An Efficient Algorithm to Compute Row and Column Counts for Sparse Cholesky Factorization*

John R. Gilbert† Esmond G. Ng‡ Barry W. Peyton‡

September 2, 1992

Abstract

Let an undirected graph G be given, along with a specified depth-first spanning tree T. We give almost-linear-time algorithms to solve the following two problems: First, for every vertex v, compute the number of descendants w of v for which some descendant of w is adjacent (in G) to v. Second, for every vertex v, compute the number of ancestors of v that are adjacent (in G) to at least one descendant of v.

These problems arise in Cholesky and QR factorizations of sparse matrices. Our algorithms can be used to determine the number of nonzero entries in each row and column of the triangular factor of a matrix from the zero/nonzero structure of the matrix. Such a prediction makes storage allocation for sparse matrix factorizations more efficient. Our algorithms run in time linear in the size of the input times a slowly-growing inverse of Ackermann's function. The best previously known algorithms for these problems ran in time linear in the sum of the nonzero counts, which is usually much larger. We give experimental results demonstrating the practical efficiency of the new algorithms.

*Work was supported in part by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Incorporated, and in part by the Institute for Mathematics and Its Applications with funds provided by the National Science Foundation.

†Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304-1314 (gilbert@parc.xerox.com).

‡Mathematical Sciences Section, Oak Ridge National Laboratory, P. O. Box 2008, Oak Ridge, Tennessee 37831-6367 (esmond@msr.epm.ornl.gov, peyton@msr.epm.ornl.gov).

Keywords: sparse Cholesky factorization, sparse QR factorization, symbolic factorization, graph algorithms, chordal graph completion, disjoint set union, column counts, row counts.

AMS(MOS) subject classifications: 65F50, 68Q20

Computing Reviews descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra — *Linear systems (direct and iterative methods)*, *Sparse and very large systems*; G.2.2 [**Discrete Mathematics**]: Graph Theory — *Graph algorithms*; G.4 [**Mathematics of Computing**]: Mathematical Software — *Algorithm analysis*

1 Introduction

Direct solution of a sparse symmetric positive definite linear system requires four steps [7, 15]: reordering, symbolic factorization, sparse Cholesky factorization, and sparse triangular solutions. Let A be the $n \times n$ coefficient matrix of the linear system after it has been reordered to reduce fill, and let L be the lower triangular Cholesky factor of A. This paper presents improved algorithms for computing the number of nonzero entries in each row and column of L *prior to the symbolic factorization step.* We will refer to these parameters as the *row counts* and *column counts* of L.

In least squares computations, A is $m \times n$, with $m \geq n$. It is often necessary to compute the orthogonal factorization $A = QR$. Our algorithms can be used also to predict the row counts and column counts of the upper triangular factor R, since the structure of R is always contained in the structure of the Cholesky factor of $A^T A$ [12].

Throughout the paper we assume familiarity with graphs, trees, and such basic techniques as depth-first search [26]. We also assume a basic knowledge of the four steps in solving sparse systems by Cholesky factorization, and with the use of graphs in these algorithms [15]. More specifically, we assume familiarity with elimination trees [19], skeleton graphs [18], postorderings, supernodes [1, 2, 16, 20, 22], and the subscript compression scheme for L [15, 27].

1.1 Applications

Here we survey some of the sparse matrix settings in which it is useful to precompute the row counts, the column counts, or the total number of nonzeros in the Cholesky factor of a sparse matrix.

Either the row or column counts can be used to compute $|L|$, the total number of nonzeros in the factor. (We write $|X|$ for the number of nonzeros in a matrix X, or the number of elements in a set X.) Knowing $|L|$ before the numeric factorization step makes it possible to allocate storage all at once, instead of dynamically. In sparse Cholesky factorization, the time required to compute $|L|$ by existing methods is dominated by the time required for numerical factorization; but there are at least two settings in which it is valuable to be able to compute $|L|$ as fast as possible.

First, some methods for large-scale numerical optimization use Cholesky factorization on a Hessian matrix [5, 6]. If the Hessian is indefinite, Cholesky factorization will abort, but the partial factorization contains enough information to help determine a good descent direction containing negative curvature information. In this case, the symbolic factorization time may dominate the time spent on the numeric factorization before it aborts. Thus it may be more efficient to skip the symbolic phase and to build the data structure for L during the numeric factorization. However, for this to be efficient, we still need to find $|L|$ (and perhaps the column counts) before starting the factorization.

Second, much research remains to be done on the issue of how best to reorder the initial matrix to reduce fill (i.e., to reduce $|L|$). It is sometimes useful to compute $|L|$ for many different orderings of the same matrix, both in experiments with reordering algorithms and when trying to optimize an ordering for a specific matrix. Our new algorithms make this much faster.

Besides fill, there are several other measures of the quality of a reordering. Some of them can be computed from the column counts; for example, the total number of arithmetic operations is the sum of the squares of the column counts, and the maximum front size is equal to the largest column count. The smallest maximum front size, over all reorderings of a graph, is one more than the graph's *treewidth* [3]. Thus the fast column count algorithm may also be useful in experimental studies of treewidth.

Two applications related to the supernodal structure of L also require the column counts. Supernodes are clusters of columns with related nonzero patterns, which can be exploited to use fast dense matrix computation kernels in sparse factorization; Section 3 describes them in more detail. First, there is a simple, flexible $O(n)$ scheme for computing supernode partitions [2, 17] that takes the column counts and the elimination tree as input. This algorithm is more versatile and faster than the $O(|A|)$ algorithm of Liu, Ng, and Peyton [20], which takes the original matrix and its elimination tree as input. The latter algorithm computes the so-called fundamental supernode partition. Given a fast algorithm to compute column counts, the more flexible scheme could be used efficiently to compute other kinds of supernode partitions, such as Ashcraft's relaxed supernode partition [2] (which trades extra fill for a simpler sparsity structure that can be used to improve efficiency on vector supercomputers or to reduce synchronization overhead on shared-memory multiprocessors).

The second supernodal application of the column counts is to compute the storage required for indexing information for L, in the usual compressed format generated by the symbolic factorization step [27]. Current software packages [4, 9] do not precompute the space needed for this compressed symbolic factorization, because it is too expensive using the currently known algorithms. The storage required for the other three steps in the solution process is usually computed in advance; we believe that the new algorithms introduced here are efficient enough to be used by a software package to precompute the storage requirement of the symbolic factorization step as well.

Finally, we know of only one application that specifically requires the row counts rather than the column counts. The row counts are the numbers of column modifications (sparse SAXPY's) required to complete each column in sparse Cholesky factorization algorithms. Some parallel implementations [13, 14] need the row counts to tell when all the modifications have arrived for each column.

1.2 Previous work

Like many combinatorial algorithms in sparse matrix factorization, all the efficient algorithms for row and column counts begin by computing the elimination tree of the matrix (defined in the next section). The fastest known elimination tree algorithm is due to Liu [19]. The time complexity for this algorithm is dominated by disjoint set union operations, which take time $O(m\,\alpha(m,n))$, where A is $n \times n$ and has $2m$ off-diagonal nonzeros. Here $\alpha(m,n)$ is a slowly-growing inverse of Ackermann's function defined by Tarjan [29]; for all values of m and n less than the number of elementary particles in the observable universe, $\alpha(m,n) \leq 4$. Thus a function that is $O(m\,\alpha(m,n))$ is often called "almost linear."

The fastest previously known algorithm for computing row and column counts is also due to Liu [19]. It first computes the elimination tree of A and then traverses each "row subtree" of the elimination tree (defined in the next section). The total size of the row subtrees is the number of nonzeros in the factor, so the running time of this step is $O(|L|)$. Unless the factor is extremely sparse, the subtree traversals dominate the time to find the elimination tree. To put this in perspective, suppose A is the matrix of an n-node finite difference mesh ordered by nested dissection. Then m is $O(n)$, and $|L|$ is $O(n \log n)$ in two dimensions or $O(n^{4/3})$ in three dimensions.

The algorithm in this paper also takes A and the elimination tree as input but runs in almost-linear time $O(m\,\alpha(m,n))$; the time complexity for the new algorithm is dominated by disjoint set union operations. Thus it computes the row and column counts in the same asymptotic time needed to find the elimination tree. As we will see in Section 4, this asymptotic efficiency is also reflected in practice.

1.3 Outline of the rest of the paper

Section 2 presents the row and column count algorithm from a graph-theoretic point of view. Here it is convenient to think of the input not as the graph $G(A)$ of a matrix, but as the graph $G(A) \cup T(A)$ that has edges both for the matrix nonzeros and for the elimination tree. The elimination tree is a depth-first spanning tree of this graph (and also of the graph of L); thus for the purpose of the high-level view in Section 2 the input is just an undirected graph with a specified depth-first spanning tree. In this setting, we suspect that our results may be useful in efficient algorithms involving chordal graphs, chordal completion, and treewidth.

In Section 3 we return to the matrix-computation point of view, and discuss details of the implementation in the sparse matrix setting. Two points of practical importance arise here: we modify the algorithm slightly to make only one pass over its input, and we take advantage of supernodal structure to compute only with a subgraph called the *skeleton graph*. We show how to organize the entire computation, including the skeleton graph reduction, within the framework of Liu et al.'s fundamental supernode algorithm [20].

Section 4 contains experimental results. We experiment with both the nodal and supernodal versions of the algorithm, and also with several implementations of the disjoint set union operations (UNION and FIND) that dominate the asymptotic running time. The best version is the supernodal algorithm with path-halving and no union by rank (definitions are in Section 3.3); it performs well enough that we argue it should be a standard part of high-performance sparse factorization codes. Finally, Section 5 contains concluding remarks.

2 The algorithm

2.1 Definitions and problem statement

Let $G = (V, E)$ be a connected undirected graph with n vertices and m edges, and let T be a specific depth-first spanning tree for G. We will call vertices v and w *adjacent* if they are joined by an edge in G; that is, if $(v, w) \in E$. We say that vertex v is an *ancestor* of vertex w if v is on the path in T from w to the root of T. Vertex v is a *descendant* of w if w is an ancestor of v. Note that a vertex is its own ancestor and its own descendant; a *proper* ancestor or descendant is one that is different from the vertex itself. We will write $T[v]$ for the set of descendants of v, and also for the subtree of T (rooted at v) that those vertices induce.

Since T is a depth-first spanning tree, every edge of G (whether or not it is an edge of T) joins an ancestor in T to a descendant in T.

To simplify notation, we will assume that the vertices of G are the integers 1 through n. We will also assume that the vertex numbers are a *postorder* on T, that is, that for every vertex v, the vertices of $T[v]$ are numbered consecutively, with v numbered last. Thus vertex n is the root of T.

The *level* of vertex v, which we write $level(v)$, is its distance in T from the root. The *least common ancestor* of vertices v and w, which we write $lca(v, w)$, is the ancestor of v and w with the smallest postorder number (or the largest level). Both a postorder numbering and the vertex levels for an arbitrary tree can be computed in linear time by depth-first search [28]. Given a set of k pairs $\{v, w\}$ of vertices, the k least common ancestors $lca(v, w)$ can be computed in $O(k\,\alpha(k, n))$ time, where α is the very slowly growing inverse of Ackermann's function mentioned above [30]. We describe these algorithms in more detail in Section 3.

We consider the following two problems.

Problem 1 (row counts). For every node $u \in V$, let $row[u]$ be the set of descendants v of u for which either $v = u$ or there exists an edge (u, w) with $w \in T[v]$. The problem is to compute $rc(u) = |row[u]|$ for every u.

Problem 2 (column counts). For every node $v \in V$, let $col[v]$ be the set of ancestors u of v for which either $u = v$ or there exists an edge (u, w) with $w \in T[v]$. The problem is to compute $cc(v) = |col[v]|$ for every v.

Note that $v \in row[u]$ if and only if $u \in col[v]$, and that u is an element of both $row[u]$ and $col[u]$. For each u, the subgraph of T induced by $row[u]$, denoted by $T_r[u]$ and referred to as the *row subtree* of u, is connected; it is a "pruned subtree" rooted at u. The subgraph of T induced by $col[v]$ may not be connected.

We conclude by briefly describing the relationship between these problems and sparse Cholesky factorization. It may seem a bit confusing that we include the elimination tree edges in the graph G in the graph problem but not in the matrix problem; however, the answer is the same in either case.

Let an $n \times n$ symmetric, positive definite matrix A be given, and let $G(A)$ be its undirected graph (whose vertices are the integers 1 through n). Let $G^+(A)$ be the *filled graph* of $G(A)$ [23] obtained by adding to $G(A)$ edge (v, w) whenever there is a path in $G(A)$ from v to w whose intermediate vertices are all smaller than both v and w. The graph $G^+(A)$ is chordal, and (ignoring numerical cancellation) is the graph of $L + L^T$, where L is the Cholesky factor of A [24].

The *elimination tree* of A, denoted $T(A)$, has vertices 1 through n, and the parent of vertex v is the smallest $w > v$ such that (v, w) is an edge of $G^+(A)$. Liu [19] surveys the uses and properties of this structure. It is a forest with one tree for each connected component of $G(A)$; if A is irreducible then $T(A)$ is a tree. The elimination tree may not be a subgraph of $G(A)$, but it is a subgraph of $G^+(A)$, and in fact it is a depth-first spanning tree of that graph. If A' is a matrix whose graph is $G(A') = G(A) \cup T(A)$, it is straightforward that $G^+(A') = G^+(A)$ and $T(A') = T(A)$.

Now consider problems (1) and (2) above for $G = G(A')$ and $T = T(A')$. It is easy to show [19] that the edges of $G^+(A) = G^+(A')$ are exactly those (u, v) for which $v \neq u$ and $v \in row[u]$ (or $u \in col[v]$). Thus $rc(u)$ is the number of nonzeros in row u of the Cholesky factor L of A, and $cc(v)$ is the number of nonzeros in column v of L.

2.2 Row counts

We will count the vertices in $row[u]$ by counting the edges in the pruned subtree $T_r[u]$ of T that $row[u]$ induces. The following lemma lets us partition those edges into paths.

Lemma 1 *Let $p_1 \leq p_2 \leq \cdots \leq p_k$ be some of the vertices of a rooted tree R (where $<$ is postorder), and suppose all the leaves and the root of R are among the p_i's. Let q_i be the least common ancestor of p_i and p_{i+1}, for $1 \leq i < k$. Then each edge (s, t) of the tree is on the tree path from p_j to q_j for exactly one j.*

Proof: Suppose t is the parent of s in R. The descendants of s include at least one leaf, so they include at least one p_i. Let p_j be the largest p_i among the descendants of s. Then $p_j \leq s < p_{j+1}$. (There must be a p_{j+1}—that is, we cannot have $j = k$—because p_k is the root, which is a proper ancestor of s.) Since s is an ancestor of p_j but not of p_{j+1}, the least common ancestor q_j of p_j and p_{j+1} is a proper ancestor of s, and hence an ancestor of t. Therefore (s, t) is on the path from p_j to q_j.

Now consider an $i \neq j$. If s is not an ancestor of p_i, then (s, t) is not on the path from p_i to its ancestor q_i. If s is an ancestor of p_i, then $p_i \leq s$, and $i \neq j$ implies $p_i \leq p_{i+1} \leq s$. Since postorder assigns consecutive numbers to the vertices in a subtree, this means that s is also an ancestor of p_{i+1}, and

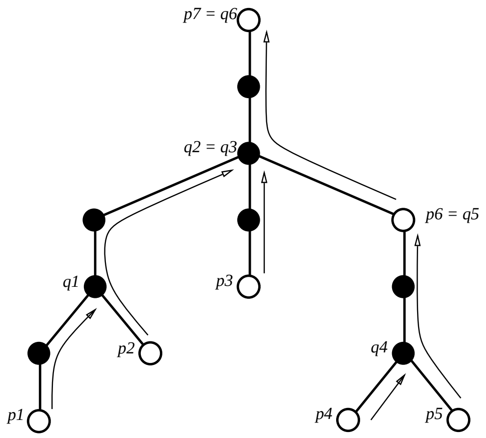

Figure 1: Example of path decomposition.

hence of the least common ancestor q_i. Thus (s,t) is not on the path from p_i to q_i. $\square$

The hypotheses of the lemma allow a vertex to be p_i for more than one i. Thus it is possible that $p_i = p_{i+1} = q_i$ and the path from p_i to q_i is trivial. Figure 1 shows an example of the path decomposition (with no trivial paths).

Recall that T is a depth-first spanning tree of G and hence every edge of G joins an ancestor in T to a descendant in T. Now consider a vertex u of G. If the lower-numbered neighbors of u in G are $p_1 < p_2 < \cdots < p_{k-1}$, and if $p_k = u$, then the pruned subtree $R = T_r[u]$ induced by $row[u]$ satisfies the hypotheses of Lemma 1. Thus the number of edges in $T_r[u]$ is the sum of the lengths of the paths in the lemma. The length of the path from p_i to its ancestor q_i is the difference of their levels. The number of vertices in $row[u]$ is one more than the number of edges, so

$$rc(u) = 1 + \sum_{1 \le i < k} \Big(level(p_i) - level(lca(p_i, p_{i+1})) \Big).$$

(Here lca and $level$ are taken in T rather than $T_r[u]$, but it is clear that for any two vertices in $row[u]$ the least common ancestor and the difference in levels are the same in either tree.)

The algorithm to compute $rc(u)$ for all u first sorts the lower-numbered neighbors of each vertex by postorder, then computes all the necessary least common ancestors, and finally computes the sum above for each u. Computing level numbers (and the postorder itself if necessary) takes linear time, and sorting all the lower-numbered neighbors by postorder takes linear time by a lexicographic bucket sort. There is one least-common-ancestor computation for each edge of G, so the dominant term in the algorithm's time complexity is $O(m\,\alpha(m,n))$.

2.3 Column counts

Because $u \in col[v]$ if and only if $v \in row[u]$, the column count $cc(v)$ is equal to the number of row subtrees $T_r[u]$ that contain v. We could compute $cc(v)$ by traversing each row subtree in turn, and counting the number of times each vertex was traversed [19]. This, however, would take time proportional to $\sum_v cc(v)$.

To get a faster algorithm, we will define weights $wt(v)$ on the vertices of G in such a way that the column count for vertex v will turn out to be the sum of the weights of the descendants of v. The key observation will be that we can compute these weights as a sum of contributions from each row subtree, and that the row subtree contributions can be computed efficiently using the same least common ancestors as in the row count algorithm.

Here are the details. For each vertex u, define χ_u to be the characteristic function of $row[u]$, so that $\chi_u(v) = 1$ if $v \in row[u]$ and $\chi_u(v) = 0$ otherwise. Define wt_u by

$$wt_u(v) = \chi_u(v) - \sum_{\text{children } y \text{ of } v} \chi_u(y). \tag{1}$$

These weights may be positive, negative, or zero. This definition implies that

$$\chi_u(v) = \sum_{x \in T[v]} wt_u(x). \tag{2}$$

In a sense, wt_u is a "first difference" down the tree of the characteristic function of $row[u]$. Finally, define

$$wt(v) = \sum_{u \in V} wt_u(v). \tag{3}$$

Now we prove three lemmas relating the column counts to the weights, the weights to the sets $row[u]$, and finally the $row[u]$, once more, to the least common ancestors.

Lemma 2 *For every vertex v,*

$$cc(v) = \sum_{x \in T[v]} wt(x).$$

Proof: Because $v \in row[u]$ if and only if $u \in col[v]$, we have

$$cc(v) = |col[v]| = \sum_{u \in V} \chi_u(v).$$

Equation 2 says that this is equal to

$$\sum_{u \in V} \sum_{x \in T[v]} wt_u(x).$$

The result follows by reversing the order of summation and using Equation 3. $\Box$

Lemma 2 implies that we can compute the column counts easily and efficiently from the weights, by traversing the tree in postorder and summing the weights of the subtrees. It remains to describe how to compute the weights.

Lemma 3 *Let u and v be vertices. Suppose that d of the children of v are vertices of $row[u]$. Then*

$$wt_u(v) = \begin{cases} 1-d \quad , & \text{if } v \in row[u], \\ -1 \quad , & \text{if } v \text{ is the parent of } u, \\ 0 \quad , & \text{otherwise.} \end{cases}$$

Proof: This is immediate from Equation 1 and the definition of χ_u. $\Box$

Lemma 3 implies that the only vertices v for which $wt_u(v)$ is nonzero are the leaves of the pruned row subtree $T_r[u]$, the internal vertices of $T_r[u]$ that have more than one child in $T_r[u]$, and the parent of u. The following lemma will let us compute $wt_u(v)$ for each v from the same p_i's and q_i's we used in the row count algorithm.

Lemma 4 *Let $p_1 \leq p_2 \leq \cdots \leq p_k$ be some of the vertices of a rooted tree R (where $<$ is postorder), and suppose all the leaves and the root of R are among the p_i's. Let q_i be the least common ancestor of p_i and p_{i+1},* *for* $1 \le i < k$*. Then for each vertex* v *of* R*, the number of children of* v *in* R *is*

$$|\{i : q_i = v\}| - |\{i : p_i = v\}| + 1.$$

Proof: Let $Q = |\{i : q_i = v\}|$, let $P = |\{i : p_i = v\}|$, and let d be the number of children of v in R. Consider the set of directed paths from p_i to q_i in R, for $1 \le i < k$. For any collection of directed paths, each path that includes vertex v either *begins* at v or *enters* v along edges from other vertices. Similarly, each path that includes vertex v either *ends* at v or *leaves* v along edges to other vertices. Consequently:

> The number of paths that either begin at v or enter v along edges from other vertices must be equal to the number of paths that either end at v or leave v along edges to other vertices.

(This is essentially Kirchoff's law for a flow of unit size from p_i to q_i for each i.) Lemma 1 says that every edge of R is on exactly one of these paths. Therefore one path enters v from each of the d children of v; exactly one path leaves v, to its parent, unless v is the root; one path begins at v for each i such that $p_i = v$ (except for $i = k$ if v is the root); and one path ends at v for each i such that $q_i = v$. A trivial path with $p_i = q_i = v$ both starts and ends at v, but does not enter or leave v. Thus the relation above is

$$P + d = Q + 1$$

if v is not the root of R, or

$$(P - 1) + d = Q + 0$$

if v is the root. In either case, we have $d = Q - P + 1$ as desired. □

Now consider a vertex u of G. If the lower-numbered neighbors of u in G are $p_1 < p_2 < \cdots < p_{k-1}$, and if $p_k = u$, then the pruned subtree $R = T_r[u]$ induced by $row[u]$ satisfies the hypotheses of Lemma 4. Therefore, using Lemma 3, if v is a vertex of $row[u]$ then $wt_u(v) = |\{i : p_i = v\}| - |\{i : q_i = v\}|$. Thus we could compute $wt_u(v)$ for all v by initializing each weight to zero, setting the weight of the parent of u to -1, and then adding one to the weight of each p_i and subtracting one from the weight of each q_i.

In fact we do not need to compute $wt_u(v)$ separately for each u; we can compute $wt(v) = \sum_u wt_u(v)$ all at once. The algorithm begins, like the row

```
Sort the vertices and their lists of neighbors by a postorder of T;
Assign u and the lower-numbered neighbors of u to adj[u], for all u;
Compute level(u) as the distance from u to n (the root), for all u;
Compute lca(p, p') for every p and its successor p' in adj[u], for all u;
rc(u) ← 1, for all u;
wt(u) ← 0, for all u;
for u ← 1 to n do
    if u ≠ n then
        wt(parent(u)) ← wt(parent(u)) − 1;
    end if
    for p ∈ adj[u] do
        wt(p) ← wt(p) + 1;
        if p is the last member of adj[u] then
            rc(u) ← rc(u) + level(p) − level(u);
        else
            p' ← the successor of p in adj[u];
            q ← lca(p, p');
            rc(u) ← rc(u) + level(p) − level(q);
            wt(q) ← wt(q) − 1;
        end if
    end for
end for
cc(v) ← wt(v), for all v;
for v ← 1 to n − 1 do
    cc(parent(v)) ← cc(parent(v)) + cc(v);
end for
```

Figure 2: Algorithm to compute row and column counts.

count algorithm, by sorting the lower-numbered neighbors of each vertex into postorder and computing all the necessary least common ancestors. It initializes $wt(v)$ to zero for each v. Then, for each u, it subtracts one from the weight of the parent of u, adds one to the weight of each lower-numbered neighbor of u, and subtracts one from the weight of the least common ancestor of each pair of consecutive (in postorder) lower-numbered neighbors of u. Finally, the algorithm computes $cc(v)$ for all v by summing the weights of each subtree in postorder. Figure 2 sketches the algorithm to compute both row and column counts. The only step that takes more than linear time is the least-common-ancestor computation, and the dominant term in the algorithm's time complexity is $O(m\,\alpha(m,n))$.

3 Implementation

The discussion in the previous section was in a general graph theoretic setting. However, in order to obtain the most efficient implementation of the new algorithm for our applications, we need to switch back to a sparse matrix setting.

Consider a symmetric matrix A and its graph $G(A)$. Assume that the elimination tree $T(A)$, the postordering, and the values $level(u)$ (with respect to $T(A)$) have been computed, as required in Figure 2. Two other requirements must be met to obtain a practical and efficient implementation of the new algorithm.

First, we must reorganize the computation to avoid sorting the adjacency lists by postorder and precomputing all the least common ancestors. Indeed, direct implementation of the algorithm in Figure 2 would require that $G(A)$ be processed three times, and we doubt that any multiple-pass implementation will come close to realizing the practical efficiency of the single-pass implementation presented in this section.

Second, we must discard some edges of $G(A)$ that do not affect the result. Recall from Liu [18] that the *skeleton graph* $G^- = G^-(A)$ is obtained from $G(A)$ by removing every edge (u,v) for which $v < u$ and the vertex v is not a leaf of $T_r[u]$. The skeleton graph is the smallest subgraph of $G(A)$ whose filled graph is identical with that of $G(A)$. Consequently, the new algorithm produces the same results when applied to G^- as when applied to $G(A)$. Indeed, if $G = G^- \cup T(A)$ rather than $G = G(A) \cup T(A)$ in Lemmas 1 and 4, then every vertex $p_1, p_2, \ldots, p_{k-1}$ is a leaf in the tree R, which reduces the number of edges searched and least common ancestors computed by the new algorithm to the minimum possible. Since G^- often has far fewer edges than $G(A)$ in practice, an implementation that processes G^- rather than $G(A)$ promises to be substantially faster; we see in Section 4 that this is indeed the case.

The skeleton graph G^- is closely related to the fundamental supernodes of A, and can be computed efficiently in linear time by a simple modification of Liu, Ng, and Peyton's algorithm [20] to find fundamental supernodes. Indeed, that algorithm is a good framework for implementing our new algorithm, whether the skeleton graph is exploited or not. We can combine the two algorithms to obtain an efficient single-pass implementation. As this implementation processes the edges of $G(A)$, it discards edges not in the skeleton graph, and uses only the skeleton edges to compute the data for the row and column counts. If m^- is the number of edges in G^-, then this scheme runs in $O(m + m^- \alpha(m^-, n))$ time.

Section 3.1 below reviews the material we need from Liu et al. [20]. Section 3.2 presents a detailed version of the new combined implementation. Section 3.3 briefly describes our implementation of the disjoint set union algorithm for computing the least common ancestors, upon which the time complexity of our algorithm depends.

3.1 A fast algorithm for finding supernodes

Liu, Ng, and Peyton [20] introduced an $O(|A|)$ algorithm to compute a fundamental supernode partition. Their algorithm assumes that the elimination tree $T(A)$ has been computed and that the vertices are numbered by a postordering of $T(A)$. Let the *monotone adjacency set* of v, denoted by $madj[v]$, be the set of neighbors of v in the filled graph $G^+(A)$ that are numbered higher than v. Ashcraft and Grimes [2] defined a *fundamental supernode* as a maximal contiguous set of vertices $\{v, v+1, \cdots, v+s\}$ such that $v+i$ is the *only* child of $v+i+1$ in the elimination tree (for $i = 0, 1, \cdots, s-1$) and

$$madj[v] = madj[v+s] \cup \{v+1, v+2, \cdots, v+s\}.$$

The fundamental supernodes partition the vertices of $G(A)$.

In matrix terms, a supernode is any group of consecutive columns in L with a full diagonal block and with identical column patterns below the diagonal block. A fundamental supernode is maximal subject to the following condition: every column of the supernode except the last is an only child in the elimination tree. Liu et al. [20] give several reasons why fundamental supernodes are the most appropriate choice of supernodes for most applications, one of which is that they are independent of the choice of postordering for $T(A)$.

Finding the set of fundamental supernodes is equivalent to finding the first vertex of each supernode. These "first vertices" are characterized by the following result.

Theorem 5 (Liu, Ng, and Peyton [20]) *Vertex v is the first vertex in a fundamental supernode if and only if vertex v has two or more children in the elimination tree, or v is a leaf of some row subtree of $T(A)$.* $\square$

The key observation is that the vertices required by the row/column count algorithm (the p_i's and q_i's) are in fact first vertices of fundamental supernodes. It follows from the discussion immediately after Lemma 3 in Section 2.3 that the vertex pairs p_i, p_{i+1} whose least common ancestors must be found can be restricted to vertices that are leaves of some row subtree of $T(A)$. This is equivalent to restricting the algorithm in Figure 2 to the skeleton graph G^-. Furthermore, when the p_i's are restricted in this manner, it is clear that every least common ancestor $q_i = lca(p_i, p_{i+1})$ has two or more children. Consequently, the Liu et al. algorithm is an excellent vehicle for an efficient implementation of our new algorithm.

3.2 Detailed implementation of the new algorithm

The details of our single-pass, column-oriented implementation are given in Figure 3. Here we take $adj[p]$ to be the set of neighbors u of p in $G(A)$ for which $u > p$; thus each undirected edge of $G(A)$ is represented only once. Note that $p \notin adj[p]$. Again, the vertices are numbered by a postorder of the tree $T(A)$, but here no assumption is made concerning the order of the vertices in the adjacency lists $adj[p]$, nor are the least common ancestors computed in advance. Consequently, this implementation makes only a single pass through $G(A)$.

The vector of markers $prev\_p(u)$ stores the most recently visited vertex p' that is a leaf in $T_r[u]$. The pairs p, p' produced by the algorithm are precisely the multiset consisting of every consecutive pair of leaves in every row subtree $T_r[u]$. The reason for this is that one of the **if** tests in the algorithm screens out all edges in $G(A)$ except those in the skeleton graph G^-. The lines marked with asterisks have been added to the algorithm solely for this purpose. Of these, the key line is the test for whether or not the first (i.e., lowest numbered) descendant of p ($fst\_desc(p)$) is greater than the most

```
    Sort the vertices by a postorder of T(A);
    Assign the higher-numbered neighbors of u to adj[u], for all u;
    Compute level(u) as the distance from u to n (the root), for all u;
*   Compute fst_desc(u) as the first (least) descendant of u in T(A), for all u;
    prev_p(u) ← 0, for all u;
*   prev_nbr(u) ← 0, for all u;
    rc(u) ← 1, for all u;
    wt(u) ← 0, for all non-leaves u in T(A);
    wt(u) ← 1, for all leaves u in T(A);
    for p ← 1 to n do
        if p ≠ n then
            wt(parent(p)) ← wt(parent(p)) − 1;
        end if
        for u ∈ adj[p] do
*           if fst_desc(p) > prev_nbr(u) then
                wt(p) ← wt(p) + 1;
                p' ← prev_p(u);
                if p' = 0 then
                    rc(u) ← rc(u) + level(p) − level(u);
                else
                    q ← FIND(p');
                    rc(u) ← rc(u) + level(p) − level(q);
                    wt(q) ← wt(q) − 1;
                end if
                prev_p(u) ← p;
*           end if
*           prev_nbr(u) ← p;
        end for
        UNION(p, parent(p));
    end for
    cc(v) ← wt(v), for all v;
    for v ← 1 to n − 1 do
        cc(parent(v)) ← cc(parent(v)) + cc(v);
    end for
```

Figure 3: Implementation of algorithm to compute row and column counts.

recently visited vertex in the lower adjacency set of u, namely the vertex stored in the marker variable $prev\_nbr(u)$. It is not difficult to verify that when the condition holds true, no descendant of p is adjacent to u in $G(A)$; hence p is indeed a leaf in $T_r[u]$. For full details of this test, see Liu et al. [20].

The implementation is correct with or without the starred lines. We have implemented both versions: we call the one with the starred lines the *supernodal* version, and the one without these lines the *nodal* version.[1] We experiment with both versions of the algorithm in our tests in Section 4.

3.3 Disjoint set union

In order to compute least common ancestors, the algorithm in Figure 3 must manipulate disjoint sets of vertices, each of which induces a subtree of the elimination tree. The highest numbered vertex in each set (the root of the subtree) is used to "name" the set, and is called the *representative vertex* of the set. Initially each vertex p from 1 to n is a singleton set. As the algorithm proceeds, it executes a sequence of FIND and UNION operations, which are defined as follows.

- FIND(p): return the representative vertex of the unique set that contains p.
- UNION(u, v): combine the two distinct sets represented by u and v into a single set, which will be represented by the larger of u and v.

It is straightforward to verify that the call to FIND(p') in our algorithm returns $lca(p', p)$; see Tarjan [30] for details.

Each disjoint set is implemented as a tree stored using a parent vector (not to be confused with the *parent* vector in the elimination tree). The operation UNION(u, v) joins the two distinct trees represented by u and v together by making one of the roots a child of the other root. Consequently, UNION is a constant-time operation. This is not the case for FIND. The operation FIND(p) traces the *find path* from p to the root of p's tree. This root either is the representative vertex or contains a pointer to the representative vertex, depending on the implementation of UNION.

Tarjan [31] describes several techniques to shorten the find paths and thus reduce the amount of work spent on the FIND operations. *Union by*

[1] In the nodal version, $prev\_p(u)$ functions precisely as $prev\_nbr(u)$ does in the supernodal version.

rank makes the shorter tree's root a child of the taller tree's root in UNION, which tends to keep the trees short and bushy. With no other enhancements, union by rank ensures that find paths are no longer than $O(\log_2(n))$. This is usually combined with one of two techniques for shortening the find path during a FIND operation. The first of these is *path compression*, which, after finding the root, makes the parent for each vertex on the find path point to the root during a second pass along the path. Alternatively, *path halving* resets the parent pointer for every other vertex on the find path to point to its grandparent. Path compression shortens the find path more but requires two passes over the find path; path halving needs only one pass.

Tarjan [29, 31] showed that when union by rank is combined with either path compression or path halving, any sequence of n UNION's and m FIND's takes only $O(m\,\alpha(m,n))$ time. Tarjan [30] pointed out how to use the disjoint set union algorithm to find the least common ancestors of an arbitrary set of pairs of vertices from the same tree; our implementation of the row and column count algorithm uses the same method. Consequently, we can implement the nodal version of our algorithm to run in $O(m\,\alpha(m,n))$ time, and similarly we can implement the supernodal version to run in $O(m + m^-\,\alpha(m^-,n))$ time.

Gabow and Tarjan [10] showed that if the order of the UNION operations is known in advance (as is the case in our problem), then disjoint set union can be implemented so that a sequence of n UNION's and m $(\geq n)$ FIND's takes only $O(m)$ time. Their sophisticated hybrid algorithm partitions the vertices into *microsets* and performs all the operations in a hierarchical fashion, using table look-up to answer queries within the microsets, and using the standard disjoint set union algorithm on the microsets themselves. We did not implement this algorithm; we believe its increased overhead would wipe out the difference between $O(m\,\alpha(m,n))$ and $O(m)$ in our application.

We implemented and tested the following six combinations:

1. no union by rank, no path compression or halving.
2. no union by rank, path compression.
3. no union by rank, path halving.
4. union by rank, no path compression or halving.
5. union by rank, path compression.
6. union by rank, path halving.

problem	brief description
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
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—Corps 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 building
BCSSTK24	Stiffness matrix—winter sports arena

Table 1: List of test problems.

We found surprisingly little difference in performance among the various options. Far more important is whether or not the row/column count processing is limited to the skeleton graph, as we shall see in the next section. We found that any gains due to union by rank were more than offset by the additional overhead required for its implementation. The third option—no union by rank, path halving—performed slightly better on most machines we tried. Path halving was clearly superior to path compression when the skeleton adjacency structure was not exploited. Consequently, we recommend path halving to those implementing the method, and in the next section all our timings were obtained using path halving and no union by rank.

4 Experimental results

We have run the new algorithms on several problems from the Harwell-Boeing sparse matrix collection [8]. Table 1 lists our test problems, and Table 2 contains the problem statistics that have a bearing on the observed performance of our algorithms. Throughout this section `supcnt` refers to the "supernodal" version of the algorithm (Figure 3 with the starred lines), which identifies the edges of the skeleton graph G^{-} and uses only those edges in its row and column count calculations, and `nodcnt` refers to the "nodal" version of the algorithm (Figure 3 without the starred lines), which uses all the edges of $G(A)$.

problem	dimension n	edges in $G(A)$ m	edges in G^- m^-	edges in G^+ m^+
NASA1824	1824	18692	3565	71875
NASA2910	2910	85693	8113	201493
NASA4704	4704	50026	9672	276768
BCSSTK13	2003	40940	5598	269668
BCSSTK14	1806	30824	4352	110461
BCSSTK15	3948	56934	13186	647274
BCSSTK16	4884	142747	11665	736294
BCSSTK17	10974	208838	24569	994885
BCSSTK18	11948	68571	23510	650777
BCSSTK23	3134	21022	8500	417177
BCSSTK24	3562	78174	6977	275360

Table 2: Problem statistics.

4.1 Performance of the disjoint set union options

The primary purpose of Table 3 is to explain two things we have observed in our tests: (1) why exploiting the skeleton graph is so beneficial, and (2) why the various disjoint set union (DSU) implementation options have so little influence on the performance of our code. The number of FIND operations required by `nodcnt` and `supcnt` is bounded above by m and m^-, respectively, and bounded below by $m - n$ and $m^- - n$. Thus, the huge difference between the number of FIND's required by `nodcnt` and the number of FIND's required by `supcnt` (see Table 3) simply reflects the fact that the skeleton graph of A is typically much sparser than the graph of A (see Table 2).

Each FIND(p) operation traverses the find path in p's tree beginning at p and ending at the root of the tree. The average number of vertices on these find paths is reported for each DSU implementation. We tested only two options for `nodcnt`: path compression and path halving, both without union by rank. Note that the average number of vertices on a find path ranges from 2 to 2.7, with path compression faring slightly better than path halving. The performance of path compression suffers, however, because the find path must be traversed twice, compared with once for path halving. Our tests indicate that path halving does indeed substantially outperform path compression, and in `nodcnt`, where the number of FIND's is large, the

problem	`nodcnt`			`supcnt`						
	vertices/path			vertices/path						
	PC	PH	FIND's	none		PC		PH		FIND's
	NR			NR	R	NR	R	NR	R	
NASA1824	2.1	2.3	17050	4.1	1.9	2.3	1.6	2.5	1.6	1923
NASA2920	2.0	2.1	83071	3.6	1.6	2.1	1.4	2.2	1.4	5491
NASA4704	2.2	2.3	45809	4.1	1.9	2.2	1.6	2.4	1.6	5455
BCSSTK13	2.1	2.2	39125	5.5	2.2	2.2	1.8	2.4	1.8	3783
BCSSTK14	2.1	2.2	29200	4.2	1.7	2.2	1.5	2.3	1.5	2728
BCSSTK15	2.2	2.3	53468	4.7	2.0	2.2	1.6	2.3	1.7	9720
BCSSTK16	2.1	2.1	138121	4.4	2.0	2.1	1.7	2.2	1.7	7039
BCSSTK17	2.1	2.1	199092	4.1	1.9	2.2	1.6	2.2	1.6	14823
BCSSTK18	2.3	2.5	59624	5.5	2.2	2.5	1.8	2.8	1.9	14563
BCSSTK23	2.4	2.7	18419	5.8	2.4	2.4	1.9	2.6	1.9	5897
BCSSTK24	2.0	2.1	74762	3.8	1.7	2.1	1.6	2.2	1.6	3565

Table 3: Average number of vertices on a find path for DSU implementation options: PC is path compression, PH is path halving, R is union by rank, and NR is no union by rank

gain in efficiency is substantial.

We tried all six options mentioned in Section 3.3 in our implementations of `supcnt`, and as noted earlier, we saw little difference in performance from one option to the next. The primary explanation for this phenomenon is the small proportion of `supcnt`'s total work devoted to DSU operations. The number of FIND operations is small relative to m, and the average number of vertices on a find path is small (from 1.4 to 2.6) for five of the six options tested. For the sixth option (no DSU enhancements), the average number of vertices on a find path is still quite modest (from 3.6 to 5.8), with less work required for each vertex visited. Consequently, even this option is competitive in our tests.

When path compression or path halving is used, union by rank obtains only modest reductions in the average number of nodes visited. The overhead costs associated with union by rank more than offset any advantages conferred by the technique. Comparing path compression and path halving with no union by rank, the same observations made previously for `nodcnt` hold for `supcnt` also. The primary difference is that the total work associated with DSU operations in `supcnt` is so small that the performance edge of path halving over path compression is quite small. Nonetheless, path halving with no union by rank has proven most effective overall and has the added advantage of simplicity. Finally, note that for our chosen option the total number of vertices visited by FIND operations is much less than m for most of the test problems.

4.2 Performance of the row and column count algorithm

We coded `nodcnt` and `supcnt` in Fortran 77 and ran our tests on an IBM RS/6000 (model 320). We used the standard Fortran compiler and compiler optimization flag (`xlf -O`). We used a high-resolution timer (`readrtc`) to obtain our timings on this machine, repeating each run ten times in succession and returning the average elapsed time. The results are shown in Table 4. We used path halving and no union by rank in the implementation of the disjoint set union algorithm for both `nodcnt` and `supcnt`. The

problem	e-tree	post-ordering	row/column counts			super-nodes
			Liu's	new		
			`lnzcnt`	`nodcnt`	`supcnt`	
NASA1824	.035	.006	.076	.047	.038	.031
NASA2920	.156	.009	.256	.198	.144	.128
NASA4704	.096	.016	.261	.128	.104	.085
BCSSTK13	.078	.006	.238	.098	.074	.064
BCSSTK14	.057	.005	.118	.074	.056	.048
BCSSTK15	.108	.013	.513	.142	.113	.091
BCSSTK16	.262	.016	.691	.331	.239	.216
BCSSTK17	.391	.037	.965	.500	.408	.329
BCSSTK18	.144	.040	.549	.197	.181	.141
BCSSTK23	.044	.010	.310	.059	.054	.039
BCSSTK24	.143	.012	.295	.184	.134	.120

Table 4: Run times in seconds on an IBM RS/6000 (model 320).

time required to compute the elimination tree and postordering are of interest for two reasons. First, they must be computed before the row/column counts can be computed. Second, the algorithm for computing the elimination tree is, like `nodcnt` and `supcnt`, a single-pass $O(m\,\alpha(m,n))$ algorithm that relies on efficient implementation of the disjoint set union operations for efficiency. Thus it is interesting to compare its performance with that of the new algorithms.

Both `nodcnt` and `supcnt` are much more efficient than `lnzcnt`, the $O(|L|)$ algorithm from Liu [19]. Algorithm `nodcnt` is 1.29 to 5.25 times faster than `lnzcnt`, while `supcnt` is, in turn, 1.08 to 1.39 times faster than `nodcnt`. For every problem but one, `supcnt` is at least twice as fast as `lnzcnt`. (For NASA2920, `supcnt` is 1.77 times faster than `lnzcnt`.) For four of the problems, `supcnt` is more than three times faster than `lnzcnt`. For BCSSTK15 `supcnt` is 4.54 times faster, and for BCSSTK23 `supcnt` is 5.74 times faster.

Finally, it is interesting to compare the timings for the elimination tree algorithm [19] and the supernode algorithm [20] with those for `supcnt`. First, `supcnt` can be viewed as an extension of the supernode algorithm, and consequently the time for `supcnt` should be bounded below by the time for the supernode algorithm. Though there are some differences in the amount and kind of $O(n)$ work performed by the two algorithms before and after the main loop, the difference in the two timings can nevertheless be viewed as a crude measure of the cost of adding the instructions necessary to compute row and column counts to the supernode algorithm. Clearly, this cost is quite small, especially considering the simplicity and demonstrated practical efficiency of the supernode algorithm. Note also that the timings for `supcnt` and the elimination tree algorithm closely track each other. From these observations, we conclude that it is probably not possible to improve the performance of `supcnt` much beyond what we are currently observing.

5 Conclusion

We have considered in this paper the problem of predicting the row counts and column counts in the Cholesky factor L of a sparse symmetric positive definite matrix A, given the zero/nonzero structure of A and the elimination tree $T(A)$. We have presented new algorithms for determining the counts, the complexities of which are linear in $|A|$ times a slowly growing inverse of Ackermann's function; the previously known algorithms ran in $O(|L|)$ time. The key to the new algorithms is the computation of least common ancestors in a tree using the disjoint set union algorithm. We have investigated different ways of implementing the disjoint set union operations in our algorithms. Based on our experimental results, we conclude that path halving with no union by rank is the best technique for an efficient implementation of the disjoint set union algorithm.

We have further improved our new algorithms by exploiting the skeleton graph of A. We have demonstrated that the supernodal version is faster than the nodal version in all of the problems we tested. Moreover, both the nodal and supernodal versions are much more efficient than the previously known $O(|L|)$-time algorithms. We expect the algorithms described in this paper to be of practical use in a wide range of sparse matrix computations.

References

[1] 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.

[2] C. C. Ashcraft and R. G. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Software*, 15:291–309, 1989.

[3] H. Bodlaender, H. Hafsteinsson, J. R. Gilbert, and T. Kloks. Approximating treewidth, pathwidth, and minimum elimination tree height. *Proceedings of the 17th International Workshop on Graph-Theoretic Concepts in Computer Science.* Springer-Verlag Lecture Notes in Computer Science volume 570, 1992.

[4] 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.

[5] T. Coleman and Y. Li. Global and quadratic convergence of reflective Newton methods for nonlinear minimization subject to bounds. Technical Report in preparation, Cornell University Computer Science Department, Ithaca, New York, 1992.

[6] T. Coleman and Y. Li. A reflective newton method for minimizing a quadratic function subject to bounds on the variables. Technical Report in preparation, Cornell University Computer Science Department, Ithaca, New York, 1992.

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

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

[9] 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.

[10] H. N. Gabow and R. E. Tarjan. A linear time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30:209–221, 1985.

[11] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *J. Combinatorial Theory B*, 16:47–56, 1974.

[12] Alan George and Michael T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra and its Applications*, 34:69–83, 1980.

[13] 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.

[14] 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.

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

[16] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[17] J. G. Lewis, B. W. Peyton, and A. Pothen. A fast algorithm for reordering sparse matrices for parallel factorization. *SIAM J. Sci. Stat. Comput.*, 10:1156–1173, 1989.

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

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

[20] J. W-H. Liu, E. Ng, and B. W. Peyton. On finding supernodes for sparse matrix computations. To appear in SIAM J. Matrix Anal. Appl., 1993.

[21] F. J. Peters. *Sparse Matrices and Substructures.* Mathematisch Centrum, Amsterdam, The Netherlands, 1980. Mathematical Centre Tracts 119.

[22] 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.

[23] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R. C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

[24] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.

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

[26] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, 1983.

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

[28] R. E. Tarjan. Depth-first search and linear graph algorithms. *SICOMP*, 1:146–160, 1972.

[29] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.

[30] R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26:690–715, 1979.

[31] R. E. Tarjan. *Data Structures and Network Algorithms.* CBMS-NSF Regional Conference Series in Applied Math, SIAM Publications, 1983.