory in a pipelined fashion. This explains why paneling the supernodes is entirely ineffective in the **sup-sup** and the two multifrontal algorithms. It is worth noting, however, that reduced integer overhead and reduced indirect indexing in the **sup-sup** and multifrontal algorithms enable them to run faster than the **sup-col** algorithm. For instance, the **sup-sup** algorithm runs 5–10% faster than the **sup-col** algorithm (excluding the two smallest problems from consideration). Evidently, our implementation of the dense matrix update kernels performs well on the Stardent P3000's vector hardware. For example, **sup-sup** is about 3.8–4.5 times faster than **col-col** (again, excluding the two smallest problems from consideration).

### 5.5. Cray Y-MP

Unlike the workstations considered in previous subsections, the Cray Y-MP has no cache memory. Its floating-point hardware is extremely fast due to vector pipelining. We have run the codes on a Cray Y-MP, and the results are provided in Table 8. As

| problem | col-col | sup-col | sup-sup | basic mf | enhanced mf |
|---|---|---|---|---|---|
| BCSSTK13 | 0.84 | 0.42 | 0.36 | 0.38 | 0.38 |
| BCSSTK14 | 0.22 | 0.15 | 0.11 | 0.12 | 0.12 |
| BCSSTK15 | 2.22 | 1.02 | 0.90 | 0.96 | 0.95 |
| BCSSTK16 | 2.18 | 1.02 | 0.89 | 0.92 | 0.89 |
| BCSSTK17 | 2.46 | 1.29 | 1.07 | 1.11 | 1.09 |
| BCSSTK18 | 2.05 | 1.23 | 1.14 | 1.15 | 1.22 |
| BCSSTK23 | 1.56 | 0.75 | 0.67 | 0.72 | 0.72 |
| BCSSTK24 | 0.62 | 0.32 | 0.26 | 0.27 | 0.26 |
| BCSSTK25 | 4.20 | 2.42 | 2.13 | 2.13 | 2.20 |
| BCSSTK29 | 5.45 | 2.79 | 2.38 | 2.47 | 2.48 |
| BCSSTK30 | 12.73 | 5.52 | 4.96 | 5.12 | 4.99 |
| BCSSTK31 | 28.96 | 12.16 | 11.48 | 11.58 | 11.43 |
| BCSSTK32 | 15.98 | 7.38 | 6.47 | 6.63 | 6.49 |
| BCSSTK33 | 13.68 | 5.61 | 5.29 | 5.47 | 5.30 |
| NASA1824 | 0.14 | 0.11 | 0.09 | 0.08 | 0.09 |
| NASA2910 | 0.43 | 0.27 | 0.21 | 0.21 | 0.21 |
| NASA4704 | 0.65 | 0.40 | 0.33 | 0.32 | 0.33 |

Table 8: Factorization times in seconds on CRAY Y-MP.

observed in [7] and [26], **sup-col** generally outperforms **col-col** by roughly a factor of 2. The use of loop-unrolling, dense matrix-vector multiplication kernels, and the consequent large reductions in indirect addressing are responsible for these gains in performance. For medium to large problems, **sup-sup** outperforms **sup-col** by 6-21%. (The performance gains are larger for the smaller problems.) The improvement is due to reductions in the cost of the indirect indexing and other integer processing overhead. The differences in performance among **sup-sup**, basic **mf**, and enhanced **mf** are very

small.

### 5.6. Work storage requirements

The preceding subsections compare the *time* efficiency of the sparse Cholesky factorization algorithms under study on various high-performance uniprocessor computers. This subsection compares the *storage* efficiency of the various Cholesky factorization algorithms. More specifically, we computed the amount of auxiliary *floating-point* storage locations required by each method for accumulating column updates. Note that this ignores the floating-point storage required for the nonzero entries of $\boldsymbol{L}$, which is the same for each method. It also ignores the amount of *integer* work storage required, since it is the sum of a small number of quantities $\leq n$, where $n$ is the order of $\boldsymbol{A}$, and hence does not vary much from one method to the next.

The `col-col` and `sup-col` algorithms have the lowest auxiliary work storage requirement because the columns are computed one at a time. For `col-col`, a floating-point work array of length $n$ is needed to accumulate the updates. For the `sup-col` algorithm, the size of the floating-point work array is the maximum, over all columns $\boldsymbol{L}_{*,j}$, of the number of nonzero entries in $\boldsymbol{L}_{*,j}$. (Recall that an extra integer $n$-vector *indmap* is required to implement the indirect indexing scheme.)

The `sup-sup` and `mf` algorithms require more floating-point work storage. The two versions of the multifrontal method need auxiliary floating-point storage for stacking the update matrices. The `sup-sup` algorithm needs auxiliary floating-point storage to accumulate individual block updates $cmod(J, K)$. Thus, we are particularly interested in the storage requirements for `sup-sup` and `mf`.

Table 9 reports the floating-point work-storage requirements for each method, normalized as a *percentage* of the number of nonzeros in $\boldsymbol{L}$. As expected, the `sup-col` and `col-col` methods do indeed require the smallest amount of floating-point work storage. Without the enhancements to reduce the stack usage, the basic multifrontal method requires by far the most work storage. For two problems the size of the stack is roughly 60% of the "size" of $\boldsymbol{L}$. The enhanced multifrontal algorithm required far less floating-point work storage than the basic multifrontal algorithm requires, but still considerably more than the `sup-sup` algorithm requires.

## 6. Concluding remarks

We have studied three different left-looking sparse Cholesky factorization algorithms: the `col-col`, `sup-col` and `sup-sup` algorithms. The use of supernode-to-column updates in the `sup-col` algorithm (instead of the column-to-column updates in the `col-col` algorithm) reduces the amount of memory traffic and indirect addressing overhead. Our tests have shown the effectiveness of this well known technique on a wide range of high-performance uniprocessor computers. The use of supernode-to-supernode updates in the `sup-sup` algorithm further reduces the amount of memory traffic on machines with high-speed local memory, such as a cache. For our test problems, the `sup-sup` algorithm obtains virtually the same performance improvements via reuse of cached data that the multifrontal method obtains. Similar test results have

| problem | col-col | sup-col | sup-sup | basic mf | enhanced mf |
|---|---|---|---|---|---|
| BCSSTK13 | .74 | .14 | 8.63 | 58.70 | 15.80 |
| BCSSTK14 | 1.61 | .16 | 4.08 | 27.49 | 6.79 |
| BCSSTK15 | .61 | .07 | 5.91 | 30.44 | 7.79 |
| BCSSTK16 | .66 | .04 | 2.63 | 17.12 | 3.90 |
| BCSSTK17 | 1.09 | .03 | 1.90 | 12.43 | 2.70 |
| BCSSTK18 | 1.80 | .06 | 5.06 | 25.37 | 7.46 |
| BCSSTK23 | .75 | .12 | 12.18 | 60.07 | 17.31 |
| BCSSTK24 | 1.28 | .09 | 4.80 | 23.45 | 5.46 |
| BCSSTK25 | 1.09 | .03 | 2.34 | 11.81 | 3.27 |
| BCSSTK29 | .83 | .03 | 3.05 | 17.01 | 7.16 |
| BCSSTK30 | .75 | .02 | 1.69 | 11.22 | 2.46 |
| BCSSTK31 | .67 | .02 | 3.27 | 24.21 | 5.49 |
| BCSSTK32 | .85 | .01 | 1.10 | 8.50 | 2.02 |
| BCSSTK33 | .34 | .04 | 5.15 | 40.31 | 9.06 |
| NASA1824 | 2.47 | .22 | 4.51 | 39.66 | 8.59 |
| NASA2910 | 1.42 | .10 | 3.96 | 25.93 | 5.99 |
| NASA4704 | 1.67 | .10 | 6.81 | 29.16 | 8.72 |

Table 9: Floating-point work storage (% of $|L|$).

appeared in Rothberg and Gupta [29]. On machines without a cache, the **sup-sup** algorithm obtains modest improvements over the **sup-col** algorithm by further reducing the integer overhead and indirect indexing costs.

Although the performance of the various left-looking factorization algorithms is machine dependent, it is interesting to note that for three high-performance workstations (the IBM RS/6000, the DEC 5000, and the Stardent P3000 without vectorization), the **sup-sup** algorithm with subdivided supernodes is the most efficient algorithm, and often runs 2.5 times faster than the **col-col** algorithm. On the Stardent P3000 with vectorization, the **sup-sup** algorithm is roughly 4–4.5 times faster than the **col-col** algorithm.

For the test problems and workstations considered in this report, the enhanced multifrontal algorithm is slightly slower than the **sup-sup** algorithm (by roughly 5–10%). The results also indicate that the enhancements we have made to the multifrontal method greatly reduce the amount of auxiliary storage required for the stack and the amount of data movement required to stack the update matrices. The work-storage requirement in **sup-sup**, however, remains smaller than that in the enhanced multifrontal method.

One of the goals in this study is to identify the "best" sequential sparse Cholesky factorization algorithm. This algorithm will be used to evaluate the performance of various parallel sparse Cholesky factorization methods. Based on our results, we conclude that the left-looking **sup-sup** algorithm is the most efficient algorithm, both in terms of its execution time and work-storage requirement. Parallel versions of left-looking **col-col** and **sup-col** algorithms have appeared in [17] and [26], respectively. Parallel implementation of the left-looking **sup-sup** algorithm is currently under investigation

and performance results will be reported elsewhere.

It should be noted that the basic multifrontal method has at least two advantages over the left-looking methods. First, the multifrontal algorithm has long been recognized as the better candidate for out-of-core implementation: only the stack of update matrices and the current frontal matrix are needed in main memory. Second, its superior data locality is of great value when solving very large problems on machines with virtual memory and paging [23]. The impact of paging on performance is not considered in this report because our main concern is the working-storage requirement and the use of blocking to exploit the first level in the memory hierarchy (i.e., fast memory or cache). The paging issue, however, will be investigated elsewhere.

## Acknowledgments

The authors thank the Minnesota Supercomputer Institute and Cray Research, Inc. for providing computer support. Also, part of the work was performed while the authors were visiting the Institute for Mathematics and its Applications at the University of Minnesota, which is funded principally by the National Science Foundation.

## 7. References

[1] P.R. Amestoy and I.S. Duff. Vectorization of a multiprocessor multifrontal code. *Internat. J. Supercomp. Appl.*, 3:41–59, 1989.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing '90*, pages 1–10. IEEE Press, 1990.

[3] C.C. Ashcraft. A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems. Technical Report ETA-TR-51, Engineering Technology Applications Division, Boeing Computer Services, Seattle, Washington, 1987.

[4] C.C. Ashcraft. The domain/segment partition for the factorization of sparse symmetric positive matrices. Technical Report ECA-TR-148, Engineering Computing and Analysis Division, Boeing Computer Services, Seattle, Washington, 1990.

[5] C.C. Ashcraft. Personal communication, 1991.

[6] C.C. Ashcraft, S. Eisenstat, J. Liu, and A. Sherman. A comparison of three column-based distributed sparse factorization schemes. Technical Report YALEU/DCS/RR-810, Department of Computer Science,Yale University, New Haven, CT, 1990.

[7] C.C. Ashcraft, R.G. Grimes, J.G. Lewis, B.W. Peyton, and H.D. Simon. Progress in sparse matrix methods for large linear systems on vector supercomputers. *Internat. J. Supercomp. Appl*, 1:10–30, 1987.

[8] E.C.H. Chu, A. George, J.W-H. Liu, and E.G-Y. Ng. User's guide for SPARSPAK-A: Waterloo sparse linear equations package. Technical Report CS-84-36, University of Waterloo, Waterloo, Ontario, 1984.

[9] A.K. Dave and I.S. Duff. Sparse matrix calculations on the Cray-2. *Parallel Computing*, 5:55–64, 1987.

[10] J.J. Dongarra and S.C. Eisenstat. Squeezing the most out of an algorithm in Cray Fortran. *ACM Trans. Math. Software*, 10:219–230, 1984.

[11] J.J. Dongarra, F.G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26:91–112, 1984.

[12] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, England, 1987.

[13] I.S. Duff, R.G. Grimes, and J.G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[14] I.S. Duff and J.K. Reid. MA27 - A set of Fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report AERE R 10533, Harwell, 1982.

[15] I.S. Duff and J.K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software*, 9:302–325, 1983.

[16] S.C. Eisenstat, M.C. Gursky, M.H. Schultz, and A.H. Sherman. The Yale sparse matrix package I. the symmetric codes. *Internat. J. Numer. Meth. Engrg.*, 18:1145–1151, 1982.

[17] A. George, M.T. Heath, J.W-H. Liu, and E.G-Y. Ng. Solution of sparse positive definite systems on a shared memory multiprocessor. *Internat. J. Parallel Programming*, 15:309–325, 1986.

[18] A. George, M.T. Heath, J.W-H. Liu, and E.G-Y. Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM J. Sci. Stat. Comput.*, 9:327–340, 1988.

[19] A. George and J.W-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[20] J.W-H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Trans. Math. Software*, 12:127–148, 1986.

[21] J.W-H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Software*, 12:249–264, 1986.

[22] J.W-H. Liu. The multifrontal method and paging in sparse Cholesky factorization. *ACM Trans. Math. Software*, 15:310–325, 1989.

[23] J.W-H. Liu. The multifrontal method and paging in sparse Cholesky factorization. *ACM Trans. Math. Software*, 15:310–325, 1989.

[24] J.W-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. Technical Report CS-90-04, Dept. of Computer Science, York University, North York, Ontario, 1990.

[25] J.W-H. Liu. A generalized envelope method for sparse factorization by rows. *ACM Trans. Math. Software*, 17:112–129, 1991.

[26] E.G-Y. Ng and B. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. Technical Report ORNL/TM-11814, Oak Ridge National Laboratory, Oak Ridge, TN, 1991. (Submitted to SIAM J. Sci. Stat. Comput.).

[27] E. Rothberg and A. Gupta. Fast sparse matrix factorization on modern workstations. Technical Report STAN-CS-89-1286, Stanford University, Stanford, California, 1989.

[28] E. Rothberg and A. Gupta. A comparative evaluation of nodal and supernodal parallel sparse matrix factorization: Detailed simulation results. Technical Report STAN-CS-90-1305, Stanford University, Stanford, California, 1990.

[29] E. Rothberg and A. Gupta. An evaluation of left-looking, right-looking, and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines. Technical Report STAN-CS-91-1377, Stanford University, Stanford, California, 1991.

[30] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Trans. Math. Software*, 8:256–276, 1982.

[31] A.H. Sherman. On the efficient solution of sparse systems of linear and nonlinear equations. Technical Report 46, Dept. of Computer Science, Yale University, 1975.

[32] C. Yang. A vector/parallel implementation of the multifrontal method for sparse symmetric definite linear systems on the Cray Y-MP. Cray Research Inc., Mendota Heights, MN, 1990.

[33] E. Zmijewski. Limiting communication in parallel sparse Cholesky factorization. Technical Report TRCS89-18, Department of Computer Science, University of California, Santa Barbara, California, 1989.

ORNL/TM-11960

## INTERNAL DISTRIBUTION

1. B.R. Appleton
2–3. T.S. Darland
4. E.F. D'Azevedo
5. J. Donato
6. J.J. Dongarra
7. G.A. Geist
8. M.R. Leuze
9–13. E.G. Ng
14. C.E. Oliver
15–19. B.W. Peyton
20–24. S.A. Raby
25. C.H. Romine
26. T.H. Rowan
27–31. R.F. Sincovec
32–36. R.C. Ward
37. P.H. Worley
38. Central Research Library
39. ORNL Patent Office
40. K-25 Appl Tech Library
41. Y-12 Technical Library
42. Lab Records Dept - RC
43–44. Laboratory Records Dept

## EXTERNAL DISTRIBUTION

45. Cleve Ashcraft, Boeing Computer Services, P.O. Box 24346, M/S 7L-21, Seattle, WA 98124-0346

46. Donald M. Austin, 6196 EECS Bldg., University of Minnesota, 200 Union St., S.E., Minneapolis, MN 55455

47. Robert G. Babb, Oregon Graduate Institute, CSE Department, 19600 N.W. von Neumann Drive, Beaverton, OR 97006-1999

48. Lawrence J. Baker, Exxon Production Research Company, P.O. Box 2189, Houston, TX 77252-2189

49. Jesse L. Barlow, Department of Computer Science, Pennsylvania State University, University Park, PA 16802

50. Edward H. Barsis, Computer Science and Mathematics, P.O. Box 5800, Sandia National Laboratories, Albuquerque, NM 87185

51. Chris Bischof, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

52. Ake Bjorck, Department of Mathematics, Linkoping University, S-581 83 Linkoping, Sweden

53. Jean R. S. Blair, Department of Computer Science, Ayres Hall, University of Tennessee, Knoxville, TN 37996-1301

54. Roger W. Brockett, Wang Professor of Electrical Engineering and Computer Science, Division of Applied Sciences, Harvard University, Cambridge, MA 02138

55. James C. Browne, Department of Computer Science, University of Texas, Austin, TX 78712

56. Bill L. Buzbee, Scientific Computing Division, National Center for Atmospheric Research, P.O. Box 3000, Boulder, CO 80307

57. Donald A. Calahan, Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109

58. John Cavallini, Acting Director, Scientific Computing Staff, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585

59. Ian Cavers, Department of Computer Science, University of British Columbia, Vancouver, British Columbia V6T 1W5, Canada

60. Tony Chan, Department of Mathematics, University of California, Los Angeles, 405 Hilgard Avenue, Los Angeles, CA 90024

61. Jagdish Chandra, Army Research Office, P.O. Box 12211, Research Triangle Park, NC 27709

62. Eleanor Chu, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

63. Melvyn Ciment, National Science Foundation, 1800 G Street N.W., Washington, DC 20550

64. Tom Coleman, Department of Computer Science, Cornell University, Ithaca, NY 14853

65. Paul Concus, Mathematics and Computing, Lawrence Berkeley Laboratory, Berkeley, CA 94720

66. Andy Conn, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

67. John M. Conroy, Supercomputer Research Center, 17100 Science Drive, Bowie, MD 20715-4300

68. Jane K. Cullum, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

69. George Cybenko, Center for Supercomputing Research and Development, University of Illinois, 104 S. Wright Street, Urbana, IL 61801-2932

70. George J. Davis, Department of Mathematics, Georgia State University, Atlanta, GA 30303

71. Tim A. Davis, Computer and Information Sciences Department, 301 CSE, University of Florida, Gainesville, Florida 32611-2024

72. John J. Dorning, Department of Nuclear Engineering Physics, Thornton Hall, McCormick Road, University of Virginia, Charlottesville, VA 22901

73. Iain Duff, Numerical Analysis Group, Central Computing Department, Atlas Centre, Rutherford Appleton Laboratory, Didcot, Oxon OX11 0QX, England

74. Patricia Eberlein, Department of Computer Science, SUNY at Buffalo, Buffalo, NY 14260

75. Stanley Eisenstat, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520

76. Lars Elden, Department of Mathematics, Linkoping University, 581 83 Linkoping, Sweden

77. Howard C. Elman, Computer Science Department, University of Maryland, College Park, MD 20742

78. Albert M. Erisman, Boeing Computer Services, P.O. Box 24346, M/S 7L-21, Seattle, WA 98124-0346

79. Geoffrey C. Fox, Northeast Parallel Architectures Center, 111 College Place, Syracuse Universitiy, Syracuse, NY 13244-4100

80. Paul O. Frederickson, NASA Ames Research Center, RIACS, M/S T045-1, Moffett Field, CA 94035

81. Fred N. Fritsch, L-300, Mathematics and Statistics Division, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550

82. Robert E. Funderlic, Department of Computer Science, North Carolina State University, Raleigh, NC 27650

83. K. Gallivan, Computer Science Department, University of Illinois, Urbana, IL 61801

84. Dennis B. Gannon, Computer Science Department, Indiana University, Bloomington, IN 47405

85. Feng Gao, Department of Computer Science, University of British Columbia, Vancouver, British Columbia V6T 1W5, Canada

86. David M. Gay, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974

87. C. William Gear, Computer Science Department, University of Illinois, Urbana, IL 61801

88. W. Morven Gentleman, Division of Electrical Engineering, National Research Council, Building M-50, Room 344, Montreal Road, Ottawa, Ontario, Canada K1A 0R8

89. J. Alan George, Vice President, Academic and Provost, Needles Hall, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

90. John R. Gilbert, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto CA 94304

91. Gene H. Golub, Department of Computer Science, Stanford University, Stanford, CA 94305

92. Joseph F. Grcar, Division 8331, Sandia National Laboratories, Livermore, CA 94550

93. John Gustafson, Ames Laboratory, Iowa State University, Ames, IA 50011

94. Per Christian Hansen, UCI*C Lyngby, Building 305, Technical University of Denmark, DK-2800 Lyngby, Denmark

95. Richard Hanson, IMSL Inc., 2500 Park West Tower One, 2500 City West Blvd., Houston, TX 77042-3020

96. Michael T. Heath, National Center for Supercomputing Applications, 4157 Beckman Institute, University of Illinois, 405 North Mathews Avenue, Urbana, IL 61801-2300

97. Don E. Heller, Physics and Computer Science Department, Shell Development Co., P.O. Box 481, Houston, TX 77001

98. Nicholas J. Higham, Department of Mathematics, University of Manchester, Grt Manchester, M13 9PL, England

99. Charles J. Holland, Air Force Office of Scientific Research, Building 410, Bolling Air Force Base, Washington, DC 20332

100. Robert E. Huddleston, Computation Department, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550

101. Ilse Ipsen, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520

102. Barry Joe, Department of Computer Science, University of Alberta, Edmonton, Alberta T6G 2H1, Canada

103. Lennart Johnsson, Thinking Machines Inc., 245 First Street, Cambridge, MA 02142-1214

104. Harry Jordan, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309

105. Bo Kagstrom, Institute of Information Processing, University of Umea, 5-901 87 Umea, Sweden

106. Malvyn H. Kalos, Cornell Theory Center, Engineering and Theory Center Bldg., Cornell University, Ithaca, NY 14853-3901

107. Hans Kaper, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Bldg. 221, Argonne, IL 60439

108. Linda Kaufman, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974

109. Robert J. Kee, Applied Mathematics Division 8331, Sandia National Laboratories, Livermore, CA 94550

110. Kenneth Kennedy, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77001

111. Thomas Kitchens, Department of Energy, Scientific Computing Staff, Office of Energy Research, ER-7, Office G-236 Germantown, Washington, DC 20585

112. Richard Lau, Office of Naval Research, 1030 E. Green Street, Pasadena, CA 91101

113. Alan J. Laub, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106

114. Robert L. Launer, Army Research Office, P.O. Box 12211, Research Triangle Park, NC 27709

115. Charles Lawson, MS 301-490, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109

116. Peter D. Lax, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012

117. James E. Leiss, Rt. 2, Box 142C, Broadway, VA 22815

118. John G. Lewis, Boeing Computer Services, P.O. Box 24346, M/S 7L-21, Seattle, WA 98124-0346

119. Jing Li, IMSL Inc., 2500 Park West Tower One, 2500 City West Blvd., Houston, TX 77042-3020

120. Heather M. Liddell, Center for Parallel Computing, Department of Computer Science and Statistics, Queen Mary College, University of London, Mile End Road, London E1 4NS, England

121. Arno Liegmann, c/o ETH Rechenzentrum, Clausiusstr. 55, CH-8092 Zurich, Switzerland

122. Joseph Liu, Department of Computer Science, York University, 4700 Keele Street, North York, Ontario, Canada M3J 1P3

123. Robert F. Lucas, Supercomputer Research Center, 17100 Science Drive, Bowie, MD 20715-4300

124. Franklin Luk, Electrical Engineering Department, Cornell University, Ithaca, NY 14853

125. Thomas A. Manteuffel, Department of Mathematics, University of Colorado - Denver, Campus Box 170, P.O. Box 173364, Denver, CO 80217-3364

126. Consuelo Maulino, Universidad Central de Venezuela, Escuela de Computacion, Facultad de Ciencias, Apartado 47002, Caracas 1041-A, Venezuela

127. James McGraw, Lawrence Livermore National Laboratory, L-306, P.O. Box 808, Livermore, CA 94550

128. Paul C. Messina, Mail Code 158-79, California Institute of Technology, 1201 E. California Blvd., Pasadena, CA 91125

129. Cleve Moler, The Mathworks, 325 Linfield Place, Menlo Park, CA 94025

130. Neville Moray, Department of Mechanical and Industrial Engineering, University of Illinois, 1206 West Green Street, Urbana, IL 61801

131. Brent Morris, National Security Agency, Ft. George G. Meade, MD 20755

132. Dianne P. O'Leary, Computer Science Department, University of Maryland, College Park, MD 20742

133. James M. Ortega, Department of Applied Mathematics, Thornton Hall, University of Virginia, Charlottesville, VA 22901

134. Chris Paige, McGill University, School of Computer Science, McConnell Engineering Building, 3480 University Street, Montreal, Quebec, Canada H3A 2A7

135. Roy P. Pargas, Department of Computer Science, Clemson University, Clemson, SC 29634-1906

136. Beresford N. Parlett, Department of Mathematics, University of California, Berkeley, CA 94720

137. Merrell Patrick, Department of Computer Science, Duke University, Durham, NC 27706

138. Robert J. Plemmons, Departments of Mathematics and Computer Science, Box 7311, Wake Forest University Winston-Salem, NC 27109

139. Jesse Poore, Department of Computer Science, Ayres Hall, University of Tennessee, Knoxville, TN 37996-1301

140. Alex Pothen, Department of Computer Science, Pennsylvania State University, University Park, PA 16802

141. Yuanchang Qi, IBM European Petroleum Application Center, P.O. Box 585, N-4040 Hafrsfjord, Norway

142. Giuseppe Radicati, IBM European Center for Scientific and Engineering Computing, via del Giorgione 159, I-00147 Roma, Italy

143. John K. Reid, Numerical Analysis Group, Central Computing Department, Atlas Centre, Rutherford Appleton Laboratory, Didcot, Oxon OX11 0QX, England

144. Werner C. Rheinboldt, Department of Mathematics and Statistics, University of Pittsburgh, Pittsburgh, PA 15260

145. John R. Rice, Computer Science Department, Purdue University, West Lafayette, IN 47907

146. Garry Rodrigue, Numerical Mathematics Group, Lawrence Livermore Laboratory, Livermore, CA 94550

147. Donald J. Rose, Department of Computer Science, Duke University, Durham, NC 27706

148. Edward Rothberg, Department of Computer Science, Stanford University, Stanford, CA 94305

149. Axel Ruhe, Dept. of Computer Science, Chalmers University of Technology, S-41296 Goteborg, Sweden

150. Joel Saltz, ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23665

151. Ahmed H. Sameh, Center for Supercomputing R&D, 1384 W. Springfield Avenue, University of Illinois, Urbana, IL 61801

152. Michael Saunders, Systems Optimization Laboratory, Operations Research Department, Stanford University, Stanford, CA 94305

153. Robert Schreiber, RIACS, Mail Stop 230-5, NASA Ames Research Center, Moffet Field, CA 94035

154. Martin H. Schultz, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520

155. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006

156. Lawrence F. Shampine, Mathematics Department, Southern Methodist University, Dallas, TX 75275

157. Andy Sherman, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520

158. Kermit Sigmon, Department of Mathematics, University of Florida, Gainesville, FL 32611

159. Horst Simon, Mail Stop T045-1, NASA Ames Research Center, Moffett Field, CA 94035

160. Anthony Skjellum, Lawrence Livermore National Laboratory, 7000 East Ave., L-316, P.O. Box 808 Livermore, CA 94551

161. Danny C. Sorensen, Department of Mathematical Sciences, Rice University, P.O. Box 1892, Houston, TX 77251

162. G. W. Stewart, Computer Science Department, University of Maryland, College Park, MD 20742

163. Paul N. Swartztrauber, National Center for Atmospheric Research, P.O. Box 3000, Boulder, CO 80307

164. Philippe Toint, Dept. of Mathematics, University of Namur, FUNOP, 61 rue de Bruxelles, B-Namur, Belgium

165. Bernard Tourancheau, LIP, ENS-Lyon, 69364 Lyon cedex 07, France

166. Hank Van der Vorst, Dept. of Techn. Mathematics and Computer Science, Delft University of Technology, P.O. Box 356, NL-2600 AJ Delft, The Netherlands

167. Charles Van Loan, Department of Computer Science, Cornell University, Ithaca, NY 14853

168. Jim M. Varah, Centre for Integrated Computer Systems Research, University of British Columbia, Office 2053-2324 Main Mall, Vancouver, British Columbia V6T 1W5, Canada

169. Udaya B. Vemulapati, Dept. of Computer Science, University of Central Florida, Orlando, FL 32816-0362

170. Robert G. Voigt, ICASE, MS 132-C, NASA Langley Research Center, Hampton, VA 23665

171. Phuong Vu, Cray Research, Inc., 655F Lone Oak Drive, Eagan, MN 55121

172. Daniel D. Warner, Department of Mathematical Sciences, O-104 Martin Hall, Clemson University, Clemson, SC 29631

173. Mary F. Wheeler, Rice University, Department of Mathematical Sciences, P.O. Box 1892, Houston, TX 77251

174. Andrew B. White, Computing Division, Los Alamos National Laboratory, P.O. Box 1663, MS-265, Los Alamos, NM 87545

175. Margaret Wright, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974

176. David Young, University of Texas, Center for Numerical Analysis, RLM 13.150, Austin, TX 78731

177. Earl Zmijewski, Department of Computer Science, University of California, Santa Barbara, CA 93106

178. Office of Assistant Manager for Energy Research and Development, U.S. Department of Energy, Oak Ridge Operations Office, P.O. Box 2001 Oak Ridge, TN 37831-8600

179–188. Office of Scientific & Technical Information, P.O. Box 62, Oak Ridge, TN 37831

# END

DATE
FILMED
01/22/92