BIT
2001, Vol. 41, No. 4, pp. 693–711
0006-3835/01/4104-0693 $16.00

COMPUTING ROW AND COLUMN COUNTS FOR SPARSE QR AND LU FACTORIZATION *

J. R. GILBERT[1] and X. S. LI[2] and E. G. NG[3] and B. W. PEYTON[4] †

[1] *Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304-1314, USA. email: gilbert@parc.xerox.com*

[2] *Lawrence Berkeley National Laboratory, National Energy Research Scientific Computing Division, 1 Cyclotron Road, MS 50F, Berkeley, CA 94720, USA. email: xiaoye@nersc.gov*

[3] *Lawrence Berkeley National Laboratory, National Energy Research Scientific Computing Division, 1 Cyclotron Road, MS 50F, Berkeley, CA 94720, USA. email: EGNg@lbl.gov*

[4] *Oak Ridge National Laboratory, Computer Science and Mathematics Division, P. O. Box 2008, Oak Ridge, Tennessee 37831-6367, USA. email: peyton@msr.epm.ornl.gov*

Abstract.

We present algorithms to determine the number of nonzeros in each row and column of the factors of a sparse matrix, for both the QR factorization and the LU factorization with partial pivoting. The algorithms use only the nonzero structure of the input matrix, and run in time nearly linear in the number of nonzeros in that matrix. They may be used to set up data structures or schedule parallel operations in advance of the numerical factorization.

The row and column counts we compute are upper bounds on the actual counts. If the input matrix is strong Hall and there is no coincidental numerical cancellation, the counts are exact for QR factorization and are the tightest bounds possible for LU factorization.

These algorithms are based on our earlier work on computing row and column counts for sparse Cholesky factorization, plus an efficient method to compute the column elimination tree of a sparse matrix without explicitly forming the product of the matrix and its transpose.

AMS subject classification: 65F20.

Key words: sparse QR and LU factorizations, column elimination tree, row and column counts, disjoint set union.

1 Introduction

Let A be an $m \times n$ (sparse) matrix, with $n \leq m$, and suppose A has full column rank n. We consider two nonsymmetric factorizations of A. The orthogonal

*Received January 2001. Revised August 2001. Communicated by Åke Björck.

†This work was supported in part by the Director, Office of Advanced Scientific Computing Research, Division of Mathematical, Information, and Computational Sciences of the U. S. Department of Energy under contract number DE-AC03-76SF00098; in part by the Applied Mathematical Sciences Research Program, Office of Science, U. S. Department of Energy, under contract DE-AC05-00OR22725 with UT-Batelle, LLC; and in part by DARPA under contract DABT63-95-C-0087.

factorization is

$$A = Q \begin{bmatrix} R \\ O \end{bmatrix},$$

where Q is an $m \times m$ orthogonal matrix and R is an $n \times n$ upper triangular matrix. The factorization is usually computed by Householder transformations, and Q is then represented compactly by the "Householder matrix", which is the $m \times n$ lower trapezoidal matrix H whose columns are the Householder vectors. Second we consider the triangular factorization with partial pivoting $PA = LU$,[1] where L is unit lower triangular, U is upper triangular, and P is a permuation matrix describing row interchanges. We may write this factorization in the form

$$A = P_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1} U,$$

where P_i is a permutation that just transposes row i and a higher-numbered row, and L_i is a Gauss transform (a unit lower triangular matrix with nonzeros only in column i). Column i of L has the same nonzero values as column i of L_i, but in a different order. We write $\hat{L}$ to denote the lower triangular matrix whose i^{th} column is that of L_i. Thus L and $\hat{L}$ have the same nonzero values, but in different orders within each column.

1.1 Summary of results

Our goal in this paper is to compute the number of nonzeros in each row and each column of the matrices H, R, $\hat{L}$, and U, given only the nonzero structure of A. The column counts for L are the same as those for $\hat{L}$. We also describe a method (which has been alluded to but not published) to compute the "column elimination tree" of A, as defined below.

Our algorithms run in time almost linear in the number of nonzeros in A. Thus they may be used as a fast way to predict and allocate the storage necessary for the QR or LU factorization. In particular, our work was motivated by the LU factorization code SuperLU [3, 4, 18]. Both the sequential and parallel versions of SuperLU use the column elimination tree to cluster similarly-structured columns of L for efficiency; the shared-memory parallel version (called SuperLU_MT) also uses the tree for scheduling and the predicted column counts to allocate working storage.

The row and column counts for $\hat{L}$ and U cannot be predicted exactly from the nonzero structure of A, because the row interchanges during factorization depend on the numerical values. However, regardless of row interchanges, the structures of $\hat{L}$ and U are *subsets* of the structures of H and R respectively [12, 13]. Therefore the rest of this paper focuses on computing counts for H and R.

We shall make three assumptions about A.

1. Every entry on the main diagonal of A is nonzero. (Every matrix of full column rank has a row permutation that puts nonzeros on its main diagonal [5, 6, 8].)

[1] In this factorization the matrix A can also be rectangular. Here we consider only square matrices.

2. A has the strong Hall property (that is, its rows and columns cannot be permuted to yield a nontrivial block upper triangular form [2, 25]).

3. No coincidental numerical cancellation takes place during any arithmetic on A. (Formally, this is guaranteed in exact arithmetic if the nonzeros of A are algebraically independent.)

The references [2, 13, 15, 17, 24] contain more complete discussions of these assumptions and their consequences. These three assumptions imply that (1) the model we use in this paper for the nonzero structure of H and R is exact, not merely an upper bound, and (2) the nonzero structures of H and R are the tightest upper bounds on those of $\hat{L}$ and U that are possible without knowing any of the values of the nonzeros of A. Therefore, under these assumptions, our algorithms compute the exact row and column counts for H and R, and the tightest possible bounds for $\hat{L}$ and U.

If A does not satisfy these three assumptions then our algorithms compute upper bounds on all the row and column counts.

1.2 Relationship to the symmetric problem

The normal equations $A^TAx = A^Tb$ allow Cholesky factorization to replace nonsymmetric LU (for $Ax = b$) or QR (for full-rank least squares), at least in theory. The latter factorizations are usually preferable for numerical reasons. When using LU or QR factorization in the sparse case, however, there are many situations where the *nonzero structure* of the normal equations matrix A^TA and its Cholesky factor contain useful information about the combinatorial structure of A and its factors.

A drawback to using the structure of A^TA is that it may be expensive to compute and store. If A is sparse, A^TA is likely to be much less sparse. Thus one tries to represent A^TA implicitly rather than explicitly. We may think of the structure of A^TA as an undirected graph with a vertex for each column of A and a *clique* (a full submatrix) for each row of A. One could then represent A^TA implicitly by storing only A, and treating each row of A (with k nonzeros, say) as a clique of k^2 nonzeros in A^TA. This representation would save space but be expensive in time. To save time as well, we notice that not all the edges of such a clique are necessary to compute the row and column counts and the column elimination tree. Thus we do our computations on an intermediate graph that has only $k - 1$ edges, instead of k^2, for a row of A with k nonzeros.

We use two different intermediate graphs in our algorithms. The column elimination tree algorithm in Section 2 substitutes for A^TA a graph that has a *path* instead of a clique for each row of A. Unlike A^TA, that graph is no larger than A, but it has the same elimination tree as A^TA. In Section 3.3, we compute row and column counts for R by following our earlier algorithm to compute counts for the Cholesky factor [16] (on this column elimination tree), but substituting for A^TA a graph that has a *star* (a subgraph whose lowest-numbered vertex has an edge to every other vertex in the subgraph) for each row of A. That graph is no larger than A, but it has the same symbolic Cholesky

factor as A^TA. In Section 3.2, we compute row and column counts for H using A and the column elimination tree, following our earlier characterization [24] of the structure of H.

1.3 Notation and details

Throughout this paper, we assume some familiarity with elimination trees [21], supernodes [1, 22], and the use of graphs in sparse matrix computation [7, 9, 11]. The *structure* of the $m \times n$ matrix A, denoted by Struct$[A]$, is defined to be

$$\text{Struct}[A] := \{(i,j) \mid A_{ij} \neq 0\}.$$

We use $|A|$ to denote the number of nonzero entries in A; that is, $|A| = |\,\text{Struct}[A]|$. Then Struct$[A^TA]$ is the set of locations of nonzero entries of A^TA (under our assumption of no exact cancellation). We write

$$\text{Struct}[A_{i*}] := \{j \mid A_{ij} \neq 0\}$$

and

$$\text{Struct}[A_{*j}] := \{i \mid A_{ij} \neq 0\}$$

to denote the structures of row i and column j of matrix A.

The *graph of* A^TA, denoted by $G(A^TA)$, has the vertex set $\{1, 2, \ldots, n\}$. For $i \neq j$, there is an edge (i,j) in $G(A^TA)$ if and only if the (i,j)-element of A^TA is nonzero. Note that the (i,j)-element of A^TA is nonzero if and only if A_{ki} and A_{kj} are both nonzero for some k. Consequently, Struct$[A_{k*}]$ forms a clique in $G(A^TA)$.

The *elimination tree* of a symmetric positive definite matrix characterizes dependencies between columns in the Cholesky factorization of the matrix. Schreiber [26] defined this tree; Liu [20, 21] surveyed its applications in sparse factorization. The *column elimination tree* of A is the elimination tree of A^TA, which we denote $T(A^TA)$. (Strictly speaking, the column elimination tree of A is defined as the elimination tree of a symmetric matrix whose (i,j) entry is nonzero whenever columns i and j of A share a nonzero row. Since we assume no coincidental numerical cancellation, A^TA has that nonzero structure.)

The vertices of $T(A^TA)$ are the integers 1 through n, the same as those of $G(A^TA)$. Vertex i is the parent of vertex $j < i$ in the tree $T(A^TA)$ if the first off-diagonal nonzero entry in column j of the (lower triangular) Cholesky factor of A^TA is in row i.

1.4 Outline of paper

Section 2 shows how to compute the elimination tree $T(A^TA)$ using Struct$[A]$ as input rather than Struct$[A^TA]$. This algorithm has time complexity $O(|A|\,\alpha(|A|, n))$, where $\alpha(*,*)$ is the very slowly growing inverse of Ackermann's function [27]; thus, the algorithm is said to be "almost linear" in the size of its input. Section 3 is the heart of the paper: it shows how to compute the row counts and column counts of H and R using Struct$[A]$ and $T(A^TA)$ as input. This algorithm

also has time complexity $O(|A|\,\alpha(|A|, n))$. Section 4 presents experimental results demonstrating that the algorithms introduced here are much faster than the straightforward ones based on A^TA. Section 5 contains our concluding remarks.

2 Computing the column elimination tree

In this section we describe an algorithm to determine the column elimination tree $T(A^TA)$ of A without forming A^TA explicitly. The algorithm runs in time nearly linear in the number of nonzeros in A, which may be much less than the number of nonzeros in A^TA. While we believe this is the first published description, Liu [20] alludes to an algorithm with this running time and Matlab's built-in function `coletree` implements such an algorithm [14]. (The Matlab implementation replaces each row of A by a star, like our QR count algorithm in Section 3. We expect a path, as we use here, to require slightly less overhead for tree traversal in the disjoint set union algorithm, but we have not compared the path and star experimentally.)

We begin by reviewing the symmetric algorithm (whose running time is almost linear in the number of nonzeros in A^TA), and then show how to modify it to work from A rather than A^TA.

2.1 Review: The symmetric algorithm

To construct $T(A^TA)$ from Struct$[A^TA]$, the symmetric algorithm processes the rows of A^TA one at a time. After processing the first $i-1$ rows of A^TA, we have built the elimination tree for the $(i-1)\times(i-1)$ leading submatrix of A^TA. (This "tree" may actually be a forest with multiple connected components, each with its own root.) To process row i, we proceed up the tree from every vertex corresponding to an off-diagonal nonzero entry of row i of the lower triangular part of A^TA until we reach a root. Then we add vertex i to the tree, as well as edges to make i the new parent of all those roots. The algorithm is summarized in Figure 2.1.

```
for row i ← 1 to n do
    Add i to T(A^TA);
    for column j ∈ Struct[(A^TA)_{i*}] ∩ {1, 2, ..., i − 1} do
        Find root r of the current tree that contains j;
        if r ≠ i then
            parent(r) ← i;
        end if
    end for
end for
```

Figure 2.1: Algorithm for computing the elimination tree $T(A^TA)$ from Struct$[A^TA]$.

The most expensive part of the algorithm in Figure 2.1 is the tree traversals

to determine the roots. However, the efficiency of the tree traversals can be improved by various kinds of *path compression*, as used in disjoint set union algorithms [27]. We refer the reader to Liu [19, 21] for details of disjoint set union in elimination tree algorithms. If balancing and path compression are both used then the elimination tree algorithm has time complexity $O(|A^TA|\,\alpha(|A^TA|, n))$. Liu [21] recommends using path compression without balancing; the resulting algorithm has time complexity $O(|A^TA| \log n)$, but is faster in practice than the algorithm with both balancing and path compression.

2.2 The nonsymmetric algorithm

We now shift our attention to how $T(A^TA)$ can be computed from Struct$[A]$ rather than Struct$[A^TA]$.

Every off-diagonal nonzero entry in A^TA, and hence every edge of $G(A^TA)$, corresponds to an ancestor-descendant pair in $T(A^TA)$ [19, 21, 26]. In particular, if (i, j) and (j, k) are two edges in $G(A^TA)$ with $i < j < k$, then vertices i, j, and k must belong to the unique path joining i and k, with i being the descendant of both j and k.

For $1 \le i \le m$, let the column indices of the first and last nonzeros in row A_{i*} be denoted by f_i and ℓ_i, respectively:

$$\begin{aligned} f_i &:= \min\{j \mid j \in \text{Struct}[A_{i*}]\}, \\ \ell_i &:= \max\{j \mid j \in \text{Struct}[A_{i*}]\}. \end{aligned}$$

Since Struct$[A_{i*}]$ is a clique in $G(A^TA)$, it follows from the preceding discussion that Struct$[A_{i*}]$ must consist of a subset of vertices on the path in $T(A^TA)$ from f_i up to ℓ_i.

The relationship between the row structures of A and the paths in $T(A^TA)$ provides a way to avoid computation of Struct$[A^TA]$ from Struct$[A]$. The key observation follows. Choose $j \in$ Struct$[A_{i*}]$ such that $j \ne f_i$, and let k be the immediate predecessor of j in Struct$[A_{i*}]$. Since each vertex $k' < k$ taken from Struct$[A_{i*}]$ is a descendant of k in $T(A^TA)$, there is no need for the elimination tree algorithm to search any such edge (k', j), even though (k', j) is a nonzero entry in A^TA. Instead it needs to examine only edges joining a vertex with its immediate predecessor in a row list Struct$[A_{i*}]$.

We are essentially using the symmetric elimination tree algorithm, not on the graph of A^TA, but on a graph that has a path for every row of A. This graph has the same elimination tree as A^TA but is only as large as A. The details are shown in Figure 2.2.

The vector *parent* is the parent function of the column elimination tree algorithm. Array element *prev_col*(i) records the predecessor of the next vertex in Struct$[A_{i*}]$ to be processed. As in the elimination tree algorithm, when the root r of a subtree containing k is found, path compression can be employed to reduce the time required to perform tree traversals. Thus, the column elimination tree algorithm has time complexity $O(|A|\,\alpha(|A|, n))$ if balancing and path compression are both used.

```
for column j ← 1 to n do
    parent(j) ← j; prev_col(j) ← 0;
end for
for column j ← 1 to n do
    for row i ∈ Struct[A_{*j}] do
        k ← prev_col(i);
        if k ≠ 0 then
            /* f_i < j */
            find root r of the current tree that contains k;
            if r ≠ j then
                parent(r) ← j;
            end if
        end if
        /* record j as predecessor of next vertex in Struct[A_{i*}] */
        prev_col(i) ← j;
    end for
end for
```

Figure 2.2: Algorithm for computing the column elimination tree $T(A^TA)$ from Struct$[A]$.

3 Row and column counts for sparse QR factorization

We begin by reviewing the efficient algorithms to compute row and column counts for Cholesky factorization [16]. Then we modify these algorithms for sparse QR factorization, so that they work on Struct$[A]$ rather than Struct$[A^TA]$. Recall that (for square A) the counts for H and R are also the best possible a priori bounds on the counts for the partial-pivoting factors L and U respectively.

3.1 Review: Row and column counts for Cholesky factors

This material is condensed from Gilbert, Ng, and Peyton [16], which has a much more leisurely explanation of the algorithms and why they work. Here we just describe the details that will later need to be modified in order to use A rather than A^TA. The reader should glean two key points from this subsection: first, the Cholesky row counts are derived by summing the lengths of certain paths in the elimination tree; second, the Cholesky column counts are derived by summing certain vertex weights in the tree. The paths and the vertex weights are both defined in terms of the leaves (and possibly other vertices) of the "row subtrees", and of certain least common ancestors of those vertices. The modifications for sparse QR factorization will consist of methods for identifying the crucial subtrees, leaves, and ancestors for H and R rather than for the Cholesky factor, and for doing so from A rather than from A^TA.

In this section, let B be an arbitrary sparse $n \times n$ symmetric positive definite matrix, and let L_B be its (lower triangular) Cholesky factor. Let $T(B)$ be the elimination tree of B. Recall [21] that the nonzeros in any row i of L_B induce a

connected subgraph of $T(B)$ called the i^{th} *row subtree* and written $T_i(B)$; and that every leaf of $T_i(B)$ corresponds to a nonzero column in row i of B.

Determining the Cholesky row counts from $T(B)$ is straightforward. The i^{th} row count of L_B is simply the number of vertices in the i^{th} row subtree $T_i(B)$, which we can get by counting the edges in $T_i(B)$. To get the desired running time we must count those edges in time that depends only on the number of nonzeros in row i of B, not on the size of $T_i(B)$. The solution is as follows. Assume that $T(B)$ is postordered. Then (for each i) the edges of $T_i(B)$ are partitioned into disjoint paths by the vertices corresponding to the subdiagonal nonzeros in row i of B, together with the *least common ancestors* of consecutive pairs of such vertices. Thus, the i^{th} row count can be obtained by computing the sum of the lengths of these disjoint paths. This can be implemented in $O(|B|\alpha(|B|, n))$ time, which is dominated by the running time of the disjoint set union algorithm used to find the least common ancestors [27].

Determining the column counts is more complicated. The j^{th} column count is the number of row subtrees that contain vertex j. We could simply traverse every row subtree, keeping a count for each j; but that would take time proportional to the total size of the Cholesky factor. The efficient algorithm counts all the row subtrees containing each j simultaneously in a single traversal of all of $T(B)$, using vertex weights defined in terms of the same least common ancestors as above.

For each pair of vertices i and j (with $1 \le i, j \le n$), define the characteristic function χ by

$$\chi_i(j) = \begin{cases} 1 & \text{if } (L_B)_{ij} \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Define the vertex weight function w by

$$w_i(j) = \chi_i(j) - \sum_{\text{children } k \text{ of } j} \chi_i(k) \ ,$$

and

$$w(j) = \sum_{\text{all } i} w_i(j) \ .$$

With the convention that the descendants of j include j, the definition of $w_i(j)$ implies that

$$\chi_i(j) = w_i(j) + \sum_{\text{children } k \text{ of } j} \chi_i(k) = \sum_{\text{descendants } s \text{ of } j} w_i(s) \ .$$

Therefore, the j^{th} column count, denoted by $cc(j)$, is given by

$$cc(j) = \sum_{\text{all } i} \chi_i(j) = \sum_{\text{all } i} \sum_{\text{descendants } s \text{ of } j} w_i(s) = \sum_{\text{descendants } s \text{ of } j} w(s).$$

That is, the j^{th} column count is precisely the sum of the weights of the descendants of j in $T(B)$. If we can compute the weights, computing the column counts involves merely accumulating the weights from leaves to the roots in $T(B)$.

The vertex weights w are not difficult to compute. Suppose that vertex j has d children in the i^{th} row subtree. Then by definition

$$w_i(j) = \chi_i(j) - d,$$

and the only nonzero values of $w_i(j)$ are

$$w_i(j) = \begin{cases} -1, & \text{if } j \text{ is the parent of } i \text{ in } T(B), \\ 1, & \text{if } j \text{ is a leaf of } T_i(B), \\ 1-d, & \text{if } j \text{ belongs to } T_i(B) \text{ and has } d > 1 \text{ children in } T_i(B). \end{cases}$$

The vertices j for which $w_i(j) \neq 0$ are easy to identify. By definition, every leaf of $T_i(B)$ corresponds to a nonzero entry in row i of B. Every vertex in $T_i(B)$ with more than one child in $T_i(B)$ is the least common ancestor of a pair of consecutive nonzero entries in row i of B, and hence is one of the vertices identified in the row count algorithm above.

In summary, we compute the vertex weights by processing the row subtrees $T_i(B)$ one by one, and adding each nonzero contribution $w_i(j)$ from the i^{th} row subtree to the appropriate $w(j)$. After computing all the $w(j)$'s, we obtain the column counts by traversing $T(B)$ from its leaves to its roots.

Again, we refer the reader to the paper [16] for the details of the row and column counts algorithm for Cholesky factors.

3.2 Row and column counts of the Householder matrix

Computing row and column counts of the $m \times n$ Householder matrix H is similar to computing Cholesky counts, because the structure of H has a tree-based characterization similar to (actually, simpler than) that of a Cholesky factor. To see why, we need the following result, which is due to George, Liu, and Ng.

THEOREM 3.1 (ROW STRUCTURE OF H). [12] *Let $T = T(A^TA)$ be the column elimination tree of A. Let f_i denote the column index of the first nonzero entry in row A_{i*}. Then* Struct$[H_{i*}]$ *consists of all the vertices on the path in T from f_i to either i or the root of T, whichever is smaller. (Vertex i is an ancestor of f_i in T because $A_{ii} \neq 0$. If T is not connected, the root in question is that of the component containing f_i.)*

The paths in this theorem play the role of the row subtrees in the Cholesky count algorithms. Since now each "subtree" consists of a single path joining a descendant to an ancestor, the algorithms are simpler. Each row count is just the length of the corresponding path in vertices, which is one more than the difference between the levels of its endpoints. Thus all the row counts can be computed in $O(|A|)$ time from T and A.

The column counts for H can also be computed just as in the Cholesky case described above, but the weights are simpler because each row subtree (path) has only one leaf. Using the notation in Section 3.1, $w_i(j)$ is nonzero if and only if either j is the parent of the last vertex of the path for row i of H (i.e., the root of the i^{th} row subtree), in which case $w_i(j) = -1$; or j is the leaf of the path for row i of H (i.e., $j = f_i$), in which case $w_i(j) = 1$.

The complete column-oriented algorithm to compute the row and column counts of the Householder matrix H (denoted by rc_H and cc_H) appears in Figure 3.1. In this algorithm, the subtree roots $r(i)$ can be computed as part of the elimination tree algorithm (Section 2). Computing all the vertex levels $level(j)$ (distances to the root) takes $O(n)$ time. The f_i's can be determined from Struct$[A]$ in $O(|A|)$ time. The set $first[j]$ contains each row whose first nonzero entry is in column j; all those sets can be constructed from Struct$[A]$ in $O(|A|)$ time. Thus the overall running time of the algorithm is $O(|A|)$.

Sort the columns of A by a postorder of $T(A^TA)$;
Compute $level(j)$ as the distance from j to the root of j's subtree, for $1 \le j \le n$;
Compute f_i as the column index of the first nonzero in row A_{i*}, for $1 \le i \le m$;
Compute $first[j] = \{i \mid f_i = j\}$, for $1 \le j \le n$;
Let $r(i)$ be the smaller of i or the root of the subtree containing f_i;
$w(j) \leftarrow 0$, for $1 \le j \le n$;
for column $j \leftarrow 1$ **to** n **do**
    for $i \in first[j]$ **do**
        $rc_H(i) \leftarrow 1 + level(j) - level(r(i))$;
        $w(j) \leftarrow w(j) + 1$; /* for the leaf */
        if $r(i)$ is not the root of a subtree **then**
            $w(parent(r(i))) \leftarrow w(parent(r(i))) - 1$; /* for the parent */
        end if
    end for
end for
$cc_H(j) \leftarrow w(j)$, for $1 \le j \le n$;
for $j \leftarrow 1$ **to** $n - 1$ **do**
    if j is not the root of a subtree **then**
        $cc_H(parent(j)) \leftarrow cc_H(parent(j)) + cc_H(j)$;
    end if
end for

Figure 3.1: The column-oriented algorithm to compute row and column counts of the Householder matrix.

3.3 Supernodal structure of H and L

A *supernode* of a lower triangular factor of a matrix is a block of consecutive columns that have identical nonzero structures, except that the square subblock on the diagonal is a full triangle. Many factorization algorithms gain efficiency by using high-level BLAS—that is, optimized dense matrix kernels—to operate on supernodes as dense blocks [1, 3].

We can determine the supernodal structure of the Householder matrix H, which gives a useful bound on the supernodal structure of the partial pivoting factor L, during the row and column count algorithm in Figure 3.1. Specifically, we identify the *fundamental supernodes* of H. The vertices corresponding to columns of a supernode form a path in the column elimination tree $T(A^TA)$. A

fundamental supernode is a supernode that is maximal subject to the property that every vertex on that path, with the possible exception of the first one, has exactly one child in the tree. See Ng and Peyton [23] for more on fundamental supernodes.

The following theorem, which is due to Li [18], characterizes the fundamental supernodes of H in terms of the column elimination tree.

THEOREM 3.2 (SUPERNODAL STRUCTURE OF H). [18] *Let $T = T(A^TA)$ be the column elimination tree of A, and assume that T is postordered. Let H be the Householder matrix. Vertex j is the first vertex in a fundamental supernode of H if and only if vertex j has two or more children in T, or j is the column index of the first nonzero entry in some row of A.*

PROOF. If vertex j has two or more children in T, then it is the first vertex of a fundamental supernode by definition. Suppose, then, that vertex j has only one child. Since T is postordered, the child is $j-1$.

"if" part: Let A_{ij} be the first nonzero entry in row i of A. Theorem 3.1 implies that $j \in \text{Struct}[H_{i*}]$ and $j-1 \notin \text{Struct}[H_{i*}]$. Therefore $\text{Struct}[H_{*j}] \not\subseteq \text{Struct}[H_{*\,j-1}]$; thus column j must begin a new supernode.

"only if" part: Suppose j is the first column of its fundamental supernode. Then $\text{Struct}[H_{*j}] \not\subseteq \text{Struct}[H_{*\,j-1}]$. Hence there exists a row i such that $j-1 \notin \text{Struct}[H_{i*}]$ but $j \in \text{Struct}[H_{i*}]$. If there is a $k \le j-1$ such that $A_{ik} \neq 0$, then $j-1 \in \text{Struct}[H_{i*}]$, because $j-1$ is on the path in T from k to $j \in \text{Struct}[H_{i*}]$, contrary to our assumption that $j-1 \notin \text{Struct}[H_{i*}]$. Therefore we must have $A_{ik} \neq 0$ for all $k \le j-1$. Since $j \in \text{Struct}[H_{i*}]$ and moreover $A_{ik} \neq 0$ for $k \le j-1$, it follows by Theorem 3.1 that $A_{ij} \neq 0$; hence, A_{ij} is the first nonzero of row i of A. □

It is straightforward to check the conditions of this theorem during the Householder counts algorithm, thus determining the supernodal partition of H.

Precomputing the fundamental supernodes of H can be useful in implementing partial pivoting on some parallel machines. Note that every fundamental supernode of L is contained in a fundamental supernode of H. This is because the first vertex of a fundamental supernode of H is necessarily the first nonzero column index in some row of A (as in the proof of Theorem 3.2), and therefore will also be the first nonzero column index in some row of L.

In our shared-memory parallel partial pivoting code SuperLU_MT [4, 18], it is useful to preallocate storage for a supernode of L before computing it. This is because each supernode may be divided into *panels* whose computation is assigned to different processors. If panels are stored in the order they are computed then panels from supernodes in different subtrees of T may end up intermingled, preventing the use of dense matrix kernels on entire supernodes. Instead, we allocate storage for a supernode of H at a time, and use that storage for all the enclosed supernodes of L. It is then impossible for a panel of one supernode to be stored between two panels of another supernode. (The SuperLU_MT code also refines this upper bound on a supernode's storage dynamically during the LU factorization [18].)

3.4 Row and column counts of the upper triangular factor

We turn our attention to the upper triangular factor R in the QR factorization of A. This R is the transpose of the Cholesky factor of A^TA [10]. Thus we could simply take $B = A^TA$ in the Cholesky count algorithm (Section 3.1). The only difficulty is that A^TA may have many more nonzeros than A, so the running time might be far from linear in $|A|$.

Therefore, instead of taking $B = A^TA$, we choose a B that has at most $|A|$ nonzeros but whose symbolic Cholesky factor has the same structure as the Cholesky factor of A^TA. Specifically, for each row of A, we include in the graph of B a *star*, with edges from the first nonzero column in that row to all the others. Using a clique instead of a star for each row of A would yield the structure of A^TA. However, the symbolic Cholesky factors of B and of A^TA are the same, because elimination of the first vertex in a row star of B causes fill among all the other vertices in that star. (A path, as we used in Section 2, would not in general lead to the same symbolic Cholesky factor.)

Here are the details of the computation. We assume that the column elimination tree $T(B) = T(A^TA)$ has been formed by the algorithm in Section 2, and that its vertices have been postordered. We do not literally form B; we just modify the Cholesky counts algorithm [16, Fig. 3] to extract the necessary information from A. In particular, that algorithm must identify in $T(A^TA)$ the leaves of the row subtrees $T_j(A^TA)$ and their least common ancestors.

Consider the row subtree $T_j(A^TA)$. Any leaf $k \neq j$ of that subtree has $k < j$ and is adjacent to j in $G(A^TA)$. Since $G(A^TA)$ is the union of one clique for each row of A, there must be some row i of A in which both A_{ik} and A_{ij} are nonzero. In fact, A_{ik} is the *first* nonzero in row i of A; otherwise, the first nonzero would be a proper descendant of k in $T_j(A^TA)$ and k would not be a leaf of the subtree.

As above, write f_i as the column index of the first nonzero in row i of A. We conclude that every leaf of a row subtree is an f_i. (Not every f_i need be a leaf of a row subtree.) Consequently, for each row i of A, it suffices for the Cholesky count algorithm to examine only the subset

$$\{(f_i, j) \mid j \in \mathrm{Struct}[A_{i*}] \text{ and } j > f_i\}$$

of the edges of A^TA—that is, the edges of B. There are fewer than $|A|$ such edges, so the running time is $O(|A|\alpha(|A|, n))$ as desired.

Figure 3.2 gives the detailed algorithm to find rc_R and cc_R, the row and column counts for R. The algorithm is column-oriented, and processes the first-vertex sets $\mathit{first}[j] = \{i \mid f_i = j\}$ in postorder. (These correspond to the centers of the stars in B.) For each first-vertex set $\mathit{first}[j]$, the algorithm needs to examine the higher-numbered adjacent vertices in $G(A^TA)$, that is, $u \in \mathrm{Struct}[A_{i*}]$ with $u > f_i$, for all i such that $f_i = j$. We therefore define the higher adjacency set $\mathit{hadj\_f}[j]$ for the first vertex set $\mathit{first}[j]$ as follows:

$$\mathit{hadj\_f}[j] := \bigcup_{i \in \mathit{first}[j]} \{u \mid u \in \mathrm{Struct}[A_{i*}], u > j\} \ .$$

This is the set of vertices in stars of B (or cliques of A^TA) whose center (or lowest-numbered vertex) is j.

The algorithm in Figure 3.2 includes one further optimization, as used by Gilbert, Ng, and Peyton [16]. The one-line test

$$\textbf{if } \mathit{fst\_desc}(j) > \mathit{prev\_nbr}(u) \textbf{ then}$$

discards some edges that do not affect the result, leaving only a minimal graph called the *skeleton graph* [19]. The resulting algorithm examines only those edges (f_i, j) such that f_i is a leaf in the row subtree of j. If e^- is the number of edges in the skeleton graph (which is at most $|A|$ and often much smaller), the overall time complexity is $O(|A| + e^- \alpha(e^-, n))$.

3.5 Combined algorithm

Since the algorithms for H (Figure 3.1) and R (Figure 3.2) both process Struct$[A]$ columnwise, and they use several common data structures, it is straightforward to combine the two algorithms into a single-pass algorithm that computes the row and column counts for both H and R. The timings we report in the next section are for this combined algorithm.

Our implementation theoretically has time complexity $O(|A|+e^- \log n)$ rather than $O(|A| + e^- \alpha(e^-, n))$, because we implement the disjoint set union algorithms without balancing. Gilbert, Ng, and Peyton [16] found this variant to be the fastest in practice for the Cholesky count algorithms, just as Liu [21] did for symmetric elimination tree algorithms.

4 Numerical results

In this section, we present the performance of the new algorithms on a variety of square and rectangular matrices, as described in Table 4.1. The square matrices are available from the `Matrix Market`[2] or the University of Florida.[3] The first three term-document matrices are available from the Cornell SMART system.[4] Hongyuan Zha provided matrix NEWSGROUP; we removed its dense rows in our test. Yin Zhang provided the three matrices from optimization. We performed our experiments on an IBM RS/6000-590 with a CPU clock rate of 66.5 MHz. We used the `AIX xlc` compiler with `-O3` optimization.

Table 4.2 reports results for the square matrices. We show the time to compute the column elimination tree and the QR counts using the algorithms in Sections 2 and 3. For comparison, we show the time to form Struct$[A^TA]$ explicitly, and to compute the elimination tree and Cholesky counts from that structure. We also show the time for LU factorization with partial pivoting as implemented in SuperLU [3]. The columns of A are preordered using minimum degree on Struct$[A^TA]$ before the factorization.

The algorithm from Section 2 to compute the column elimination tree $T(A^TA)$ from Struct$[A]$ is usually faster than the older algorithm to compute the tree

[2] http://math.nist.gov/MatrixMarket/
[3] http://www.cis.ufl.edu/~davis/sparse/
[4] ftp.cs.cornell.edu:pub/smart/

given Struct$[A^TA]$. When we include the time to form Struct$[A^TA]$ in the latter, we find that the newer algorithm (computing $T(A^TA)$ directly from Struct$[A]$) is much faster than the older one (computing $T(A^TA)$ from Struct$[A]$ by way of Struct$[A^TA]$); the speedups range from 1.1 to 266 fold, with an average of 34. This shows that forming and manipulating Struct$[A^TA]$ is often very expensive compared to the alternatives.

The runtime of the QR count algorithm is comparable to that of the column elimination tree algorithm, as we might expect since both are dominated by disjoint set union operations. In any setting where the column elimination tree is desired, computing row and column counts adds little extra time.

The last two columns of the table show that the row and column count algorithm is much faster than the numerical LU factorization algorithm. For the largest matrices, the QR count algorithm takes less than 1% of the LU factorization time. This demonstrates that if static data structures are desirable in LU factorization (e.g., on shared-memory parallel machines), the storage allocation phase based on the QR counts is very fast.

Table 4.3 reports runtimes for the rectangular matrices. Again, the new column elimination tree algorithm is much faster than the one that forms A^TA, with speedups ranging from 3 to 87 fold, and the QR row and column count algorithm is comparable in cost to the column elimination tree algorithm.

5 Conclusion

In this paper we have presented new fast algorithms to compute parameters of the nonzero structure of the QR factorization of a sparse matrix, including the row and column counts of the Householder matrix and the row and column counts of the upper triangular factor. The new algorithms are modifications of an earlier algorithm for computing row and column counts of a Cholesky factor [16]. Our new algorithms require a fast method to compute the column elimination tree without forming A^TA; we give the first detailed description of such a method.

All these algorithms run in time almost linear in $|A|$ rather than in time almost linear in $|A^TA|$. Numerical experiments confirm that the new algorithms are efficient and practical.

REFERENCES

1. 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.* International Journal of Supercomputer Applications, 1(4):10–30, 1987.
2. T. F. Coleman, A. Edenbrandt, and J. R. Gilbert. *Predicting fill for sparse orthogonal factorization. J. ACM*, 33:517–532, 1986.
3. J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. *A supernodal approach to sparse partial pivoting.* SIAM J. Matrix Anal. Appl., 20(3):720–755, 1999.

4. J. W. Demmel, J. R. Gilbert, and X. S. Li. *An asynchronous parallel supernodal algorithm for sparse Gaussian elimination.* SIAM J. Matrix Anal. Appl., 20(4):915–952, 1999.

5. I. S. Duff. *Algorithm 575. Permutations for a zero-free diagonal.* ACM Trans. Math. Software, 7:387–390, 1981.

6. I. S. Duff. *On algorithms for obtaining a maximum transversal.* ACM Trans. Math. Software, 7:315–330, 1981.

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 and T. Wiberg. Implementations of $O(n^{1/2}\tau)$ assignment algorithms. *ACM Trans. Math. Software*, pages 267–287, 1988.

9. A. George, J. R. Gilbert, and J. W. H. Liu. Graph Theory and Sparse Matrix Computation. Springer-Verlag, 1993.

10. A. George and M. T. Heath. *Solution of sparse linear least squares problems using Givens rotations.* Linear Algebra Appl., 34:69–83, 1980.

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

12. A. George, J. W. H. Liu, and E. G. Ng. *A data structure for sparse QR and LU factors.* SIAM J. Sci. Statist. Comput., 9:100–121, 1988.

13. A. George and E. G. Ng. *Symbolic factorization for sparse Gaussian elimination with partial pivoting.* SIAM J. Sci. Statist. Comput., 8:877–898, 1987.

14. J. R. Gilbert, C. Moler, and R. Schreiber. *Sparse matrices in Matlab: Design and implementation.* SIAM J. Matrix Anal. Appl., 13:333–356, 1992.

15. J. R. Gilbert and E. G. Ng. *Predicting structure in nonsymmetric sparse matrix factorizations.* In George et al. [9].

16. J. R. Gilbert, E. G. Ng, and B. W. Peyton. *An efficient algorithm to compute row and column counts for sparse Cholesky factorization.* SIAM J. Matrix Anal. Appl., 15:1075–1091, 1994.

17. D. Hare, C. Johnson, D. Olesky, and P. van den Driessche. *Sparsity analysis of the QR factorization.* SIAM J. Matrix Anal. Appl., 14:665–669, 1993.

18. X. S. Li. *Sparse Gaussian elimination on high performance computers.* Technical Report UCB//CSD-96-919, Computer Science Division, U.C. Berkeley, September 1996. Ph.D. dissertation.

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

20. J. W. H. Liu. *The role of elimination trees in sparse factorization.* Technical Report CS-87-12, Dept. of Computer Science, York University, 1987. This is a longer version of Liu [21].

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

22. J. W. H. Liu, E. G. Ng, and B. W. Peyton. *On finding supernodes for sparse matrix computations.* SIAM J. Matrix Anal. Appl., 14:242–252, 1993.

23. E. G. Ng and B. W. Peyton. *Block sparse Cholesky algorithms on advanced uniprocessor computers.* SIAM J. Sci. Comput., 14:1034–1056, 1993.

24. E. G. Ng and B. W. Peyton. Some results on structure prediction in sparse QR factorization. SIAM J. Matrix Anal. Appl., 17:443–459, 1996.

25. A. Pothen and C. J. Fan. *Computing the block triangular form of a sparse matrix.* ACM Trans. Math. Software, 16:303–324, 1990.

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

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

```
Sort the columns of A by a postorder of T(A^T A);
Compute level(j), the distance from j to the root of j's subtree, for 1 ≤ j ≤ n;
Compute f_i, the column index of the first nonzero in row A_{i*}, for 1 ≤ i ≤ m;
Compute hadj_f[j], for 1 ≤ j ≤ n;
prev_f(j) ← 0, for 1 ≤ j ≤ n;
prev_nbr(j) ← 0, for 1 ≤ j ≤ n;
cc_R(j) ← 1, for 1 ≤ j ≤ n;
w(j) ← 0, for all nonleaves j in T(A^T A);
w(j) ← 1, for all leaves j in T(A^T A);
for column j ← 1 to n do
    if j is not the root of a subtree then
        w(parent(j)) ← w(parent(j)) − 1;
    end if
    for u ∈ hadj_f[j] do
        if fst_desc(j) > prev_nbr(u) then
            /* j is a leaf of the row subtree of u */
            w(j) ← w(j) + 1;
            p_leaf ← prev_f(u);
            if p_leaf = 0 then
                cc_R(u) ← cc_R(u) + level(j) − level(u);
            else
                q ← FIND(p_leaf);
                cc_R(u) ← cc_R(u) + level(j) − level(q);
                w(q) ← w(q) − 1;
            end if
            prev_f(u) ← j;
        end if
        prev_nbr(u) ← j;
    end for
    UNION(j, parent(j));
end for
rc_R(j) ← w(j), for 1 ≤ j ≤ n;
for j ← 1 to n − 1 do
    if j is not the root of a subtree then
        rc_R(parent(j)) ← rc_R(parent(j)) + rc_R(j);
    end if
end for
```

Figure 3.2: The column-oriented algorithm to compute row and column counts of the upper triangular matrix in orthogonal factorization.

Table 4.1: Test matrices.

Square Matrices					
Matrix	n	$\|A\|$	$\|A\|/n$	$\|A^TA\|$	Discipline
MEMPLUS	17,758	99,147	5.6	2,552,314	circuit simulation
GEMAT11	4,929	33,185	6.7	78,676	electrical power
RDIST1	4,134	9,408	2.3	487,160	chemical engineering
ORANI678	2,529	90,158	35.6	1,858,894	economics
MCFE	765	24,382	31.8	144,976	astrophysics
LNSP3937	3,937	25,407	6.5	97,736	fluid flow
LNS3937	3,937	25,407	6.5	97,736	fluid flow
SHERMAN5	3,312	20,793	6.3	86,454	oil reservoir modeling
JPWH991	991	6,027	6.1	24,150	circuit physics
SHERMAN3	5,005	20,033	4.0	54,904	oil reservoir modeling
ORSREG1	2,205	14,133	6.4	43,994	oil reservoir simulation
SAYLR4	3,564	22,316	6.3	70,456	oil reservoir modeling
SHYY161	76,480	329,762	4.3	808,656	fluid flow
GOODWIN	7,320	324,772	44.4	1,768,680	fluid mechanics
VENKAT01	62,424	1,717,792	27.5	4,557,544	flow simulation
INACCURA	16,146	1,015,156	62.9	3,372,106	CFD
AF23560	23,560	460,598	19.6	2,414,716	airfoil simulation
RAEFSKY3	21,200	1,488,768	70.2	4,032,176	fluid turbulence
EX11	16,614	1,096,948	66.0	4,501,260	fluid flow
WANG3	26,064	177,168	6.8	588,958	device simulation
RAEFSKY4	19,779	1,316,789	66.6	5,281,844	buckling problem
AV41092	41,092	1,683,902	41.0	26,961,314	2D PDE
Rectangular Matrices					
Matrix	m	n	$\|A\|$	$\|A^TA\|$	Discipline
CRAN	2,331	1,625	74,902	2,327,086	term-document
MED	5,504	1,063	51,389	765,978	term-document
NPL	11,529	4,322	225,634	1,548,536	term-document
NEWSGROUP	104,260	19,709	1,573,790	167,754,124	term-document
DFL001	12,230	6,071	35,632	76,196	optimization
STOCFOR3	23,541	16,675	72,721	206,720	optimization
OSA-60	243,209	10,243	849,356	510,138	optimization

Table 4.2: Running time in seconds of several algorithms for square matrices on the IBM RS/6000-590: forming A^TA, computing the etree, computing the Cholesky counts of A^TA, computing the QR counts, and sparse LU factorization.

	Based on A^TA			Based on A		
Matrix	Form A^TA	Compute etree	Cholesky counts	Compute etree	QR counts	LU factors
MEMPLUS	1.143	0.599	0.519	0.069	0.106	0.57
GEMAT11	0.091	0.021	0.021	0.023	0.030	0.27
RDIST1	0.887	0.114	0.097	0.059	0.054	0.96
ORANI678	4.948	0.432	0.367	0.057	0.073	1.11
MCFE	0.243	0.041	0.029	0.015	0.015	0.24
LNSP3937	0.073	0.025	0.025	0.019	0.026	1.50
LNS3937	0.073	0.025	0.025	0.018	0.026	1.65
SHERMAN5	0.085	0.021	0.020	0.014	0.019	0.82
JPWH991	0.016	0.006	0.006	0.004	0.006	0.52
SHERMAN3	0.048	0.015	0.017	0.014	0.024	1.37
ORSREG1	0.035	0.011	0.012	0.010	0.014	1.21
SAYLR4	0.055	0.021	0.019	0.017	0.023	2.18
SHYY161	0.699	0.233	0.270	0.241	0.413	25.42
GOODWIN	4.578	0.426	0.348	0.203	0.165	12.55
VENKAT01	11.853	1.083	0.940	1.077	0.926	42.99
INACCURA	15.220	0.784	0.677	0.628	0.506	67.73
AF23560	2.519	0.573	0.495	0.306	0.344	75.91
RAEFSKY3	24.310	0.937	0.800	0.924	0.698	107.60
EX11	17.729	1.045	0.885	0.699	0.557	247.05
WANG3	0.461	0.153	0.144	0.122	0.174	116.58
RAEFSKY4	21.319	1.238	1.040	0.838	0.656	263.13
AV41092	271.354	7.244	5.356	1.051	1.188	786.94

Table 4.3: Running time in seconds of several algorithms for rectangular matrices on the IBM RS/6000-590: forming A^TA, computing the etree, computing the Cholesky counts of A^TA, computing the QR counts.

	Based on A^TA			Based on A	
Matrix	Form A^TA	Compute etree	Cholesky counts	Compute etree	QR counts
CRAN	3.099	0.535	0.453	0.046	0.054
MED	0.495	0.177	0.152	0.030	0.038
NPL	1.626	0.38	0.462	0.149	0.200
NEWSGROUP	206.348	75.731	80.649	3.743	4.982
DFL001	0.064	0.022	0.030	0.023	0.047
STOCFOR3	0.184	0.058	0.086	0.051	0.093
OSA-60	1.516	0.125	0.173	0.492	0.839