ORNL/TM-11814

OAK RIDGE NATIONAL LABORATORY

MARTIN MARIETTA

# A Supernodal Cholesky Factorization Algorithm for Shared-Memory Multiprocessors

E. G. Ng
B. W. Peyton



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

---

ORNL/TM--11814

DE91 011255

Engineering Physics and Mathematics Division

Mathematical Sciences Section

# A SUPERNODAL CHOLESKY FACTORIZATION ALGORITHM FOR SHARED-MEMORY MULTIPROCESSORS

Esmond G. Ng
Barry W. Peyton

Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2009, Bldg. 9207-A
Oak Ridge, TN 37831-8083

Date Published: April 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

# A SUPERNODAL CHOLESKY FACTORIZATION ALGORITHM FOR SHARED-MEMORY MULTIPROCESSORS

Esmond G. Ng
Barry W. Peyton

**Abstract**

This paper presents a new left-looking parallel sparse Cholesky factorization algorithm for shared-memory MIMD multiprocessors. The algorithm is particularly well-suited for vector supercomputers with multiple processors, such as the Cray Y-MP. The new algorithm uses supernodes in the Cholesky factor to improve performance by reducing indirect addressing and memory traffic. Earlier factorization algorithms have also used supernodes in this manner. The new algorithm, however, also uses supernodes to reduce the number of system synchronization calls, often by an order of magnitude or more in practice. Experimental results on a Sequent Balance 8000 and a Cray Y-MP demonstrate the effectiveness of the new algorithm. On eight processors of a Cray Y-MP, the new routine performs the factorization at rates exceeding one Gflop for several test problems from the Harwell Boeing test collection, none of which are exceedingly large by current standards.

## 1. Introduction

Large sparse symmetric positive definite systems arise frequently in many scientific and engineering applications. One way to solve such a system is to use Cholesky factorization. Let $A$ be a symmetric positive definite matrix. The Cholesky factor of $A$, denoted by $L$, is a lower triangular matrix with positive diagonal such that $A = LL^T$. When $A$ is sparse, fill occurs during the factorization; that is, some of the zero elements in $A$ will become nonzero elements in $L$. In order to reduce time and storage requirements, only the nonzero positions of $L$ are stored and operated on during *sparse* Cholesky factorization. Techniques for accomplishing this task and for reducing fill have been studied extensively (see [16] 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 $L$, have been performed. Details on the preprocessing can be found in [16].

In recent years, because of advances in computer architectures, there has been much interest in the solution of large sparse linear systems on high performance computers. In particular, there have been investigations into the solution of such problems on computers with multiple processors [18]. Basically, multiprocessor systems can be classified by how their memory is organized. In a shared-memory multiprocessor system, every processor has direct access to a globally shared memory. In this case, the processors can read from or write into the same memory location simultaneously. Of course, for data integrity, writing into the same memory location at any time by more than one processor must be synchronized. Examples of shared-memory multiprocessor systems include the Cray Y-MP, Encore Multimax, Sequent Balance, and Sequent Symmetry. Another way of organizing the memory in a multiprocessor system is to give each processor its own memory to which the owner alone has direct access. For one processor to access data in another processor's memory, the two processors must communicate with each other, for example, by message passing. Examples of distributed-memory multiprocessor systems include the NCUBE 3200 and 6400, and the Intel iPSC/2 and iPSC/i860. It should be noted that there are also hybrid multiprocessor systems in which both local and shared memory are available, such as the BBN Butterfly.

In this paper, we are concerned with the factorization of a sparse symmetric positive definite matrix $A$ on a shared-memory multiprocessor system. This paper can be regarded as a sequel to [15], in which a parallel implementation of a sequential algorithm from [16] was described. We will show however that the number of synchronization operations (i.e., locking and unlocking operations) required by the parallel algorithm in [15] is relatively high; it is proportional to the number of nonzeros in the Cholesky factor $L$. The object of our paper is to describe a new version of the algorithm that reduces the amount of synchronization overhead by exploiting the *supernodal* structure found in the sparsity pattern of $L$. (The notion of supernodes will be introduced

in Section 3.) The role of supernodes in improving both left- and right-looking sparse Cholesky factorization algorithms is well documented [1,3,5,12,25,28]. The new parallel algorithm uses supernodes to reduce memory traffic and indirect indexing operations as previous algorithms have done, which is particularly important on vector supercomputers [1,3,5]. The primary contribution of the paper is the way supernodes are used to improve the parallel efficiency of a left-looking algorithm.

An outline of the paper is as follows. Section 2 reviews the sequential and parallel factorization algorithms discussed in [15]. Section 3 describes the notion of supernodes and their usefulness in a sequential sparse Cholesky factorization algorithm. A parallel supernodal Cholesky factorization algorithm will be presented in Section 3 as well. Section 4 provides experimental results on an IBM RS/6000, a Cray Y-MP, and a Sequent Balance 8000. Finally, Section 5 contains a few concluding remarks and discusses possible future work.

## 2. Background material

### 2.1. Notation and terminology

Assume that $A$ is an $n \times n$ symmetric and positive definite matrix, and let $L$ denote the Cholesky factor of $A$. We use $L_{*,j}$ and $L_{i,*}$ to represent respectively the $j$-th column and $i$-th row of $L$. The sparsity structures of column $j$ and row $i$ of $L$ (excluding the diagonal entry) are denoted by $Struct(L_{*,j})$ and $Struct(L_{i,*})$, respectively. That is,

$$\begin{aligned} Struct(L_{*,j}) &:= \{s > j : l_{s,j} \neq 0\}, \\ Struct(L_{i,*}) &:= \{t < i : l_{i,t} \neq 0\}. \end{aligned}$$

Assume that $l_{k,j} \neq 0$ and suppose that $l_{k,j}$ is not the last nonzero in column $j$ of $L$. The function $next(k,j)$ returns the row index of the *first* nonzero beneath $l_{k,j}$ in the column $L_{*,j}$ [15]. If $l_{k,j}$ is the last nonzero in $L_{*,j}$, then we define $next(k,j)$ to be $n+1$.

The two computational tasks occurring at each step in the Cholesky factorization are scaling a vector and subtracting a multiple of a vector from another vector. These two tasks will be denoted by *cdiv* and *cmod*, respectively [14].

```
cdiv(j):
    l_{j,j} ← (a_{j,j})^{1/2}
    for i = j + 1 to n do
        l_{i,j} ← a_{i,j} / l_{j,j}
    end for
```

*cmod*$(j,k)$, $k < j$:
    **for** $i = j$ **to** $n$ **do**
        $a_{i,j} \leftarrow a_{i,j} - l_{j,k}\, l_{i,k}$
    **end for**

Finally, if $M$ is an $m \times n$ matrix, then $|M|$ denotes the number of nonzero elements in $M$.

### 2.2. Sequential sparse Cholesky factorization

We begin our discussion by first reviewing a sequential general sparse Cholesky factorization algorithm, details of which can be found in [16]. The algorithm is column-oriented and is a *left-looking* algorithm. That is, when column $L_{*,j}$ is to be computed, the algorithm modifies column $A_{*,j}$ with multiples of the previous columns of $L$, namely $L_{*,k}$, $1 \leq k \leq j-1$. Of course, sparsity will be exploited when $A$ is sparse. We will assume throughout that the nonzeros of $A$ and $L$ are stored by columns. The sequential factorization algorithm is given in Figure 2.1. This algorithm and its variations are widely used in many sparse matrix packages, such as SPARSPAK [7].

---

**for** $j = 1$ **to** $n$ **do**
    **for** $k \in Struct(L_{j,*})$ **do**
        *cmod*$(j,k)$
    **end for**
    *cdiv*$(j)$
**end for**

Figure 2.1: A sequential sparse Cholesky factorization algorithm.

---

Since the algorithm in Figure 2.1 is column-oriented and the nonzeros of $L$ are stored by columns, its implementation is quite straightforward except for the determination of the structure of row $j$ of $L$ (i.e., $Struct(L_{j,*})$). Instead of computing the structure of every row of $L$ *prior to* the factorization, the factorization algorithm itself can efficiently generate these sets *during* the factorization, as shown in Figure 2.2. For each column $L_{*,j}$, we maintain a set $\mathcal{S}_j$ of column indices, which will contain precisely the column indices belonging to $Struct(L_{j,*})$ when the column $L_{*,j}$ is computed.

After $L_{*,j}$ has been computed, $j$ is inserted into $\mathcal{S}_q$, where $q$ is the row index of the first nonzero beneath the diagonal in column $j$ (i.e., $q = next(j,j)$). When the algorithm is ready to compute $L_{*,q}$, it will examine $\mathcal{S}_q$ to find the columns of $L$ needed to modify $A_{*,q}$. Among those columns it will find $L_{*,j}$, and thus it will perform *cmod*$(q,j)$ as required. It is easy to see that the next column of $A$ that $L_{*,j}$ will modify

```
for j = 1 to n do
    S_j ← ∅
end for

for j = 1 to n do
    for k ∈ S_j do
        cmod(j, k)
        p ← next(j, k)
        if p ≤ n then
            S_p ← S_p ∪ {k}
        end if
    end for
    cdiv(j)
    q ← next(j, j)
    if q ≤ n then
        S_q ← S_q ∪ {j}
    end if
end for
```

Figure 2.2: A sequential sparse Cholesky factorization algorithm, with the generation of row structure.

is given by $p = next(q, j)$. Hence, the algorithm puts $j$ in $\mathcal{S}_p$ for use when it later computes $L_{*,p}$. More informally, immediately after $L_{*,j}$ has been computed it begins "migrating" from one column of $A$ to another as determined by the values of $next(*, j)$ (or equivalently the structure of $L_{*,j}$). The columns visited by $L_{*,j}$ are exactly those that must be modified by $L_{*,j}$. At any point during the factorization, $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset$ for $i \neq j$. Consequently, the sets $\mathcal{S}_j$ $(1 \leq j \leq n)$ can be stored economically as linked lists using a single integer array of length $n$. This is the primary reason for generating the sets $Struct(L_{j,*})$ in this manner.

### 2.3. Sources of parallelism

As indicated in [15], there are two sources of potential parallelism in sparse Cholesky factorization. The first one is in performing *cmod* operations with the same "updating" column. Suppose $Struct(L_{*,j}) = \{i_1, i_2, \ldots, i_\rho\}$, with $j < i_1 < i_2 < \ldots < i_\rho$. When $L_{*,j}$ has been computed, columns $i_1$, $i_2$, ..., $i_\rho$ of $A$ have to be modified by $L_{*,j}$. These *cmod*'s are independent: they can be performed simultaneously or in any order. Thus, if there are enough processors and if the nonzero entries of $L_{*,j}$ are available to these processors, the operations $cmod(i_1, j)$, $cmod(i_2, j)$,..., $cmod(i_\rho, j)$ can be performed concurrently. The independence of *cmod*'s using the same updating column but different target columns has nothing to do with the sparsity of $L$; indeed, they are the primary source of parallelism in a *dense* column-based factorization.

Sparsity in $L$ gives rise to large-grained parallelism that is not available in a dense factorization. Consider columns $L_{*,k}$ and $L_{*,j}$ where $j > k$. We shall say that $L_{*,j}$ *depends on* $L_{*,k}$ if $L_{*,j}$ cannot be completed until after $L_{*,k}$ has been completed. When neither $L_{*,j}$ depends on $L_{*,k}$ nor $L_{*,k}$ depends on $L_{*,j}$, the two columns are said to be *independent* of one another. The column dependencies are very simple when $L$ is dense: since computation of $L_{*,j}$ requires modification of $A_{*,j}$ by a multiple of every column $L_{*,k}$ where $k < j$, $L_{*,j}$ depends on every such column $L_{*,k}$. To identify column dependencies in the sparse case, we introduce *elimination trees*.

Consider the Cholesky factor $L$. For each column $L_{*,j}$ having off-diagonal nonzero elements, we define $parent[j]$ to be the row index of the first off-diagonal nonzero in that column; that is, $parent[j] = next(j, j)$. For convenience, we define $parent[j]$ to be $j$ when column $L_{*,j}$ has no off-diagonal nonzeros. The *elimination forest* of $L$ is a graph $\mathcal{T}$ with $\{1, 2, \ldots, n\}$ as its node set, and an edge connecting $i$ and $j$ if and only if $j = parent[i]$ and $i \neq j$ [21,26]. It is also easy to show that $\mathcal{T}$ is a tree if and only if the matrix $A$ is *irreducible*. Without loss of generality, we will assume from now on that the given matrix $A$ is irreducible, so that $\mathcal{T}$ is indeed an *elimination tree*. We assume familiarity with the standard terminology associated with rooted trees: e.g., root, parent, child, ancestor, and descendant. We use the notation $\mathcal{T}[i]$ to denote the *subtree* rooted at node $i$; that is, $\mathcal{T}[i]$ is a tree consisting of $i$ and all of its descendants in $\mathcal{T}$.

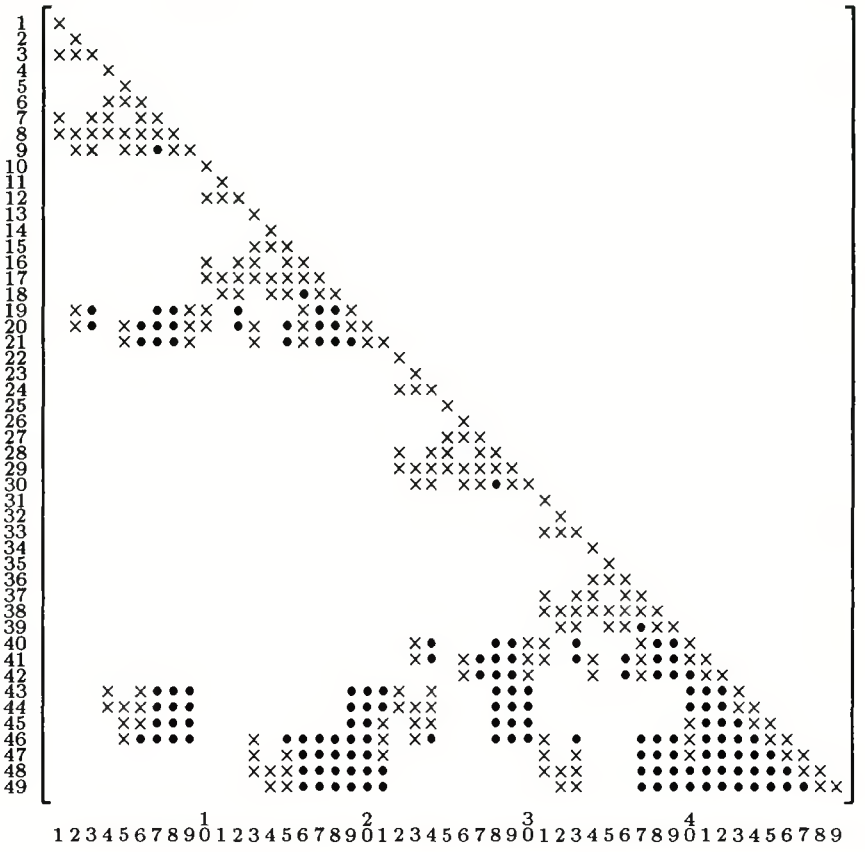

Figure 2.3: A matrix example defined on a $7 \times 7$ nine-point grid ordered by nested dissection. (each $\times$ and $\bullet$ refers to a nonzero in $A$ and a fill entry in $L$, respectively.)

Consider the example in Figure 2.3, which contains the matrix and Cholesky factor associated with a $7 \times 7$ nine-point grid ordered by the nested dissection algorithm [13]. In the figure, each $\times$ is a nonzero entry in the matrix $A$, and each $\bullet$ is a fill entry in the Cholesky factor $L$. The reader may verify that the tree shown in Figure 2.4 is the elimination tree of the matrix $L$ shown in Figure 2.3.

One of the many uses of elimination trees in sparse matrix computation is the analysis of column dependencies in sparse Cholesky factorization. (A survey of elimination trees and their applications in sparse matrix computations is contained in [22].) A key observation [21,26] is that $Struct(L_{j,*}) \subseteq \mathcal{T}[j]$; that is, every $k \in Struct(L_{j,*})$ is a descendant of $j$ in the elimination tree. Of course, column $j$ of $L$ cannot be completed until all columns in $Struct(L_{j,*})$ have been completed. Recursive application of this observation to the descendants of $j$ demonstrates that column $j$ of $L$ cannot be completed until the columns associated with *all* descendants of $j$ (i.e., *all* members of $\mathcal{T}[j] - \{j\}$) have been completed. Moreover, $L_{*,j}$ does not depend on any other columns. Hence, columns $i$ and $j$ are independent if and only if $\mathcal{T}[i]$ and $\mathcal{T}[j]$ are disjoint subtrees. For

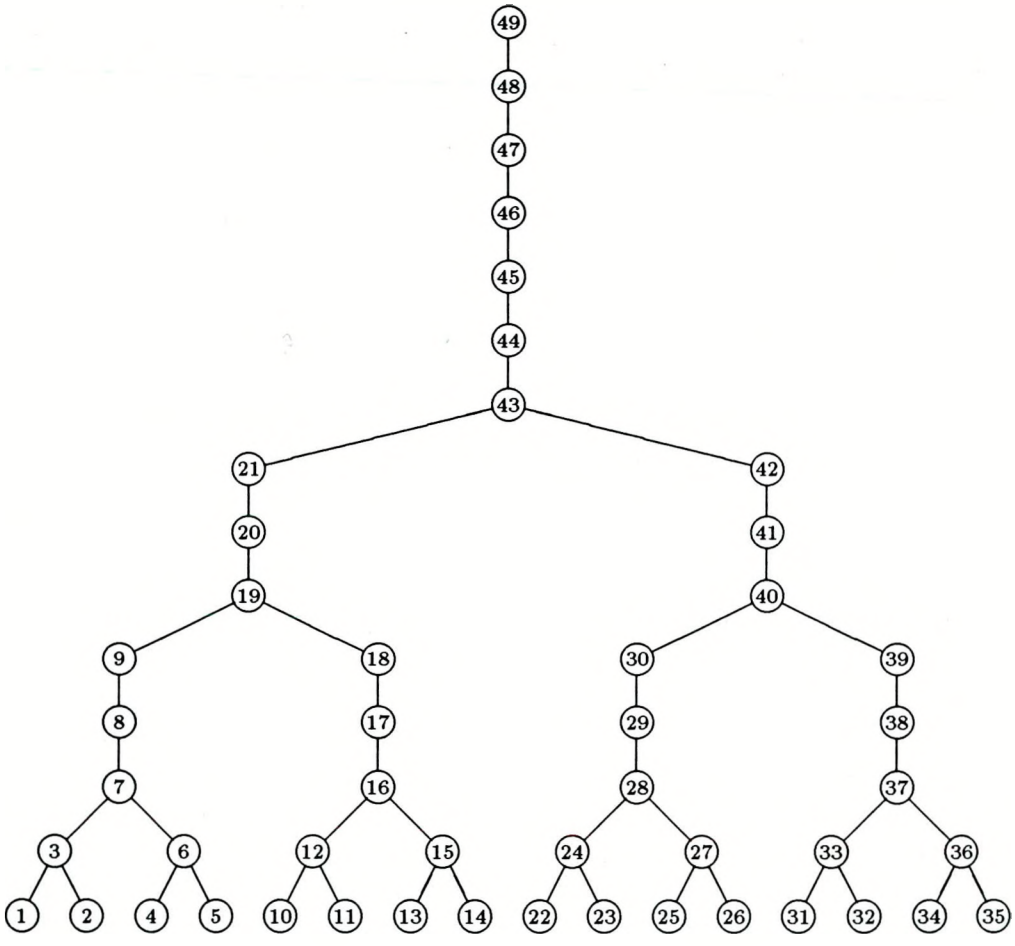


Figure 2.4: Elimination tree for the matrix shown in Figure 2.3.

example, column 41 in Figure 2.4 depends on columns $22 - 40$, and depends on no other columns of the matrix. Columns 41 and 21 are independent because $\mathcal{T}[41]$ and $\mathcal{T}[21]$ are disjoint subtrees.

### 2.4. Parallel sparse Cholesky factorization

We now describe an algorithm for shared-memory multiprocessor systems that exploits these two sources of parallelism. (The algorithm was introduced in [15].) The task of computing column $L_{*,j}$ is referred to as a *column task* in the computation and is denoted by $Tcol(j)$. More precisely,

$$Tcol(j) := \{cmod(j,k) \mid k \in Struct(L_{*,j})\} \cup \{cdiv(j)\}.$$

The parallel algorithm maintains a pool of column tasks, and each processor will be responsible for performing a subset of these column tasks. The assignment of column tasks to processors is dynamic. When a processor is free, it will get a column task from the pool, perform the necessary *cmod* operations, and then carry out the required *cdiv* operation. When the processor has finished a column task, it will get another column task from the pool. Efficient implementation of this dynamic scheduling strategy requires that the pool of tasks be made available to all processors. This is particularly appropriate for shared-memory multiprocessor systems. This approach usually results in good load-balancing, as might be expected.

The parallel algorithm in [15] is presented in Figure 2.5. A few comments on the parallel algorithm are in order. First, note that it is quite similar to the algorithm in Figure 2.2. Second, we assume that the data reside in a globally-shared memory so that every processor can access the entire set of data. Third, since every processor will access the pool of tasks $\mathcal{Q}$, popping a column task from $\mathcal{Q}$ is a critical section and must be performed in a synchronized manner.

Fourth, updating an index set $\mathcal{S}_p$ requires another critical section since $\mathcal{S}_p$ may be simultaneously updated by more than one processor. In Figure 2.5, we have used two primitives, `lock` and `unlock`, to synchronize this operation. The first primitive, `lock`, signals the beginning of a critical section and allows only one processor to proceed. If there is already a processor executing the critical section, a second processor attempting to enter the same section must wait until the first processor has exited the section. The second primitive, `unlock`, signals the end of a critical section, and its execution by one processor permits another processor to enter the critical section. The number of synchronization operations required to maintain the pool of tasks is $O(n)$. It is easy to see that the number of synchronization calls required to update each set $\mathcal{S}_i$ is $O(|L_{i,*}|)$. Thus, the total number of synchronization calls required in the parallel algorithm is $O(|L|)$.

```
Global initialization:
    Q ← {Tcol(1), Tcol(2), ..., Tcol(n)}
    for j = 1 to n do
        S_j ← ∅
    end for

Work performed by each processor:
    while Q ≠ ∅ do
        pop Tcol(j) from Q
        while column j requires further cmod's do
            if S_j = ∅ then
                wait until S_j ≠ ∅
            end if
            lock
            obtain k from S_j
            q ← next(j, k)
            if q ≤ n then
                S_q ← S_q ∪ {k}
            end if
            unlock
            cmod(j, k)
        end while
        cdiv(j)
        q ← next(j, j)
        if q ≤ n then
            lock
            S_q ← S_q ∪ {j}
            unlock
        end if
    end while
```

Figure 2.5: A parallel sparse Cholesky factorization algorithm for shared-memory multiprocessor machines.

## 3. Supernodal Cholesky factorization algorithms

Although the results reported in [15] indicated that the parallel algorithm in Figure 2.5 achieved good speed-up ratios, the algorithms in Figures 2.2 and 2.5 are far from optimal for at least two important reasons. First, both the sequential and parallel algorithms are poor at exploiting some of the hardware features available on advanced computer architectures, in particular, the pipelined arithmetic units on current vector supercomputers. Second, the number of synchronization operations connected with critical sections in the parallel algorithm is relatively high.

In this section, we discuss the notion of *supernodes* in the Cholesky factor of a sparse symmetric positive definite matrix, and show how these supernodes can be used to improve the algorithms in Figures 2.2 and 2.5. In particular, we show how both difficulties with the algorithm in [15] can be dealt with by taking advantage of the supernodal structure.

### 3.1. Notion of supernodes

In the Cholesky factor of a sparse symmetric positive definite matrix, columns with the "same" sparsity structure are often clustered together. Such a grouping of columns is referred to as a *supernode*[1]. We define a supernode of a sparse Cholesky factor $L$ to be a contiguous block of columns in $L$, $\{p, p+1, \ldots, p+q-1\}$, such that

$$Struct(L_{*,p}) = Struct(L_{*,p+q-1}) \cup \{p+1, \ldots, p+q-1\}.$$

It is quite easy to show that for $p \leq i \leq p+q-2$, $Struct(L_{*,i}) = Struct(L_{*,p+q-1}) \cup \{i+1, \ldots, p+q-1\}$. (For details consult [23,24]). Thus, the columns of the supernode $\{p, p+1, \ldots, p+q-1\}$ have a dense diagonal block and have *identical* structure below row $p+q-1$. Figure 3.1 shows a set of supernodes for the matrix of Figure 2.3. The partition of the columns of $L$ into supernodes is often referred to as a *supernode partition*.

Apparently, the term "supernode" first appeared in [5], although the basic idea behind the term was used much earlier. For example, the notion of supernodes has played an important role in improving the efficiency of the minimum degree ordering algorithm [17] and the symbolic factorization process [27]. More recently, supernodes have been used to organize sparse numerical factorization algorithms around matrix-vector or matrix-matrix operations that reduce memory traffic, thereby making more efficient use of vector registers [3,5] or cache [1,25]. They play such a role in both the serial and the new parallel Cholesky factorization algorithms presented in this section.

Note that supernode partitions are not uniquely specified in our definition. Indeed, the choices of a supernode partition depend heavily on the *maximal* sets of contiguous

[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.

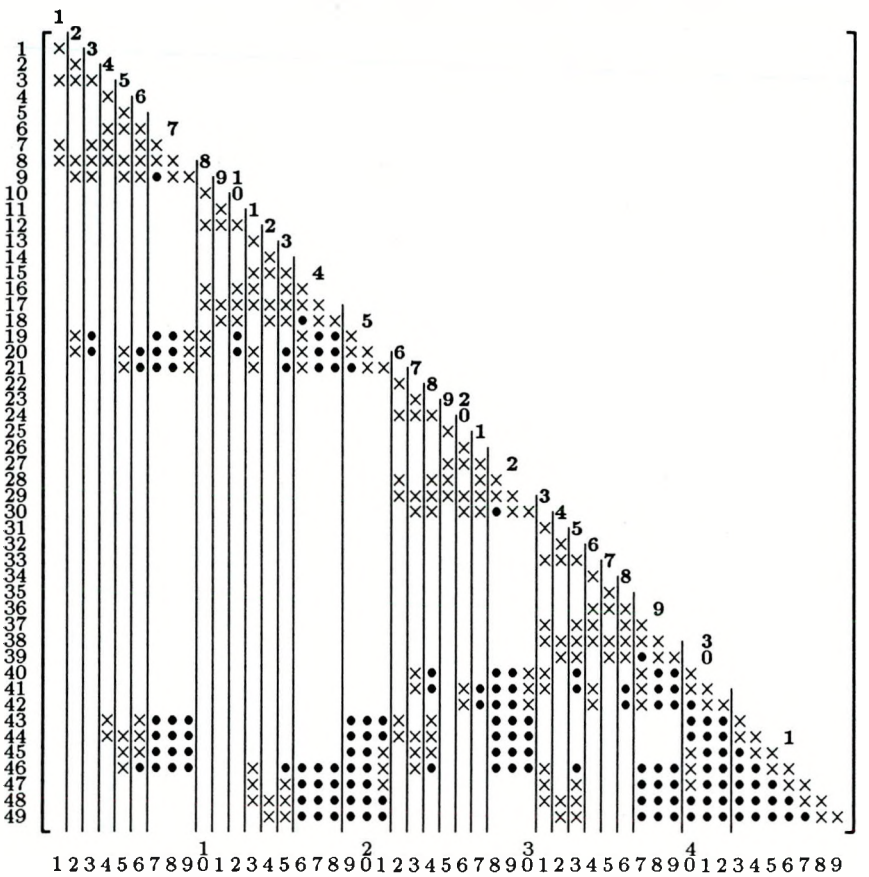

Figure 3.1: *Fundamental supernodes* in the matrix given in Figure 2.3. (Each × and • represents a nonzero in $A$ and a fill in $L$, respectively. Numbers over diagonal entries label supernodes.)

columns that can be supernodes and from which one or more supernodes can be formed. We have used so-called *fundamental* supernodes in our algorithms. The set $\boldsymbol{K} = \{p, p+1, \ldots, p+q-1\}$ is a fundamental supernode if $\boldsymbol{K}$ is a maximal subset of contiguous columns that forms a supernode for which the following holds: for $i = 1, 2, \ldots, q-1$, the node $p+i-1$ is the *sole child* of $p+i$ in the *elimination tree*. The notion of fundamental supernodes was introduced in [4] and was discussed extensively in [23]. The fundamental supernodes for our model problem are shown in Figure 3.1. 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. The supernodal elimination tree for the partition in Figure 3.1 is shown in Figure 3.2, superimposed on the underlying elimination tree.

The primary reason for using the fundamental supernode partition in this application was pointed out in [23]: it is the coarsest supernode partition for which the supernode dependencies can be observed in the supernode elimination tree in a manner strictly analogous to the way the column dependencies are observed in the nodal elimination tree. Consequently, a fundamental supernode partition can be used more cleanly and naturally in a parallel factorization algorithm, where data dependencies are of great practical importance. Liu et al. [23] contains a full discussion of this point.

Given the matrix $A$, the supernode partition can be obtained by several means. When the ordering of the columns and rows of $A$ is a minimum degree or nested dissection ordering, the partition can be obtained easily as a natural by-product of the reordering step. Otherwise, the supernode partition can be obtained directly from the structure of $L$ *after* the symbolic factorization; it can also be obtained *before* the symbolic factorization using the algorithm given in [23].

### 3.2. Sequential supernodal Cholesky factorization

In this section we describe a left-looking sequential sparse Cholesky factorization algorithm that exploits the supernodal structure in $L$. The algorithm is not new; its variants have appeared in [5] and [25]. Let $\boldsymbol{K} = \{p, p+1, \ldots, p+q-1\}$ be a supernode in $L$. Consider the computation of $L_{*,j}$ for some $j > p+q-1$. Suppose column $A_{*,j}$ has to be modified by $L_{*,i}$ where $i \in \boldsymbol{K}$. It follows from the definition of supernodes that column $A_{*,j}$ will be modified by *all* columns of $\boldsymbol{K}$. In other words, a column $j > p+q-1$ is either updated by *no* column of $\boldsymbol{K}$ or *every* column of $\boldsymbol{K}$. This observation has some important ramifications for the performance of sparse Cholesky factorization. Loosely speaking, the columns in a supernode can now be treated as a single unit in the computation. Since the columns in a supernode have the same sparsity structure below the dense diagonal block, modification of a particular column $j > p+q-1$ 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

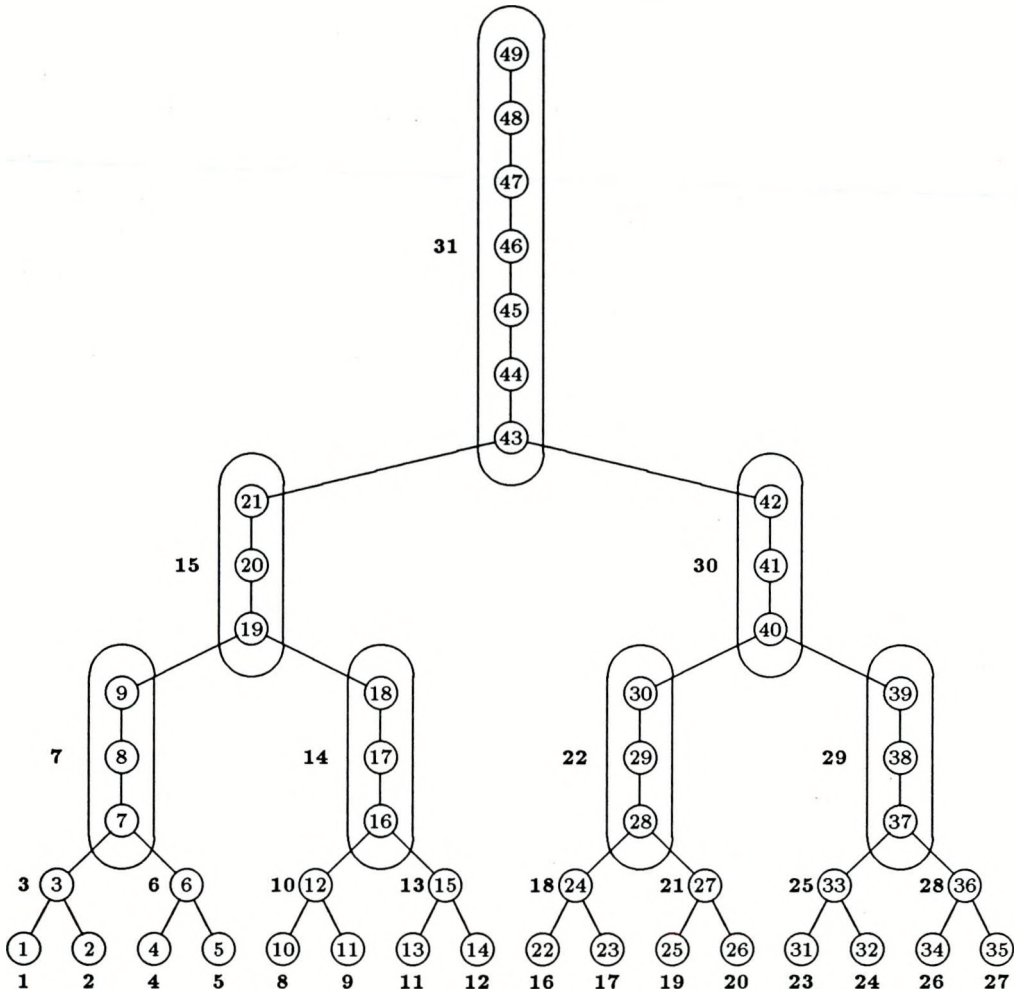


Figure 3.2: Supernodal elimination tree induced by the fundamental supernodes of the matrix shown in Figure 2.3. Ovals enclose supernodes that contain more than one node; nodes not enclosed by ovals are singleton supernodes. Bold-face numbers label supernodes.

accumulation, as described in [9], further reduces memory traffic. These issues have been addressed in detail in [1,3,5,25].

In Figure 3.3, we present a supernodal Cholesky factorization algorithm, which is quite similar to the one in Figure 2.2. In Figure 2.2, $\mathcal{S}_j$ identifies the columns of $L$ needed to modify $A_{*,j}$ when $L_{*,j}$ is computed. Incorporating supernodes into the algorithm, we exploit the fact that columns in the same supernode update the same set of columns outside the supernode. Thus, $\mathcal{S}_j$ will identify the *supernodes* needed to modify $A_{*,j}$ when $L_{*,j}$ is to be computed. In Figure 3.3, we have adopted the following notation. Supernodes are denoted by bold capital letters, and in order to keep the notation simple $\boldsymbol{K}$ is to be interpreted in one of two different senses, depending on the context in which it appears. In one context, $\boldsymbol{K}$ is interpreted as the set of columns composing the supernode, i.e., $\boldsymbol{K} = \{p, p+1, \ldots, p+q-1\}$. In other lines of the algorithm, the supernodes are treated as an ordered set of loop indices $\mathbf{1}$, $\mathbf{2}$, ..., $\boldsymbol{K}$, ..., $\boldsymbol{N}$, where $\boldsymbol{K} < \boldsymbol{J}$ if and only if $p < p'$, where $p$ and $p'$ are the first columns of $\boldsymbol{K}$ and $\boldsymbol{J}$, respectively. This dual-purpose notation is illustrated in Figure 3.1, where the supernode labels are written over the diagonal entries, yet we can still write $\mathbf{30} = \{40, 41, 42\}$, for example. We denote both the last supernode and the number of supernodes by $\boldsymbol{N}$.

Suppose $\boldsymbol{K} = \{p, p+1, \ldots, p+q-1\}$. Whenever $j > p+q-1$ and $l_{j,p+q-1} \neq 0$, the task $cmod(j, \boldsymbol{K})$ consists of the operations $cmod(j,k)$ where $k = p, p+1, \ldots, p+q-1$. If, however, $j \in \boldsymbol{K}$, then $cmod(j, \boldsymbol{K})$ consists of the operations $cmod(j,k)$, for $k = p, p+1, \ldots, j-1$. Suppose $L_{*,\ell}$ is the last column in a supernode $\boldsymbol{K}$ and let $l_{j,\ell} \neq 0$. Then $next(j, \boldsymbol{K})$ is defined to be $next(j,\ell)$. Similarly, we define $next(\boldsymbol{K}, \boldsymbol{K})$ to be $next(\ell,\ell)$.

To reiterate the advantage of exploiting the supernodal structure of $L$, we note that the operation $cmod(j, \boldsymbol{K})$ for $j \notin \boldsymbol{K}$ can be accumulated in work storage by a sequence of dense vector operations (`saxpy` using the BLAS terminology [19]), after which the accumulated column modifications can be applied to the target column $L_{*,j}$ using a single column operation that requires indirect addressing. Execution of the operation $cmod(j, \boldsymbol{J})$ for $j \in \boldsymbol{J}$ is even easier, requiring no work storage or indirect addressing. In both cases, loop unrolling can be employed to reduce memory traffic, thereby improving the utilization of pipelined arithmetic units, especially on vector supercomputers. These capabilities are not available in the "nodal" Cholesky factorization algorithm in Figure 2.2.

### 3.3. Parallel supernodal Cholesky factorization

As far as we know, the first attempt to parallelize a supernodal Cholesky factorization algorithm was described in [28]. Using the notation in Figure 3.3, the basic idea in [28] is to partition the work in $cmod(j, \boldsymbol{K})$ and $cmod(j, \boldsymbol{J})$ evenly among the available processors. This approach is similar to that employed in the LAPACK project [2],

```
for j = 1 to N do
    S_j ← ∅
end for

for J = 1 to N do
    for j ∈ J (in order) do
        for K ∈ S_j do
            cmod(j, K)
            q ← next(j, K)
            if q ≤ n then
                S_q ← S_q ∪ {K}
            end if
        end for
        cmod(j, J)
        cdiv(j)
    end for
    q ← next(J, J)
    if q ≤ n then
        S_q ← S_q ∪ {J}
    end if
end for
```

Figure 3.3: A sequential supernodal Cholesky factorization algorithm.

where, in the interest of software portability and reliability, use of multiple processors occurs *strictly within* each call to some computationally intensive variant of a matrix-matrix multiply (BLAS3) or matrix-vector multiply (BLAS2) kernel subroutine. Hence each call to the kernel involves a fork-and-join operation. For large dense matrices, where the vectors are quite long and each call to the kernel routine typically involves a substantial amount of work, this approach is quite effective [8]. For sparse matrices, however, short vectors and a limited amount of work within a typical call to the kernel routine make it quite difficult to implement this approach in an effective manner. The performance of the code in [28] apparently suffers from these defects, and the stripmining technique used to distribute the tasks $cmod(j, \boldsymbol{K})$ and $cmod(j, \boldsymbol{J})$ among the processors greatly shortens the vector lengths, which is quite detrimental on the target machine, a Cray Y-MP multiprocessor.

In this section, we describe a different way to parallelize the supernodal Cholesky factorization algorithm in Figure 3.3. Our parallel version, shown in Figure 3.4, exploits far more of the potential parallelism than the technique used in [28].

A few comments on the algorithm in Figure 3.4 are in order. First, note that a supernode $\boldsymbol{J}$ is inserted into $\mathcal{S}_q$, where $q = next(\boldsymbol{J}, \boldsymbol{J})$, only after the last column of $\boldsymbol{J}$ has been completed. Thus, when a processor working on column $q$ obtains $\boldsymbol{J}$ from $\mathcal{S}_q$, *all* columns of the supernode $\boldsymbol{J}$ are available for updating column $q$ of $A$. This is potentially inefficient, as the columns of $\boldsymbol{J}$ are not made available to update other columns of $A$ as soon as they have been completed. An alternative to this approach is to insert $\boldsymbol{J}$ into $\mathcal{S}_q$ as soon as the *first* column of the supernode has been completed. Of course, when a processor working on column $q$ obtains $\boldsymbol{J}$ from $\mathcal{S}_q$, not all columns of $\boldsymbol{J}$ are necessarily available. Some flags must be maintained so that the processor can determine which columns of $\boldsymbol{J}$ have been completed. This approach attempts to make every column of $L$ available to update other columns as soon as it has been completed. We have implemented both approaches. Preliminary tests indicate that the difference in performance for these two approaches is extremely small. We have chosen to use the approach shown in Figure 3.4 because it is much simpler to describe and it simplifies our implementation, especially the incorporation of loop unrolling into the code.

Another remark concerns the $cmod(j, \boldsymbol{K})$ operations. As in the sequential case, since all the columns in $\boldsymbol{K}$ share the same sparsity structure below the diagonal entry of the last member of $\boldsymbol{K}$, these operations can be accumulated in work storage using dense vector operations. Extra care is required however when performing the $cmod(j, \boldsymbol{J})$ operation. Since some of the columns $j' < j$ belonging to $\boldsymbol{J}$ may not be completed, a flag has to be associated with each column to record the column's current status. The flag for a particular column is set immediately after the column has been completed. Since the flag for a particular column is set only by the processor that computes the column, synchronization is not needed.

The number of `lock` and `unlock` synchronization operations required in the parallel supernodal Cholesky factorization algorithm is often much smaller than that required

```
Global initialization:
    Q ← {Tcol(1), Tcol(2), ..., Tcol(n)}
    for j = 1 to N do
        S_j ← ∅
    end for

Work performed by each processor:
    while Q ≠ ∅ do
        pop Tcol(j) from Q
        let J be the supernode containing column j
        while column j requires further cmod's do
            if S_j = ∅ then
                wait until S_j ≠ ∅
            end if
            lock
            obtain K from S_j
            q ← next(j, K)
            if q ≤ n then
                S_q ← S_q ∪ {K}
            end if
            unlock
            cmod(j, K)
        end while
        cmod(j, J)
        cdiv(j)
        if j is the last column of supernode J then
            q ← next(J, J)
            if q ≤ n then
                lock
                S_q ← S_q ∪ {J}
                unlock
            end if
        end if
    end while
```

Figure 3.4: A parallel supernodal Cholesky factorization algorithm for shared-memory multiprocessor machines.

in the nodal version, as the following discussion shows. One of the key features in Figures 3.3 and 3.4 is that each set $\mathcal{S}_j$ contains supernodes, rather than columns as it did in Figures 2.2 and 2.5. It is easy to see in Figure 3.3 that each supernode $J$ has to be inserted into $O(|L_{*,\ell}|)$ sets, where $L_{*,\ell}$ is the last column of $J$. Consequently, the total number of synchronizations required in Figure 3.4 depends on the number of *compressed subscripts* [27], which is basically the number of nonzero entries in the last columns of all supernodes. To illustrate the reduction in the amount of synchronization, consider a model $m \times m$ grid problem using either a 5-point or a 9-point operator. Suppose the grid points are labelled using the nested dissection algorithm [13]. It is easy to show that the number of nonzeros in $L$ is $O(m^2 \log m)$ [13] and the number of compressed subscripts is $O(m^2)$ [27]. Thus, the amount of synchronization in the parallel supernodal Cholesky factorization is reduced by a factor of $\log m$. Experimental results in the next section show the reductions in a collection of test problems.

### 3.4. Scheduling column tasks

Our discussion thus far has ignored an important issue: the scheduling of the column tasks $Tcol(j)$ on the available processors. While we have found this issue to be less important on shared-memory machines than it is on distributed-memory machines, it nonetheless deserves some attention, and is likely to be of more consequence as shared-memory machines with substantially more processors become available in the future. Again, the column-dependency information contained in the elimination tree is indispensable in dealing with this problem.

We use the following simple technique to schedule the column tasks. Before the factorization begins, all columns are placed into the column task pool *in the order in which they will be selected from the pool*. Thus, the pool of tasks can be viewed as a static queue. A scheduling, then, is essentially the order in which the column tasks are placed in the static queue.

The goal of the heuristic we use to order the column tasks in the queue is to exploit as much as possible the high-level parallelism available for sets of independent columns. Recall that $Tcol(i)$ and $Tcol(j)$ are independent column tasks if $i$ and $j$ belong to two disjoint subtrees in the elimination tree. Consider the example in Figure 2.4. Columns associated with the leaves of the elimination tree (1, 2, 4, 5, 10, 11, 13, 14, 22, 23, 25, 26, 31, 32, 34 and 35) are independent, since they belong to disjoint subtrees. If there are enough processors available, the corresponding column tasks can be performed in parallel *with no delays due to data-dependencies*. Similarly, columns 9, 18, 30 and 39 are independent. Thus, $Tcol(9)$, $Tcol(18)$, $Tcol(30)$ and $Tcol(39)$ can be carried out concurrently if the column tasks associated with the subtrees $\mathcal{T}[8]$, $\mathcal{T}[17]$, $\mathcal{T}[29]$ and $\mathcal{T}[38]$ have already been completed. The elimination tree therefore provides a natural way to schedule the column tasks. The order in which the column tasks are placed in the work pool $\mathcal{Q}$ is generated by a *breadth-first*, *bottom-up* traversal of a post-ordered

version of the elimination tree. The same strategy was used in [15].

## 4. Numerical experiments

### 4.1. Test problems

| 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 |
| NASASRB | Structure from NASA Langley, shuttle rocket booster |

Table 4.1: List of test problems.

Most of the test problems used in our numerical experiments were taken from the Harwell-Boeing Test Collection [11]. A brief description of the problems is given in Table 4.1. In the experiments, each matrix was initially ordered using an implementation of the minimum degree algorithm due to Liu [20], followed by a postordering of the elimination tree [22]. The reason for postordering the elimination tree is that the algorithms in [23] were used to compute the fundamental supernodes and the symbolic factorization, and they require such a postordering. Some statistics, such as the size of each matrix, nonzero counts for both $A$ and $L$, number of subscripts required to represent the supernodal structure of $L$ (denoted by $\mu(L)$), the number of fundamental supernodes in $L$, and the number of floating-point operations[2] are provided in Table 4.2.

[2] A single floating-point operation is either a floating-point addition or a floating-point multiplication, and is denoted by "flop".

| problem | $n$ | $\|A\|$ | $\|L\|$ | $\mu(L)$ | $\boldsymbol{N}$ | 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 |
| NASASRB | 54,870 | 2,677,324 | 11,904,998 | 592,254 | 8,027 | 4,672,895,526 |

Table 4.2: 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 $L$,

$\boldsymbol{N}$: number of fundamental supernodes in $L$,

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

Throughout this section, we use `colfct` to refer to the column-based approach to Cholesky factorization used in Figures 2.2 and 2.5. Likewise, we use `supfct` to refer to supernode-based approach used in Figures 3.3 and 3.4. For each approach there are two distinct but similar routines: a serial routine and a parallel routine. Thus, there are four routines, each implementing one of the algorithms found in Figures 2.2, 2.5, 3.3, and 3.4. It is worth noting that the serial `colfct` routine is a version of SPARSPAK's `gsfct` routine that has been slightly modified for fair comparison with the other routines, which were written from scratch.

### 4.2. Numerical results on an IBM RS/6000

The primary purpose of Table 4.3 is to show the impact on performance of various levels of loop unrolling in the `supfct` routines. The kernel subroutine called by both the serial and parallel `supfct` routines is capable of unrolling the outer loop of a column-oriented matrix-vector multiply, as in [9]. (The loop unrolling performs multiple `saxpy`'s in a single loop.) Such loop unrolling cannot be introduced into the `colfct` routines because they are not cognizant of the supernode structure on which the technique depends. Loop unrolling levels $\ell = 1, 2, 4$, and 8 have been tried on several machines.

| | | `supfct` with loop unrolling | | | |
|---|---|---|---|---|---|
| problem | `colfct` | $\ell$=1 | $\ell$=2 | $\ell$=4 | $\ell$=8 |
| BCSSTK13 | 9.86 | 7.39 | 5.98 | 5.45 | 5.23 |
| BCSSTK14 | 1.74 | 1.33 | 1.07 | 1.02 | 1.00 |
| BCSSTK18 | 24.02 | 18.59 | 15.17 | 13.93 | 13.60 |
| BCSSTK23 | 19.91 | 14.88 | 11.94 | 10.74 | 10.52 |
| BCSSTK24 | 5.59 | 4.10 | 3.27 | 3.02 | 2.95 |
| NASA1824 | .94 | .74 | .62 | .59 | .58 |
| NASA2910 | 3.68 | 2.83 | 2.32 | 2.13 | 2.13 |
| NASA4704 | 6.09 | 4.53 | 3.71 | 3.42 | 3.37 |

Table 4.3: Factorization times in seconds for various levels of loop-unrolling on an IBM RS/6000 (model 320).

Table 4.3 records the results of these tests on an IBM RS/6000 workstation (model 320). Our double precision Fortran code was compiled using the IBM Fortran compiler `xlf` with optimization turned on (i.e., `xlf -O`). The results for some of the smaller problems in our test set are reported in Table 4.3.

The first thing to note is that `supfct` with no loop unrolling ($\ell$=1) is significantly faster than `colfct`. We believe the improvement is due to better use of the cache by `supfct`, which is due, in turn, to the reduction in indirect addressing and increased locality of the data references obtained via supernodes and careful attention to certain implementation details. On other machines we have tried, `supfct` with no loop unrolling generally runs no faster than `colfct`, and quite often runs slightly slower.

The improvements due to loop unrolling shown in Figure 4.3 are fairly typical of

what we have observed on other machines, too. While the benefits of loop unrolling levels higher than $\ell$=4 are minimal on the IBM RS/6000, the point of diminishing returns is usually higher on other machines. Experience has shown $\ell$=8 to be a good overall choice for the machines we have worked with. In all subsequent experiments, sequential and parallel `supfct` use loop unrolling to level $\ell$=8.

### 4.3. Numerical results on a Sequent Balance 8000

Next, we compare the performance of parallel `colfct` with that of parallel `supfct` on a Sequent Balance 8000, a shared-memory multiprocessor with 12 processors and 16 Mbytes of memory. The parallel routines used Sequent Fortran compiler directives to access the parallel capabilities of the machine and to perform the necessary synchronization operations. The Sequent's Fortran preprocessor transformed these compiler directives into appropriate Fortran code, which, in turn, issued the required system subroutine calls. The Fortran source code was compiled using the Fortran compiler `fortran` with the optimization and preprocessing options turned on (i.e., `fortran -O4 -mp`).

Table 4.4 contains factorization times and speed-up ratios (enclosed in parentheses) for runs on some of the smaller problems in our test set. Since we are interested primarily in comparing the ability of `colfct` and `supfct` to exploit multiple processors, each speed-up ratio is formed by dividing the time required for a parallel run into the time required for a serial run *of the same method*. Note that the serial time is quite distinct from the time required by the parallel algorithm on a single processor, which is always greater.

| | | | parallel | | | | |
|---|---|---|---|---|---|---|---|
| problem | method | serial | p=1 | p=2 | p=4 | p=7 | p=10 |
| BCSSTK13 | colfct | 1147.2 | 1299.4 (0.88) | 652.2 (1.8) | 333.7 (3.4) | 195.6 (5.9) | 140.7 (8.2) |
| | supfct | 874.4 | 878.3 (1.00) | 440.3 (2.0) | 225.9 (3.9) | 132.0 (6.6) | 93.7 (9.3) |
| BCSSTK14 | colfct | 195.2 | 225.6 (0.87) | 114.2 (1.7) | 58.7 (3.3) | 34.8 (5.6) | 25.5 (7.7) |
| | supfct | 155.4 | 157.5 (0.99) | 79.6 (2.0) | 40.4 (3.8) | 23.4 (6.6) | 16.8 (9.3) |
| BCSSTK18 | colfct | 2790.9 | 3152.3 (0.89) | 1589.7 (1.8) | 810.0 (3.4) | 477.3 (5.8) | 346.3 (8.1) |
| | supfct | 2144.4 | 2179.3 (0.98) | 1097.9 (2.0) | 556.7 (3.9) | 326.2 (6.6) | 234.8 (9.1) |
| BCSSTK23 | colfct | 2328.8 | 2627.9 (0.89) | 1322.5 (1.8) | 670.2 (3.5) | 387.0 (6.0) | 275.1 (8.5) |
| | supfct | 1755.1 | 1776.8 (0.99) | 893.1 (2.0) | 447.5 (3.9) | 259.1 (6.8) | 182.6 (9.6) |
| BCSSTK24 | colfct | 640.3 | 733.8 (0.87) | 371.6 (1.7) | 188.8 (3.4) | 110.7 (5.8) | 80.2 (8.0) |
| | supfct | 493.9 | 500.4 (0.99) | 250.6 (2.0) | 126.8 (3.9) | 73.1 (6.8) | 52.1 (9.5) |
| NASA1824 | colfct | 104.9 | 122.4 (0.86) | 62.0 (1.7) | 32.0 (3.3) | 19.4 (5.4) | 14.7 (7.1) |
| | supfct | 84.8 | 86.8 (0.98) | 43.9 (1.9) | 22.4 (3.8) | 13.2 (6.4) | 9.6 (8.8) |
| NASA2910 | colfct | 417.0 | 482.7 (0.86) | 242.5 (1.7) | 124.1 (3.4) | 73.2 (5.7) | 53.3 (7.8) |
| | supfct | 330.0 | 337.0 (0.98) | 167.6 (2.0) | 85.5 (3.9) | 49.5 (6.7) | 35.2 (9.4) |
| NASA4704 | colfct | 691.9 | 791.8 (0.87) | 401.3 (1.7) | 204.2 (3.4) | 119.8 (5.8) | 86.6 (8.0) |
| | supfct | 539.4 | 542.5 (0.99) | 274.6 (2.0) | 138.7 (3.9) | 80.2 (6.7) | 57.0 (9.5) |

Table 4.4: Factorization times in seconds (and speed-ups) on a Sequent Balance 8000.

Comparing the factorization times for the two methods in the last column ($p$=10) clearly indicates the superiority of `supfct` over `colfct`. Indeed the differences in their performance on these problems are large and remarkably consistent, ranging from a

low of 47.5%[3] to a high of 57.8%. Two observations largely account for the superior performance of `supfct`. First, the loop unrolling discussed in the previous section is quite valuable on the Sequent also. The effects of loop unrolling are apparent in the serial runs, and they are quite consistent among the problems, with improvements in performance ranging from a low of 23.7% to a high of 32.7%. The benefits of loop unrolling are largely preserved in the parallel implementation of `supfct`.

Second, `supfct`'s speed-up ratios are consistently better than `colfct`'s; for `colfct` they range from 7.1 to 8.5, and for `supfct` they range from 8.8 to 9.6. The "speed-up" ratios for parallel runs on a single processor ($p$=1) suggest that one of the primary reasons for `colfct`'s inferior speed-up ratios is the high synchronization overhead incurred by the method. Since there is no contention for access to the critical sections of the code when the parallel codes are run on a single processor, it is likely that the relative difference in synchronization overhead costs is significantly greater on 10 processors. The speed-ups for `colfct` on 10 processors are nonetheless quite respectable (7.1–8.5).

### 4.4. Numerical results on a Cray Y-MP

Finally, we compare parallel `colfct` and parallel `supfct` on a Cray Y-MP, a powerful vector supercomputer with 8 processors and 128 Mwords of memory. The code run on this machine was the same code run on the Sequent, with a few minor changes required to take care of machine-dependent constructs for exploiting parallelism. Again, the loop unrolling level used by `supfct` was $\ell$=8, and Fortran compiler directives were used to exercise the machine's parallel capabilities and to perform the necessary synchronization operations. The code was compiled using the Fortran compiler `cf77` with optimization (the default) and preprocessing options on. (i.e., `cf77 -Zu`).

The top half of Table 4.5 reports factorization times and speed-up ratios (enclosed in parentheses) for both methods applied to some small problems in our test set. The bottom half of the table records performance data for `supfct` on the remaining problems in our test set.

Not surprisingly, `supfct` performs much better than `colfct` on this machine. Loop unrolling is more effective on the Cray Y-MP than it is on the Sequent. Comparing the serial runs for the two methods, we find differences in performance ranging from a low of 53% to a high of 132%, due to loop unrolling and reduced indirect indexing in `supfct`. Similar results have been reported previously in [5]. We also find that `supfct` parallelizes much better than `colfct`. For example, on eight processors ($p$=8) the speed-up ratios for `supfct` range from a low of 6.0 to a high of 6.9, which is quite good, especially on such small problems. The speed-up ratios for `colfct`, however, are very poor, ranging from a low of 2.0 to a high of 3.7. As was the case on the Sequent, the "speed-ups" obtained on a single processor indicate that the high synchronization costs

[3]The base for each percentage is the smaller of the two times. This applies to percentages presented later in this section as well.

| problem | method | serial | parallel | | | | |
|---|---|---|---|---|---|---|---|
| | | | p=1 | p=2 | p=4 | p=6 | p=8 |
| BCSSTK13 | colfct | .929 | 1.347 (.69) | .685 (1.4) | .379 (2.5) | .314 (3.0) | .301 (3.1) |
| | supfct | .439 | .493 (.89) | .249 (1.8) | .128 (3.4) | .089 (4.9) | .069 (6.4) |
| BCSSTK14 | colfct | .238 | .391 (.61) | .203 (1.2) | .126 (1.9) | .121 (2.0) | .121 (2.0) |
| | supfct | .156 | .185 (.84) | .093 (1.7) | .048 (3.2) | .033 (4.7) | .026 (6.0) |
| BCSSTK15 | colfct | 2.485 | 3.471 (.72) | 1.770 (1.4) | .962 (2.6) | .768 (3.2) | .728 (3.4) |
| | supfct | 1.071 | 1.172 (.91) | .594 (1.8) | .299 (3.6) | .204 (5.2) | .157 (6.8) |
| BCSSTK16 | colfct | 2.444 | 3.549 (.69) | 1.814 (1.3) | .995 (2.5) | .834 (2.9) | .812 (3.0) |
| | supfct | 1.067 | 1.178 (.91) | .590 (1.8) | .299 (3.6) | .202 (5.3) | .154 (6.9) |
| BCSSTK17 | colfct | 2.712 | 4.121 (.66) | 2.142 (1.3) | 1.223 (2.2) | 1.104 (2.5) | 1.094 (2.5) |
| | supfct | 1.373 | 1.540 (.89) | .773 (1.8) | .392 (3.5) | .267 (5.1) | .206 (6.7) |
| BCSSTK18 | colfct | 2.288 | 3.284 (.70) | 1.675 (1.4) | .928 (2.5) | .767 (3.0) | .729 (3.1) |
| | supfct | 1.314 | 1.481 (.89) | .741 (1.8) | .382 (3.4) | .265 (5.0) | .213 (6.2) |
| BCSSTK23 | colfct | 1.755 | 2.408 (.73) | 1.219 (1.4) | .652 (2.7) | .514 (3.4) | .473 (3.7) |
| | supfct | .798 | .879 (.91) | .441 (1.8) | .224 (3.6) | .153 (5.2) | .125 (6.4) |
| BCSSTK24 | colfct | .674 | 1.071 (.63) | .551 (1.2) | .329 (2.0) | .302 (2.2) | .301 (2.2) |
| | supfct | .338 | .381 (.89) | .190 (1.8) | .096 (3.5) | .065 (5.2) | .050 (6.8) |
| BCSSTK25 | supfct | 2.580 | 2.872 (.90) | 1.433 (1.8) | .731 (3.5) | .504 (5.1) | .394 (6.5) |
| BCSSTK29 | supfct | 2.939 | 3.195 (.92) | 1.602 (1.8) | .810 (3.6) | .547 (5.4) | .421 (7.0) |
| BCSSTK30 | supfct | 5.816 | 6.301 (.92) | 3.154 (1.8) | 1.590 (3.7) | 1.073 (5.4) | .815 (7.1) |
| BCSSTK31 | supfct | 12.607 | 13.299 (.95) | 6.653 (1.9) | 3.330 (3.8) | 2.249 (5.6) | 1.706 (7.4) |
| BCSSTK32 | supfct | 7.773 | 8.491 (.92) | 4.249 (1.8) | 2.134 (3.6) | 1.442 (5.4) | 1.100 (7.1) |
| BCSSTK33 | supfct | 5.831 | 6.146 (.95) | 3.074 (1.9) | 1.539 (3.8) | 1.040 (5.6) | .788 (7.4) |
| NASA1824 | supfct | .114 | .137 (.83) | .069 (1.7) | .036 (3.2) | .025 (4.6) | .019 (6.0) |
| NASA2910 | supfct | .287 | .331 (.87) | .166 (1.7) | .084 (3.4) | .058 (4.9) | .045 (6.4) |
| NASA4704 | supfct | .429 | .483 (.89) | .243 (1.8) | .124 (3.5) | .085 (5.0) | .065 (6.6) |
| NASASRB | supfct | 23.563 | 24.850 (.95) | 12.404 (1.9) | 6.202 (3.8) | 4.179 (5.6) | 3.164 (7.4) |

Table 4.5: Factorization times in seconds (and speed-ups) on a Cray Y-MP.

incurred by `colfct` seriously degrade its parallel performance. Indeed, on this machine the overhead appears to be considerably higher than it was on the Sequent. This high overhead, combined with the fast floating-point computational rates on this machine, probably account for most of the degradation in parallel performance of `colfct`.

The performance of `supfct` on the large problems in the bottom half of the table is consistently good. Ignoring the three smallest problems in this portion of the table (NASA1824, NASA2910, and NASA4704), speed-up ratios range from a low of 6.5 to a high of 7.4. On six out of seven of these problems the speed-up ratio is 7.0 or greater, with the 6.5 speed-up ratio reserved for the problem requiring the least work, namely BCSSTK25.

Table 4.6 compares the performance of `supfct` with the parallel supernodal factorization algorithm used in [28], which we will designate as `supfct-SVY`. The performance figures are expressed in Mflops[4], as is commonly done for vector supercomputers such as the Cray Y-MP. We report the performance of both codes on those problems in our test set for which results for `supfct-SVY` were available to us. The performance data for `supfct-SVY` were obtained from an unpublished manuscript [29].

Consider the results obtained on 8 processors ($p$=8). On six of the twelve prob-

[4]Mflops (megaflops) are millions of floating-point operations per second. Gflops (gigaflops) are billions of floating-point operations per second.

| problem | method | serial | parallel p=4 | parallel p=8 |
|---|---|---|---|---|
| BCSSTK15 | supfct | 154.1 | 551.9 | 1051.2 |
| | supfct-SVY | 197.8 | 301.1 | 320.8 |
| BCSSTK16 | supfct | 139.7 | 498.6 | 968.2 |
| | supfct-SVY | 190.8 | 287.5 | 297.4 |
| BCSSTK23 | supfct | 149.3 | 531.9 | 953.2 |
| | supfct-SVY | 191.6 | 293.3 | 315.1 |
| BCSSTK24 | supfct | 95.9 | 337.8 | 648.5 |
| | supfct-SVY | 139.4 | 168.2 | 168.7 |
| BCSSTK30 | supfct | 159.6 | 583.8 | 1139.0 |
| | supfct-SVY | 212.2 | 350.0 | 375.0 |
| BCSSTK31 | supfct | 202.3 | 766.0 | 1495.3 |
| | supfct-SVY | 251.4 | 566.3 | 689.2 |
| BCSSTK32 | supfct | 142.6 | 519.5 | 1007.9 |
| | supfct-SVY | 193.5 | 291.4 | 307.0 |
| BCSSTK33 | supfct | 206.4 | 782.0 | 1527.3 |
| | supfct-SVY | 258.4 | 593.1 | 717.2 |
| NASA1824 | supfct | 45.3 | 143.3 | 271.5 |
| | supfct-SVY | 64.3 | 69.8 | 69.8 |
| NASA2920 | supfct | 73.4 | 250.8 | 468.1 |
| | supfct-SVY | 97.5 | 121.6 | 121.4 |
| NASA4704 | supfct | 81.6 | 282.3 | 538.4 |
| | supfct-SVY | 117.0 | 143.5 | 143.5 |
| NASASRB | supfct | 198.3 | 753.4 | 1476.9 |
| | supfct-SVY | 250.6 | 531.6 | 625.2 |

Table 4.6: Factorization computational rates (Mflops) on a Cray Y-MP.

lems, `supfct` performs the factorization at over a Gflop, with highs of 1.48 Gflops on NASASRB, 1.50 Gflops on BCSSTK31, and 1.53 Gflops on BCSSTK33. For two others problems, the computational rate is nearly a Gflop: .97 Gflop on BCSSTK16 and .95 Gflop on BCSSTK23. Thus, for 8 out of 12 of the problems, `supfct` computed the factorization at nearly a Gflop or more. Table 4.2 indicates that the remaining four problems (NASA1824, NASA2910, NASA4704, and BCSSTK24) are quite small. Moreover, in Table 4.5 we see that serial `supfct` requires less than half a second to factor any of these matrices.

The performance of `supfct-SVY` is much poorer due to the problems with this approach mentioned earlier in Section 3.3. The code runs at less than a Gflop on every problem, despite having significantly higher serial efficiency due to assembly language programming of the compute-intensive kernel routines and other machine-specific optimizations. Our parallel implementation of `supfct` is a Fortran 77 code, with *no* machine-specific optimizations.

## 5. Concluding remarks

We have implemented a new parallel sparse Cholesky factorization algorithm for shared-memory multiprocessors. This new left-looking algorithm uses techniques from [5] and [15]: it uses supernodes to reduce indirect addressing and memory traffic [5], and it decomposes the computation into column tasks $Tcol(j)$ and schedules these tasks dynamically on the available processors [15]. Incorporation of supernodes into the algorithm in [15] reduces the synchronization overhead required to manage the row structure sets $\mathcal{S}_q$ from $O(|L|)$ to $O(\mu(L))$, where $\mu(L)$ is the number of row subscripts required to represent the supernodal structure of $L$. In practice, $\mu(L)$ is often much smaller than $|L|$; consequently, contention for the critical sections is likely to be much higher in `colfct` than in `supfct`. Since the algorithms use a single lock variable for the critical sections, the sections are executed serially. Thus, the serial component of `supfct` is in practice much smaller than that of `colfct`. Our tests indicate that this is the single most important factor contributing to the new algorithm's superior parallel performance.

Right-looking sparse Cholesky algorithms (i.e., multifrontal algorithms) for shared-memory multiprocessors have appeared in [6,10,29]. These algorithms exploit supernodes in much the same way that the new parallel left-looking algorithm does. Parallelizing the multifrontal algorithm however is considerably more complicated than parallelizing the simpler left-looking algorithms. For instance, parallel multifrontal Cholesky for shared-memory machines can no longer use a simple and efficient stack to manage the "update" matrices required by the method. Methods for dealing with this fragmented component of work storage are necessarily more complicated and storage inefficient [10,29]. On the other hand, `supfct` requires only a modest amount of work storage, which can be determined *before* the numerical factorization begins. Break-

ing multifrontal Cholesky factorization into tasks and scheduling these tasks on the available processors is also more complicated than it is for the left-looking algorithms. The additional complexity has lead to parameterized implementations, where the performance of the code is quite sensitive to parameter selection (see [10] and especially [29]).

There are however significant advantages enjoyed by the multifrontal method; e.g., it is a superior out-of-core method and is better able to improve performance by loading and reusing data in cache. It appears to us that the multifrontal method very likely will always be the method of choice for out-of-core sparse Cholesky factorization. However, we think that a block-to-block left-looking algorithm may be quite competitive with the multifrontal method at exploiting cache to improve performance. Such an algorithm would be built around a $cmod(\boldsymbol{J}, \boldsymbol{K})$ operation that updates the appropriate subset of columns from supernode $\boldsymbol{J}$ with all the columns of the updating supernode $\boldsymbol{K}$. With the rising importance of cache memory on recent supercomputers and workstations, exploring this approach on current serial and parallel machines is a promising area for future work.

## 6. 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. 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. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Software*, 15:291–309, 1989.

[5] 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.

[6] R.E. Benner, G.R. Montry, and G.G. Weigand. Concurrent multifrontal methods: shared memory, cache, and frontwidth issues. *Internat. J. Supercomp. Appl*, 1:26–44, 1987.

[7] 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.

[8] J.J. Dongarra, I.S. Duff, J. Du Croz, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

[9] 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.

[10] I.S. Duff. Multiprocessing a sparse matrix code on the Alliant FX/8. *J. Comput. Appl. Math.*, 27:229–239, 1989.

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

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

[13] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[14] A. George, M.T. Heath, and J. W-H. Liu. Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Alg. Appl.*, 77:165–187, 1986.

[15] 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.

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

[17] A. George and D.R. McIntyre. On the application of the minimum degree algorithm to finite element systems. *SIAM J. Numer. Anal.*, 15:90–111, 1978.

[18] M.T. Heath, E. Ng, and B.W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 1991. (To appear).

[19] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:308–371, 1979.

[20] J. W-H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11:141–153, 1985.

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

[22] J. W-H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11:134–172, 1990.

[23] J.W.H. Liu, E. Ng, and B.W. Peyton. On finding supernodes for sparse matrix computations. Technical Report ORNL/TM-11563, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.

[24] A. Pothen. Simplicial cliques, shortest elimination trees, and supernodes in sparse Cholesky factorization. Technical Report CS-88-13, Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania, 1988.

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

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

[27] A.H. Sherman. *On the efficient solution of sparse systems of linear and nonlinear equations.* PhD thesis, Yale University, 1975.

[28] H.D. Simon, P.A. Vu, and C.W. Yang. Sparse matrix at 1.68 Gflops. Technical report, Boeing Computer Services, Seattle, Washington, 1989.

[29] 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.

ORNL/TM-11814

## INTERNAL DISTRIBUTION

1. B. R. Appleton
2–3. T. S. Darland
4. E. F. D'Azevedo
5. J. J. Dongarra
6. G. A. Geist
7. E. R. Jessup
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 Plant Library
41. Y-12 Technical Library / Document Reference Station
42. Laboratory Records - RC
43–44. Laboratory Records Department

## 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, Department of Physics, Room 229.1, Syracuse University, Syracuse, NY 13244-1130

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, Center for Supercomputing Research and Development, 305 Talbot Laboratory, University of Illinois, 104 South Wright Street, Urbana, IL 61801-2932

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

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

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

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, 13013 Chestnut Oak Drive, Gaithersburg, MD 20878

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

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

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

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

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

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

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

133. Chris Paige, Department of Computer Science, McGill University, 805 Sherbrooke Street W., Montreal, Quebec, Canada H3A 2K6

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

165. 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

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

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

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

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

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

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

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

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

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

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

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

177. 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

178–187. Office of Scientific & Technical Information, P.O. Box 62, Oak Ridge, TN 37831