A Supernodal Approach to Sparse Partial Pivoting

James W. Demmel* Stanley C. Eisenstat† John R. Gilbert‡
Xiaoye S. Li* Joseph W. H. Liu§

July 10, 1995

Abstract

We investigate several ways to improve the performance of sparse LU factorization with partial pivoting, as used to solve unsymmetric linear systems. To perform most of the numerical computation in dense matrix kernels, we introduce the notion of unsymmetric supernodes. To better exploit the memory hierarchy, we introduce unsymmetric supernode-panel updates and two-dimensional data partitioning. To speed up symbolic factorization, we use Gilbert and Peierls's depth-first search with Eisenstat and Liu's symmetric structural reductions. We have implemented a sparse LU code using all these ideas. We present experiments demonstrating that it is significantly faster than earlier partial pivoting codes. We also compare performance with UMFPACK, which uses a multifrontal approach; our code is usually faster.

Keywords: sparse matrix algorithms; unsymmetric linear systems; supernodes; column elimination tree; partial pivoting.

AMS(MOS) subject classifications: 65F05, 65F50.

Computing Reviews descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra — *Linear systems (direct and iterative methods)*, *Sparse and very large systems.*

*Computer Science Division, University of California, Berkeley, CA 94720 ({demmel,xiaoye}@cs.berkeley.edu). The research of these authors was supported in part by NSF grant ASC–9313958, DOE grant DE–FG03–94ER25219, UT Subcontract No. ORA4466 from ARPA Contract No. DAAL03–91–C0047, DOE grant DE–FG03–94ER25206, and NSF Infrastructure grants CDA–8722788 and CDA–9401156.

†Department of Computer Science, Yale University, P.O. Box 208285, New Haven, CT 06520-8285 (sce@cs.yale.edu). The research of this author was supported in part by NSF grant CCR-9400921.

‡Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304 (gilbert@xerox.com). The research of this author was supported in part by the Institute for Mathematics and Its Applications at the University of Minnesota.

§Department of Computer Science, York University, North York, Ontario, Canada M3J 1P3 (joseph@cs.yorku.ca). The research of this author was supported in part by the Natural Sciences and Engineering Research Council of Canada under grant A5509.

for column $j = 1$ **to** n **do**
    $f = A(:,j)$;
    Symbolic factor: determine which columns of L will update f;
    for each updating column $r < j$ in topological order **do**
        Col-col update: $f = f - f(r) \cdot L(:,r)$;
    end for;
    Pivot: interchange $f(j)$ and $f(k)$, where $|f(k)| = \max |f(j:n)|$;
    Separate L and U: $U(1:j,j) = f(1:j)$; $L(j:n,j) = f(j:n)$;
    Divide: $L(:,j) = L(:,j)/L(j,j)$;
    Prune symbolic structure based on column j;
end for;

Figure 1: LU factorization with column-column updates.

1 Introduction

The problem of solving sparse symmetric positive definite systems of linear equations on sequential and vector processors is fairly well understood. Normally, the solution process is broken into two phases:

- First, symbolic factorization to determine the nonzero structure of the Cholesky factor;
- Second, numeric factorization and solution.

Elimination trees [24] and compressed subscripts [30] reduce the time and space for symbolic factorization to a low-order term. Supernodal [5] and multifrontal [11] elimination allow the use of dense vector operations for nearly all of the floating-point computation, thus reducing the symbolic overhead in numeric factorization to a low-order term. Overall, the megaflop rates of modern sparse Cholesky codes are nearly comparable to those of dense solvers [26].

For unsymmetric systems, where pivoting is required to maintain numerical stability, progress has been less satisfactory. Recent research has concentrated on two basic approaches: submatrix-based methods and column-based (or row-based) methods. Submatrix methods typically use some form of Markowitz pivoting, in which each stage's pivot element is chosen from the uneliminated submatrix by criteria that attempt to balance numerical quality and preservation of sparsity. Recent submatrix codes include MA48 from the Harwell subroutine library [10], and Davis and Duff's unsymmetric multifrontal code UMFPACK [6].

Column methods, by contrast, typically use ordinary partial pivoting.[1] The pivot is chosen from the current column according to numerical considerations alone; the columns may be preordered before factorization to preserve sparsity. Figure 1 sketches a generic left-looking column LU factorization. Notice that the bulk of the numeric computation occurs in column-column updates, or, to use BLAS terminology [8], in sparse AXPYs.

Column methods have the advantage that the preordering for sparsity is completely separate from the factorization, just as in the symmetric case. However, symbolic factorization cannot be

[1]Row methods are exactly analogous to column methods, and codes of both sorts exist. We will use column terminology; those who prefer rows may interchange the terms throughout the paper.

separated from numeric factorization, because the nonzero structures of the factors depend on the numerical pivoting choices. Thus column codes must do some symbolic factorization at each stage; typically this amounts to predicting the structure of each column of the factors immediately before computing it. (George and Ng [14, 15] described ways to obtain upper bounds on the structure of the factors based only on the nonzero structure of the original matrix.)

An early example of such a code is Sherman's NSPIV [31] (which is actually a row code). Gilbert and Peierls [20] showed how to use depth-first search and topological ordering to get the structure of each factor column. This gives a column code that runs in total time proportional to the number of nonzero floating-point operations, unlike the other partial pivoting codes. Eisenstat and Liu [13] designed a pruning technique to reduce the amount of structural information required for the symbolic factorization, as we describe further in Section 4. The result was that the time and space for symbolic factorization were in practice reduced to a low order term.

In view of the success of supernodal techniques for symmetric matrices, it is natural to consider the use of supernodes to enhance the performance of unsymmetric solvers. One difficulty is that, unlike the symmetric case, supernodal structure cannot be determined in advance but rather emerges depending on pivoting choices during the factorization.

In this paper, we generalize supernodes to unsymmetric matrices, and we give efficient algorithms for locating and using unsymmetric supernodes during a column-based LU factorization. We describe a new code called SUPERLU that uses depth-first search and symmetric pruning to speed up symbolic factorization, and uses unsymmetric supernodes to speed up numeric factorization. The rest of the paper is organized as follows. Section 2 introduces the tools we use: unsymmetric supernodes, panels, and the column elimination tree. Section 3 describes the supernodal numeric factorization. Section 4 describes the supernodal symbolic factorization. In Section 5, we present experimental results: we benchmark our code on several test matrices, we compare its performance to other column and submatrix codes, and we investigate its cache behavior in some detail. Finally, Section 6 presents conclusions and open questions.

2 Unsymmetric supernodes

The idea of a supernode is to group together columns with the same nonzero structure, so they can be treated as a dense matrix for storage and computation. Supernodes were originally used for (symmetric) sparse Cholesky factorization; the first published results are by Ashcraft, Grimes, Lewis, Peyton, and Simon [5]. In the factorization $A = LL^T$, a supernode is a range $(r\colon s)$ of columns of L with the same nonzero structure below the diagonal; that is, $L(r\colon s, r\colon s)$ is full lower triangular and every row of $L(s\colon n, r\colon s)$ is either full or zero.[2] (Columns of Cholesky supernodes need not be contiguous, but we will consider only contiguous supernodes.)

Ng and Peyton [26] analyzed the effect of supernodes in Cholesky factorization on modern uniprocessor machines with memory hierarchies and vector or superscalar hardware. All the updates from columns of a supernode are summed into a dense vector before the sparse update is performed. This reduces indirect addressing, and allows the inner loops to be unrolled. In effect, a sequence of column-column updates is replaced by a supernode-column update. The sup-col update can be implemented using a call to a standard dense BLAS-2 matrix-vector multiplication kernel. This idea can be further extended to supernode-supernode updates, which can be implemented using a BLAS-3 dense matrix-matrix kernel. This can reduce memory traffic by an order of magnitude,

[2]We use Matlab notation for integer ranges and submatrices: $r\colon s$ or $(r\colon s)$ is the vector of integers $(r, r+1, \ldots, s)$. If I and J are vectors of integers, then $A(I,J)$ is the submatrix of A with rows whose indices are from I and with columns whose indices are from J. $A(:,J)$ abbreviates $A(1:n,J)$. $nnz(A)$ denotes the number of nonzeros in A.

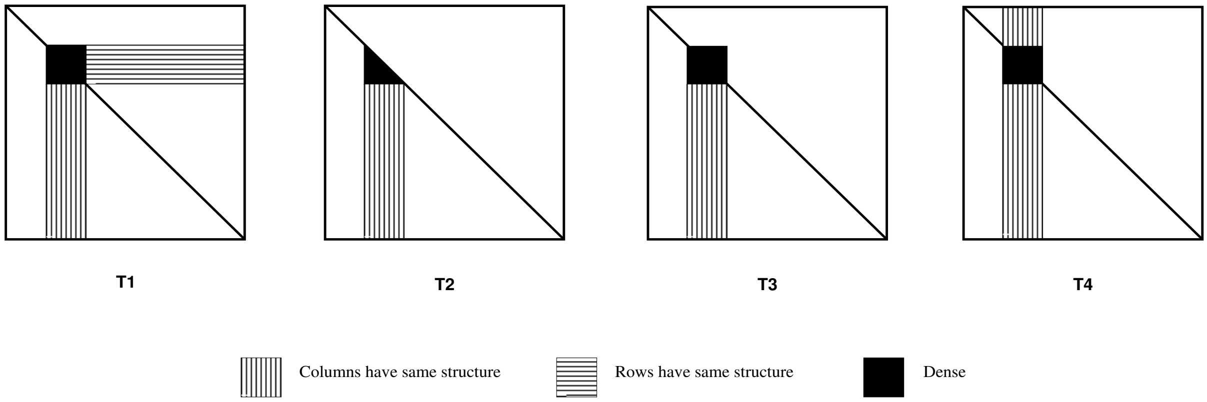


Figure 2: Four possible types of unsymmetric supernodes.

because a supernode in the cache can participate in multiple column updates. Ng and Peyton reported that a sparse Cholesky algorithm based on sup-sup updates typically runs 2.5 to 4.5 times as fast as a col-col algorithm. Indeed, supernodes have become a standard tool in sparse Cholesky factorization [5, 26, 27, 32].

To sum up, supernodes as the source of updates help because:

1. The inner loop (over rows) has no indirect addressing. (Sparse BLAS-1 is replaced by dense BLAS-1.)

2. The outer loop (over columns in the supernode) can be unrolled to save memory references. (BLAS-1 is replaced by BLAS-2.)

Supernodes as the destination of updates help because:

3. Elements of the source supernode can be reused in multiple columns of the destination supernode to reduce cache misses. (BLAS-2 is replaced by BLAS-3.)

Supernodes in sparse Cholesky can be determined during symbolic factorization, before the numeric factorization begins. However, in sparse LU, the nonzero structure cannot be predicted before numeric factorization, so we must identify supernodes on the fly. Furthermore, since the factors L and U are no longer transposes of each other, we must generalize the definition of a supernode.

2.1 Definition of unsymmetric supernodes

We considered several possible ways to generalize the symmetric definition of supernodes to unsymmetric factorization. We define $F = L + U - I$ to be the *filled matrix* containing both L and U.

T1. Same row and column structures: A supernode is a range $(r\colon s)$ of columns of L and rows of U, such that the diagonal block $F(r\colon s, r\colon s)$ is full, and outside that block all the columns of L in the range have the same structure and all the rows of U in the range have the same structure. T1 supernodes make it possible to do sup-sup updates, realizing all three benefits.

	T1	T2	T3	T4
median	0.236	0.345	0.326	0.006
mean	0.284	0.365	0.342	0.052

Table 1: Fraction of nonzeros not in first column of supernode.

T2. Same column structure in L: A supernode is a range $(r\colon s)$ of columns of L with full triangular diagonal block and the same structure below the diagonal block. T2 supernodes allow sup-col updates, realizing the first two benefits.

T3. Same column structure in L, full diagonal block in U: A supernode is a range $(r\colon s)$ of columns of L and U, such that the diagonal block $F(r\colon s, r\colon s)$ is full, and below the diagonal block the columns of L have the same structure. T3 supernodes allow sup-col updates, like T2. In addition, if the storage for a supernode is laid out as for a two-dimensional array (for BLAS-2 or BLAS-3 calls), T3 supernodes do not waste any space in the diagonal block of U.

T4. Same column structure in L and U: A supernode is a range $(r\colon s)$ of columns of L and U with identical structure. (Since the diagonal is nonzero, the diagonal block must be full.) T4 supernodes allow sup-col updates, and also simplify storage of L and U.

T5. Supernodes of A^TA: A supernode is a range $(r\colon s)$ of columns of L corresponding to a Cholesky supernode of the symmetric matrix A^TA. T5 supernodes are motivated by George and Ng's observation [14] that (with suitable representations) the structures of L and U in the unsymmetric factorization $PA = LU$ are contained in the structure of the Cholesky factor of A^TA. In unsymmetric LU, these supernodes themselves are sparse, so we would waste time and space operating on them. Thus we do not consider them further.

Figure 2 is a schematic of definitions T1 through T4.

Supernodes are only useful if they actually occur in practice. The occurrence of symmetric supernodes is related to the clique structure of the chordal graph of the Cholesky factor, which arises because of fill during the factorization. Unsymmetric supernodes seem harder to characterize, but they also are related to dense submatrices arising from fill. We measured the supernodes according to each definition for 126 unsymmetric matrices from the Harwell-Boeing sparse matrix library [9] under various column orderings. Table 1 shows, for each definition, the fraction of nonzeros of L that are not in the first column of a supernode; this measures how much row index storage is saved by using supernodes. Corresponding values for symmetric supernodes for the symmetric Harwell-Boeing structural analysis problems usually range from about 0.5 to 0.9. Larger numbers are better, indicating larger supernodes. We reject T4 supernodes as being too rare to make up for the simplicity of their storage scheme. T1 supernodes allow BLAS-3 updates, but as we will see in Section 3.2 we can get most of their cache advantage with the more common T2 or T3 supernodes by using sup-panel updates. Thus we conclude that either T2 or T3 is the definition of choice. Our code uses T2, which gives slightly larger supernodes than T3 at a small extra cost in storage.

Figure 3 shows a sample matrix, and the nonzero structure of its factors with no pivoting. Using definition T2, this matrix has four supernodes: $\{1,2\}$, $\{3\}$, $\{4,5,6\}$, and $\{7,8,9,10\}$. For example, in columns 4, 5, and 6 the diagonal blocks of L and U are full, and the columns of L all have nonzeros in rows 8 and 9. By definition T3, the matrix has five supernodes: $\{1,2\}$, $\{3\}$, $\{4,5,6\}$, $\{7\}$, and $\{8,9,10\}$. Column 7 fails to join $\{8,9,10\}$ as a T3 supernode because u_{78} is zero.

$$
\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & & & & \\
\bullet & 2 & & \bullet & & & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & \bullet & & & \bullet & \\
& & & \bullet & & & \bullet & & & \\
\bullet & & & & & 6 & & & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & & \bullet & & & & & & & \\
& & & \bullet & & \bullet & & & 9 & \\
& & \bullet & & & & \bullet & & \bullet & 10
\end{pmatrix}
\qquad
\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & & & & \\
\bullet & 2 & \bullet & \bullet & & \bullet & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & \bullet & & & \bullet & \\
& & & \bullet & 5 & \bullet & \bullet & & \bullet & \\
\bullet & \bullet & \bullet & \bullet & \bullet & 6 & \bullet & \bullet & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 8 & \bullet & \bullet \\
& & & \bullet & \bullet & \bullet & \bullet & \bullet & 9 & \bullet \\
& & \bullet & & & & \bullet & \bullet & \bullet & 10
\end{pmatrix}
$$

Original matrix A Factors $F = L + U - I$

Figure 3: A sample matrix and its LU factors. Diagonal elements a_{55} and a_{88} are zero.

2.2 Storage of supernodes

A standard way to lay out storage for a sparse matrix is a one-dimensional array of nonzero values in column major order, plus integer arrays giving row numbers and column starting positions. We use this layout for both L and U, but with a slight modification: we store the entire square diagonal block of each supernode as part of L, including both the strict lower triangle of values from L and the upper triangle of values from U. We store this square block as if it were completely full (it is full in T3 supernodes, but its upper triangle may contain zeros in T2 supernodes). This allows us to address each supernode as a two-dimensional array in calls to BLAS routines. In other words, if columns $(r\colon s)$ form a supernode, then all the nonzeros in $F(r\colon n, r\colon s)$ are stored as a single dense two-dimensional array. This also lets us save some storage for row indices: only the indices of nonzero rows outside the diagonal block need be stored, and the structures of all columns within a supernode can be described by one set of row indices. This is similar to the effect of compressed subscripts in the symmetric case [30].

We represent the part of U outside the supernodal blocks with a standard sparse structure: the values are stored by columns, with a companion integer array the same size to store row indices; another array of n integers indicates the start of each column.

Figure 4 shows the structure of the factors in the example from Figure 3, with s_k denoting a nonzero in the k-th supernode and u_k denoting a nonzero in the k-th column of U outside the supernodal block. Figure 5 shows the storage layout. (We omit the indexing vectors that point to the beginning of each supernode and the beginning of each column of U.)

2.3 The column elimination tree

Since our definition requires the columns of a supernode to be contiguous, we should get larger supernodes if we bring together columns of L with the same nonzero structure. But the column ordering is fixed, for sparsity, before numeric factorization; what can we do?

In symmetric Cholesky factorization, the so-called fundamental supernodes can be made contiguous by permuting the matrix (symmetrically) according to a postorder on its elimination tree [4]. This postorder is an example of what Liu calls an equivalent reordering [24], which does not change the sparsity of the factor. The postordered elimination tree can also be used to locate the super-

$$
\begin{array}{c}
1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10
\end{array}
\left(
\begin{array}{cccccccccc}
s_1 & s_1 & u_3 & & & u_6 & & & & \\
s_1 & s_1 & u_3 & u_4 & & u_6 & & u_8 & & \\
 & & s_2 & & & & u_7 & u_8 & & \\
 & & & s_3 & s_3 & s_3 & & & u_9 & \\
 & & & s_3 & s_3 & s_3 & u_7 & & u_9 & \\
s_1 & s_1 & s_2 & s_3 & s_3 & s_3 & u_7 & u_8 & u_9 & \\
 & & & & & & s_4 & & s_4 & s_4 \\
s_1 & s_1 & s_2 & s_3 & s_3 & s_3 & s_4 & s_4 & s_4 & s_4 \\
 & & & s_3 & s_3 & s_3 & s_4 & s_4 & s_4 & s_4 \\
 & & s_2 & & & & s_4 & s_4 & s_4 & s_4
\end{array}
\right)
$$

Figure 4: Supernodal structure (by definition T2) of the factors of the sample matrix.

$$
\begin{array}{cc}
 & \begin{array}{cc} 1 & 2 \end{array} \\
\begin{array}{c} \\ \\ 6 \\ 8 \end{array} &
\begin{pmatrix} s_1 & s_1 \\ s_1 & s_1 \\ s_1 & s_1 \\ s_1 & s_1 \end{pmatrix}
\end{array}
\quad
\begin{array}{cc}
 & 3 \\
\begin{array}{c} \\ 6 \\ 8 \\ 10 \end{array} &
\begin{pmatrix} s_2 \\ s_2 \\ s_2 \\ s_2 \end{pmatrix}
\end{array}
\quad
\begin{array}{cc}
 & \begin{array}{ccc} 4 & 5 & 6 \end{array} \\
\begin{array}{c} \\ \\ \\ 8 \\ 9 \end{array} &
\begin{pmatrix} s_3 & s_3 & s_3 \\ s_3 & s_3 & s_3 \\ s_3 & s_3 & s_3 \\ s_3 & s_3 & s_3 \\ s_3 & s_3 & s_3 \end{pmatrix}
\end{array}
\quad
\begin{array}{c}
\begin{array}{cccc} 7 & 8 & 9 & 10 \end{array} \\
\begin{pmatrix} s_4 & 0 & s_4 & s_4 \\ s_4 & s_4 & s_4 & s_4 \\ s_4 & s_4 & s_4 & s_4 \\ s_4 & s_4 & s_4 & s_4 \end{pmatrix}
\end{array}
$$

Supernodal blocks (stored in column-major order)

Row Subscripts	1	2	2	1	2	3	5	6	2	3	6	4	5	6
	u_3	u_3	u_4	u_6	u_6	u_7	u_7	u_7	u_8	u_8	u_8	u_9	u_9	u_9

Off-supernodal nonzeros in columns of U

Figure 5: Storage layout for factors of the sample matrix, using T2 supernodes.

nodes before the numeric factorization.

We proceed similarly for the unsymmetric case. Here the appropriate analog of the symmetric elimination tree is the *column elimination tree*, or column etree for short. The vertices of this tree are the integers 1 through n, representing the columns of A. The column etree of A is the (symmetric) elimination tree of the column intersection graph of A, or equivalently the elimination tree of $A^T A$ provided there is no cancellation in computing $A^T A$. See Gilbert and Ng [19] for complete definitions. The column etree can be computed from A in time almost linear in the number of nonzeros of A by a variation of an algorithm of Liu [24].

The following theorem says that the column etree represents potential dependencies among columns in LU factorization, and that (for strong Hall matrices) no stronger information is obtainable from the nonzero structure of A. Note that column i updates column j in LU factorization if and only if $u_{ij} \neq 0$.

Theorem 1 (Column Elimination Tree) [19] *Let A be a square, nonsingular, possibly unsymmetric matrix, and let $PA = LU$ be any factorization of A with pivoting by row interchanges. Let T be the column elimination tree of A.*

1. *If vertex i is an ancestor of vertex j in T, then $i \geq j$.*
2. *If $l_{ij} \neq 0$, then vertex i is an ancestor of vertex j in T.*
3. *If $u_{ij} \neq 0$, then vertex j is an ancestor of vertex i in T.*
4. *Suppose in addition that A is strong Hall (that is, A cannot be permuted to a nontrivial block triangular form). If vertex j is the parent of vertex i in T, then there is some choice of values for the nonzeros of A that makes $u_{ij} \neq 0$ when the factorization $PA = LU$ is computed with partial pivoting.*

Just as a postorder on the symmetric elimination tree brings together symmetric supernodes, we expect a postorder on the column etree to bring together unsymmetric supernodes. Thus, before we factor the matrix, we compute its column etree and permute the matrix columns according to a postorder on the tree. We now show that this does not change the factorization in any essential way.

Theorem 2 *Let A be a matrix with column etree T. Let π be a permutation such that whenever $\pi(i)$ is an ancestor of $\pi(j)$ in T, we have $i \geq j$. Let P be the permutation matrix such that $\pi = P \cdot (1{:}\, n)^T$. Let $\bar{A} = PAP^T$.*

1. *$\bar{A} = A(\pi, \pi)$.*
2. *The column etree $\bar{T}$ of $\bar{A}$ is isomorphic to T; in particular, relabeling each node i of $\bar{T}$ as $\pi(i)$ yields T.*
3. *Suppose in addition that $\bar{A}$ has an LU factorization without pivoting, $\bar{A} = \bar{L}\bar{U}$. Then $P^T \bar{L} P$ and $P^T \bar{U} P$ are respectively unit lower triangular and upper triangular, so $A = (P^T \bar{L} P)(P^T \bar{U} P)$ is also an LU factorization.*

Remark: Liu [24] attributes to F. Peters a result similar to part (3) for the symmetric positive definite case, concerning the Cholesky factor and the (usual, symmetric) elimination tree.

Proof: Part (1) is immediate from the definition of P. Part (2) follows from Corollary 6.2 in Liu [24], with the symmetric structure of the column intersection graph of our matrix A taking the place of Liu's symmetric matrix A. (Liu exhibits the isomorphism explicitly in the proof of his Theorem 6.1.)

Now we prove part (3). We have $a_{\pi(i)\pi(j)} = \bar{a}_{ij}$ for all i and j. Write $L = P^T \bar{L} P$ and $U = P^T \bar{U} P$, so that $l_{\pi(i)\pi(j)} = \bar{l}_{ij}$ and $u_{\pi(i)\pi(j)} = \bar{u}_{ij}$. Then $A = LU$; we need only show that L and U are triangular.

Consider a nonzero $u_{\pi(i)\pi(j)}$ of U. In the triangular factorization $\bar{A} = \bar{L}\bar{U}$, element $\bar{u}_{ij}$ is equal to $u_{\pi(i)\pi(j)}$ and is therefore nonzero. By part (3) of Theorem 1, then, j is an ancestor of i in $\bar{T}$. By the isomorphism between $\bar{T}$ and T, this implies that $\pi(j)$ is an ancestor of $\pi(i)$ in T. Then it follows from part (1) of Theorem 1 that $\pi(j) \geq \pi(i)$. Thus every nonzero of U is on or above the diagonal, so U is upper triangular. A similar argument shows that every nonzero of L is on or below the diagonal, so L is lower triangular. The diagonal elements of L are a permutation of those of $\bar{L}$, so they are all equal to 1. □

Since the triangular factors of A are just permutations of the triangular factors of PAP^T, they have the same sparsity. Indeed, they require the same arithmetic to compute; the only possible difference is the order of updates. If addition for updates is commutative and associative, this implies that with partial pivoting (i, j) is a legal pivot in $\bar{A}$ iff $(\pi(i), \pi(j))$ is a legal pivot in A. In floating-point arithmetic, the different order of updates could conceivably change the pivot sequence. Thus we have the following corollary.

Corollary 1 *Let π be a postorder on the column elimination tree of A, let P_1 be any permutation matrix, and let P_2 be the permutation matrix with $\pi = P_2 \cdot (1{:}\,n)^T$. If $P_1 A P_2^T = LU$ is an LU factorization, then so is $(P_2^T P_1)A = (P_2^T L P_2)(P_2^T U P_2)$. In exact arithmetic, the former is an LU factorization with partial pivoting of AP_2^T if and only if the latter is an LU factorization with partial pivoting of A.*

This corollary says that an LU code can permute the columns of its input matrix by postorder on the column etree, and then fold the column permutation into the row permutation on output. Thus our SuperLU code has the option of returning either four matrices P_1, P_2, L, and U (with $P_1 A P_2^T = LU$), or just the three matrices $P_2^T P_1$, $P_2^T L P_2$, and $P_2^T U P_2$, which are a row permutation and two triangular matrices. The advantage of returning all four matrices is that the columns of each supernode are contiguous in L, which permits the use of a Blas-2 supernodal triangular solve for the forward-substitution phase of a linear system solver. The supernodes are not contiguous in $P_2^T L P_2$.

2.4 Relaxed supernodes

We have explored various ways of relaxing the denseness of a supernode. We experimented with both T2 and T3 supernodes, and found that T2 supernodes (those with only nested column structures in L) give slightly better performance.

We observe that, for most matrices, the average size of a supernode is only about 2 to 3 columns (though a few supernodes are much larger). A large percentage of supernodes consist of only a single column, many of which are leaves of the column etree. Therefore we have devised a scheme to merge groups of columns at the fringe of the etree into *artificial supernodes* regardless of their row structures. A parameter r controls the granularity of the merge. Our merge rule is: node i is merged with its parent node j when the subtree rooted at j has at most r nodes. In practice,

for column $j = 1$ **to** n **do**
    $f = A(:, j)$;
    Symbolic factor: determine which supernodes of L will update f;
    Determine whether j belongs to the same supernode as $j-1$;
    for each updating supernode $(r\colon s) < j$ in topological order **do**
        Apply supernode-column update to column j:
            $f(r\colon s) = L(r\colon s, r\colon s)^{-1} \cdot f(r\colon s)$;
            $f(s+1\colon n) = f(s+1\colon n) - L(s+1\colon n, r\colon s) \cdot f(r\colon s)$;
    end for;
    Pivot: interchange $f(j)$ and $f(k)$, where $|f(k)| = \max |f(j\colon n)|$;
    Separate L and U: $U(1\colon j, j) = f(1\colon j)$; $L(j\colon n, j) = f(j\colon n)$;
    Divide: $L(:, j) = L(:, j)/L(j, j)$;
    Prune symbolic structure based on column j;
end for;

Figure 6: LU factorization with supernode-column updates.

the best values of r are generally between 4 and 8, and yield improvements in running time of 5% to 15%.

Artificial supernodes are a special case of *relaxed supernodes*, which Ashcraft and Grimes have used in the context of multifrontal methods for symmetric systems [4]. Ashcraft and Grimes allow a small number of nonzeros in the structure of any supernode, thus relaxing the condition that the columns must have strictly nested structures. It would be possible to use this idea in the unsymmetric code as well, though we have not experimented with it. Relaxed supernodes could be constructed either on the fly (by relaxing the nonzero count condition described in Section 4.3 below), or by preprocessing the column etree to identify small subtrees that we would merge into supernodes.

3 Supernodal numeric factorization

Now we show how to modify the column-column algorithm to use supernode-column updates and supernode-panel updates. This section describes the numerical computation involved in the updates. Section 4 describes the symbolic factorization that determines which supernodes update which columns, and also the detection of boundaries between supernodes.

3.1 Supernode-column updates

Figure 6 sketches the supernode-column algorithm. The only difference from the column-column algorithm is that all the updates to a column from a single supernode are done together. Consider a supernode $(r\colon s)$ that updates column j. The coefficients of the updates are the values from a segment of column j of U, namely $U(r\colon s, j)$. The nonzero structure of such a segment is particularly simple: all the nonzeros are contiguous, and follow all the zeros (as proved in Corollary 2 below). Thus, if k is the index of the first nonzero row in $U(r\colon s, j)$, the updates to column j from supernode $(r\colon s)$ come from columns k through s. Since the supernode is stored as a dense matrix, these updates can be performed by a dense lower triangular solve (with the matrix $L(k\colon s, k\colon s)$) and a dense matrix-vector multiplication (with the matrix $L(s+1\colon n, k\colon s)$). As described in Section 4, the symbolic phase determines the value of k, that is, the position of the first nonzero in the segment $U(r\colon s, j)$.

The advantages of using supernode-column updates are similar to those in the symmetric case [26]. Efficient BLAS-2 matrix-vector kernels can be used for the triangular solve and matrix-vector multiply. Furthermore, all the updates from the supernodal columns can be collected in a dense vector before doing a single scatter into the target vector. This reduces the amount of indirect addressing.

3.2 Supernode-panel updates

We can improve the supernode-column algorithm further on machines with a memory hierarchy by changing the data access pattern. The data we are accessing in the inner loop (lines 5–9 of Figure 6) include the destination column j and all the updating supernodes $(r\colon s)$ to the left of column j. Column j is accessed many times, while each supernode $(r\colon s)$ is used only once. In practice, the number of nonzero elements in column j is much less than that in the updating supernodes. Therefore, the access pattern given by this loop provides little opportunity to reuse cached data. In particular, the same supernode $(r\colon s)$ may be needed to update both columns j and $j+1$. But when we factor the $(j+1)$-st column (in the next iteration of the outer loop), we will have to fetch supernode $(r\colon s)$ again from memory, instead of from cache (unless the supernodes are small compared to the cache).

To exploit memory locality, we factor several columns (say w of them) at a time in the outer loop, so that one updating supernode $(r\colon s)$ can be used to update as many of the w columns as possible. We refer to these w consecutive columns as a *panel*, to differentiate them from a supernode; the row structures of these columns may not be correlated in any fashion, and the boundaries between panels may be different from those between supernodes. The new method requires rewriting the doubly nested loop as the triple loop shown in Figure 7. This is analogous to loop tiling techniques used in optimizing compilers to improve cache behavior for two-dimensional arrays with regular stride. It is also somewhat analogous to the supernode-supernode updates that Ng and Peyton [26], and Rothberg and Gupta [27] have used in symmetric Cholesky factorization.

The structure of each supernode-column update is the same as in the supernode-column algorithm. For each supernode $(r\colon s)$ to the left of column j, if $u_{kj} \neq 0$ for some $r \leq k \leq s$, then $u_{ij} \neq 0$ for all $k \leq i \leq s$. Therefore, the nonzero structure of the panel of U consists of dense column segments that are row-wise separated by supernodal boundaries, as in Figure 7. Thus, it is sufficient for the symbolic factorization algorithm to record only the first nonzero position of each column segment. As detailed in Section 4.4, symbolic factorization is applied to all the columns in a panel at once, outside the numeric-factorization loop over updating supernodes.

In dense factorization, the entire supernode-panel update in lines 3–7 of Figure 7 would be implemented as two BLAS-3 calls: a dense triangular solve with w right-hand sides, followed by a dense matrix-matrix multiply. In the sparse case, this is not possible, because the different supernode-column updates begin at different positions k within the supernode, and the submatrix $U(r\colon s, j\colon j+w-1)$ is not dense. Thus the sparse supernode-panel algorithm still calls the level-2 BLAS. However, we get the similar cache benefits as the level-3 BLAS, at the cost of doing the loop reorganization ourselves. Thus we sometimes call the kernel of this algorithm a "BLAS-$2\frac{1}{2}$" method.

In the double loop nest (3–7), the ideal circumstance is that all w columns in the panel require

for column $j = 1$ **to** n **step** w **do**
Symbolic factor: determine which supernodes will update any of $L(:, j: j + w - 1)$;
for each updating supernode $(r: s) < j$ in topological order **do**
for column $jj = j$ **to** $j + w - 1$ **do**
Apply supernode-column update to column jj;
end for;
end for;
Inner factorization: apply the sup-col algorithm to the panel;
end for;

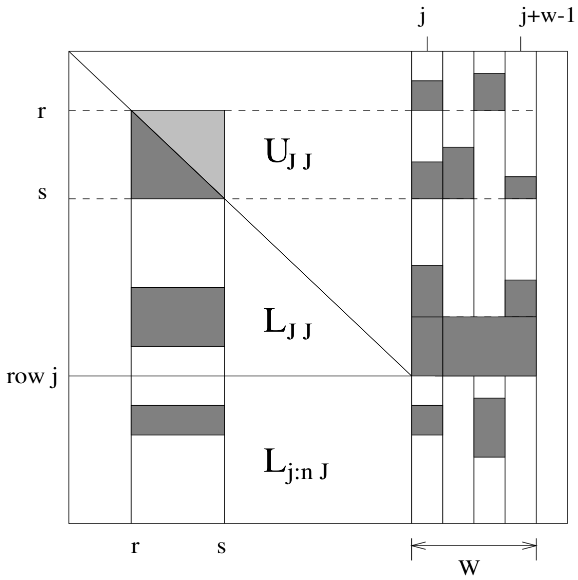


Figure 7: The supernode-panel algorithm, with column-wise blocking. $J = 1 : j - 1$.

updates from supernode $(r\colon s)$. Then this supernode will be used w times before it is forced out of the cache. There is a trade-off between the value of w and the size of cache. For this scheme to work efficiently, we need to ensure that the nonzeros in the w columns do not cause cache thrashing. That is, we must keep w small enough that all the data accessed in this doubly nested loop fit in cache. Otherwise, the cache cross-interference between the source supernode and the destination panel can offset the benefit of the new algorithm.

3.2.1 Outer and inner factorization

At the end of the supernode-panel update (line 7), columns j through $j+w-1$ of L and U have received all their updates from columns to the left of j. We call this the *outer factorization.* What remains is to apply updates that come from columns within the panel. This amounts to forming the LU factorization of the panel itself (in columns $(j\colon j+w-1)$, and rows $(j\colon n)$). This *inner factorization* is performed by the supernode-column algorithm, almost exactly as shown in Figure 6. The inner factorization includes a columnwise symbolic factorization just as in the supernode-column algorithm. The inner factorization also includes the supernode identification, partial pivoting, and symmetric structure reduction for the entire algorithm.

3.2.2 Reducing cache misses by row-wise blocking

Our first experiments with the supernode-panel algorithm showed speedups for some medium-sized problems of around 20–30%. However, the improvement for large matrices was often only a few percent. We now study the reasons and remedies for this.

To implement loops (3–7) of Figure 7, we first expand the nonzeros of the panel columns of A into an n by w full working array, called the SPA (for *sparse accumulator* [18]). This allows random access to the entries of the active panel. A temporary array stores the results of the BLAS operations, and the updates are scattered into the SPA. At the end of panel factorization, the data in the SPA are copied into storage for L and U. Although increasing the panel size w gives more opportunity for data reuse, it also increases the size of the active data set that must fit into cache. The supernode-panel update loop accesses the following data:

- the nonzeros in the updating supernode $L(r\colon n, r\colon s)$.
- the SPA data structure, consisting of an n by w full array and a temporary store of size n.

By instrumenting the code, we found that the working sets of large matrices are much larger than the cache size. Hence, cache thrashing limits performance.

We experimented with a scheme suggested by Rothberg [28], in which the SPA has only as many rows as the number of nonzero rows in the panel (as predicted by symbolic factorization), and an extra indirection array of size n is used to address the SPA. Unfortunately, the cost incurred by double indirection is not negligible, and this scheme was not as effective as the two-dimensional blocking method we now describe.

We implemented a row-wise blocking scheme on top of the column-wise blocking in the supernode-panel update. The 2-D blocking adds another level of looping between the two loops in lines 3 and 4 of Figure 7. This partitions the supernodes (and the SPA structure) into block rows. Then each block row of the updating supernode is used for up to w partial matrix-vector multiplies, which are pushed all the way through into the SPA before the next block row of the supernode is accessed. The active data set accessed in the inner loops is thus much smaller than in the 1-D scheme. The 2-D blocking algorithm is organized as in Figure 8. The key performance gains come from

```
1.  for j = 1 to n step w do
2.      ...
3.      for each updating supernode (r: s) < j in topological order do
4.          Apply triangular solves to A(r: s, j: j + w − 1) using L(r: s, r: s);
5.          for each row block B in L(s + 1: n, r: s) do
6.              for jj = j to j + w − 1 do
7.                  Multiply B · U(r: s, jj), and scatter into SPA(: , jj);
8.              end for;
9.          end for;
10.     end for;
11.     ...
12  end for;
```

Figure 8: The supernode-panel algorithm, with 2-D blocking.

the loops (5–9), where each row block is reused as much as possible before the next row block is brought into the cache. The innermost loop is still a dense-matrix vector multiply, performed by a BLAS-2 kernel.

3.2.3 Combining 1-D and 2-D blocking

The 2-D blocking works well when the rectangular supernodal matrix $L(r\colon n, r\colon s)$ is large in both dimensions. If all of $L(r\colon n, r\colon s)$ can fit into cache, then the row-wise blocking gives no benefit, but still incurs overhead for setting up loop variables, skipping the empty loop body, and so on. This overhead can be nearly 10% for some of the sparser problems in our test suite. Thus we have devised a hybrid update algorithm that uses either the 1-D or 2-D partitioning scheme, depending on the size of each updating supernode. The decision is made at run-time, as shown in Figure 9. The overhead is limited to the test at line 4 of Figure 9. It turns out that this hybrid scheme works better than either 1-D or 2-D codes for many problems. Therefore, this is the algorithm that we use in the performance analysis in Section 5. In Section 5.5.3 we will discuss in more detail what we mean by "large" in the test on line 4.

4 Symbolic factorization

Symbolic factorization is the process that determines the nonzero structure of the triangular factors L and U from the nonzero structure of the matrix A. This in turn determines which columns of L update each column j of the factors (namely, those columns r for which $u_{rj} \neq 0$), and also which columns of L can be combined into supernodes.

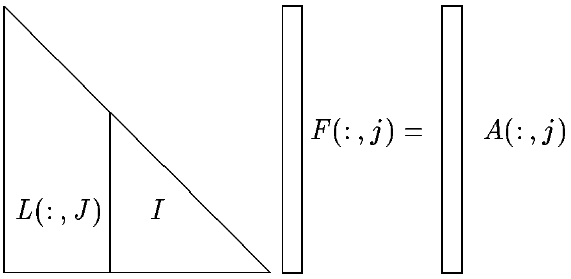
Without numeric pivoting, the complete symbolic factorization can be performed before any numeric factorization. Partial pivoting, however, requires that the numeric and symbolic factorizations be interleaved. The supernode-column algorithm performs symbolic factorization for each column just before it is computed, as described in Section 4.1. The supernode-panel algorithm performs symbolic factorization for an entire panel at once, as described in Section 4.4.

```
for j = 1 to n step w do
    ...
    for each updating supernode (r: s) < j in topological order do
        if (r: s) is large then /* use 2-D blocking */
            Apply triangular solves to A(r: s, j: j + w − 1) using L(r: s, r: s);
            for each row block B in L(s + 1: n, r: s) do
                for jj = j to j + w − 1 do
                    Multiply B · U(r: s, jj), and scatter into SPA(: , jj);
                end for;
            end for;
        else /* use 1-D blocking */
            for jj = j to j + w − 1 do
                Apply triangular solve to A(r: s, jj);
                Multiply L(s + 1: n, r: s) · U(r: s, jj), and scatter into SPA(: , jj);
            end for;
        end if;
    end for;
    ...
end for;
```

Figure 9: The supernode-panel algorithm, with both 1-D and 2-D blocking.

4.1 Column depth-first search

From the numeric factorization algorithm, it is clear that the structure of column $F(:,j)$ depends on the structure of column $A(:,j)$ of the original matrix and on the structure of $L(:,J)$, where $J = 1{:}j{-}1$. Indeed, $F(:,j)$ has the same structure as the solution vector for the following triangular system [20]:



A straightforward way to compute the structure of $F(:,j)$ from the structures of $L(:,J)$ and $A(:,j)$ is to simulate the numerical computation. A less expensive way is to use the following characterization in terms of paths in the directed graph of $L(:,J)$.

For any matrix M, the notation $i \xrightarrow{M} j$ means that there is an edge from i to j in the directed graph of M, that is, $m_{ij} \neq 0$. The notation $i \xRightarrow{M} j$ means that there is a directed path from i to j in the directed graph of M. Such a path may have length zero; that is, $i \xRightarrow{M} i$ always holds.

Theorem 3 [16] *f_{ij} is nonzero (equivalently, $i \xrightarrow{F} j$) if and only if $i \xRightarrow{L(:,J)} k \xrightarrow{A} j$ for some $k \le i$.*

This result implies that the symbolic factorization of column j can be obtained as follows. Consider the nonzeros in $A(:,j)$ as a subset of the vertices of the directed graph $G = G(L(:,J)^T)$, which is the reverse of the directed graph of $L(:,J)$. The nonzero positions of $F(:,j)$ are then given by the vertices reachable by paths from this set in G. We use the graph of L^T here because of the convention that edges are directed from rows to columns. Since L is actually stored by columns, our data structure gives precisely the adjacency information for G. Therefore, we can determine the structure of column j of L and U by traversing G from the set of starting nodes given by the structure of $A(:,j)$.

The traversal of G determines the structure of $U(:,j)$, which in turn determines the columns of L that will participate in updates to column j in the numerical factorization. These updates must be applied in an order consistent with a topological ordering of G. We use depth-first search to perform the traversal, which makes it possible to generate a topological order (specifically, reverse postorder) on the nonzeros of $U(:,j)$ as they are located [20].

Another consequence of the path theorem is the following corollary. It says that if we divide each column of U into *segments*, one per supernode, then within each segment the column of U just consists of a consecutive sequence of nonzeros. Thus we need only keep track of the position of the first nonzero in each segment.

Corollary 2 *Let $(r:s)$ be a supernode (of either type T2 or T3) in the factorization $PA = LU$. Suppose u_{kj} is nonzero for some j with $r \le k \le s$. Then $u_{ij} \ne 0$ for all i with $k \le i \le s$.*

Proof: Let $k \le i \le s$. Since $u_{kj} \ne 0$, we have $k \overset{L(:,J)}{\Longrightarrow} \overset{A}{\longrightarrow} j$ by Theorem 3. Now l_{ik} is in the diagonal block of the supernode, and hence is nonzero. Thus $i \overset{L(:,J)}{\longrightarrow} k$, so $i \overset{L(:,J)}{\Longrightarrow} \overset{A}{\longrightarrow} j$, whence u_{ij} is nonzero by Theorem 3. □

4.2 Pruning the symbolic structure

We can speed up the depth-first search traversals by using a reduced graph in place of G, the reverse of the graph of $L(:,J)$. We have explored this idea in a series of papers [12, 13, 17]. Any graph H can be substituted for G, provided that $i \overset{H}{\Longrightarrow} j$ if and only if $i \overset{G}{\Longrightarrow} j$. The traversals are more efficient if H has fewer edges; but any gain in efficiency must be traded off against the cost of computing H.

An extreme choice of H is the *elimination dag* [17], which is the transitive reduction of G, or the minimal subgraph of G that preserves paths. However, the elimination dag is expensive to compute. The *symmetric reduction* [12] is a subgraph that has (in general) fewer edges than G but more edges than the elimination dag, and that is much less expensive to compute. The symmetric reduction takes advantage of symmetry in the structure of the filled matrix F; if F is completely symmetric, it is just the symmetric elimination tree. The symmetric reduction of $L(:,J)$ is obtained by removing all nonzeros l_{rs} for which $l_{ts}u_{st} \ne 0$ for some $t < \min(r,j)$. Eisenstat and Liu [13] give an efficient method to compute the symmetric reduction during symbolic factorization, and demonstrate experimentally that it significantly reduces the total factorization time with an algorithm that does column-column updates.

Our supernodal code uses symmetric reduction to speed up its symbolic factorization. The example in Figure 10 illustrates symmetric reduction in the presence of supernodes. We use S to represent the supernodal structure of $L(:,J)^T$, and R to represent the symmetric reduction of S. It is this R that we use in our code. Note that the edges of the graph of R are directed from columns of L to rows of L.

$$
\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & \\
\bullet & 2 & \bullet & \bullet & & \bullet & \\
 & & 3 & & & & \bullet \\
 & & & 4 & \bullet & \bullet & \\
 & & & \bullet & 5 & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & 6 & \bullet \\
 & & & & & & 7 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
 & & & \bullet & \bullet & \bullet & \bullet \\
 & & \bullet & & & & \bullet
\end{pmatrix}
\quad
\begin{pmatrix}
1 & & & & & & \\
\bullet & 2 & & & & & \\
 & & 3 & & & & \\
 & & & 4 & & & \\
 & & & \bullet & 5 & & \\
\bullet & \bullet & \bullet & \bullet & \bullet & 6 & \\
 & & & & & & 7 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
 & & & \bullet & \bullet & \bullet & \bullet \\
 & & \bullet & & & & \bullet
\end{pmatrix}
\quad
\begin{pmatrix}
1 & & & \\
2 & & & \\
 & 3 & & \\
 & & 4 & \\
 & & 5 & \\
\bullet & \bullet & 6 & \\
 & & & 7 \\
\bullet & \bullet & \bullet & \bullet \\
 & & \bullet & \bullet \\
 & \bullet & & \bullet
\end{pmatrix}
\quad
\begin{pmatrix}
1 & & & \\
2 & & & \\
 & 3 & & \\
 & & 4 & \\
 & & 5 & \\
\bullet & \bullet & 6 & \\
 & & & 7 \\
\cdot & \bullet & \bullet & \bullet \\
 & & \bullet & \bullet \\
 & \bullet & & \bullet
\end{pmatrix}
$$

Filled matrix $F(:,J)$ | Lower triangle G | Supernodal S | Reduced R

Figure 10: Supernodal and symmetrically reduced structures.

$$
\begin{pmatrix}
1 & & & \\
2 & & & \\
 & 3 & & \\
 & & 4 & \\
 & & 5 & \\
\bullet & \bullet & 6 & \\
 & & & 7 \\
\cdot & \bullet & \bullet & \bullet \\
 & & \bullet & \bullet \\
 & \bullet & & \bullet
\end{pmatrix}
\begin{pmatrix}
 \\ \bullet \\ \bullet \\ \\ \\ \circ \\ \\ \circ \\ \circ \\ \circ
\end{pmatrix}
\Longrightarrow
\begin{pmatrix}
1 & & & & \\
2 & & & & \\
 & 3 & & & \\
 & & 4 & & \\
 & & 5 & & \\
\bullet & \bullet & 6 & & \\
 & & & 7 & \\
\cdot & \bullet & \bullet & \bullet & 8 \\
 & & \cdot & \bullet & \bullet \\
 & \cdot & & \bullet & \bullet
\end{pmatrix}
$$

Figure 11: One step of symbolic factorization in the reduced structure.

In the figure, the small dot "$\cdot$" indicates an entry in S that was pruned from R by symmetric reduction. The $(8,2)$ entry was pruned due to the symmetric nonzero pair $(6,2)$ and $(2,6)$. The figure shows the current state of the reduced structure based on the first seven columns of the filled matrix.

It is instructive to follow this example through one more column to see how symbolic factorization is carried out in the reduced supernodal structure. Consider the symbolic step for column 8. Suppose a_{28} and a_{38} are nonzero. The other nonzeros in column 8 of the factor are generated by paths in the reduced supernodal structure (we just show one possible path for each nonzero):

$$8 \stackrel{A^T}{\rightarrow} 2 \stackrel{R}{\rightarrow} 6,$$

$$8 \stackrel{A^T}{\rightarrow} 3 \stackrel{R}{\rightarrow} 8,$$

$$8 \stackrel{A^T}{\rightarrow} 2 \stackrel{R}{\rightarrow} 6 \stackrel{R}{\rightarrow} 9,$$

$$8 \stackrel{A^T}{\rightarrow} 3 \stackrel{R}{\rightarrow} 10,$$

Figure 11 shows the reduced supernodal structure before and after column 8. In column 8 of A, the original nonzeros are shown as "$\bullet$" and the fill nonzeros are shown as "$\circ$". Once the structure of column 8 of U is known, more symmetric reduction is possible. The entry $l_{10,3}$ is pruned due to the symmetric nonzeros in l_{83} and u_{38}. Also, l_{96} is pruned by l_{86} and u_{68}. Figure 11 shows the new structure.

The supernodal symbolic factorization relies on the path characterization in Theorem 3 and on the path-preserving property of symmetric reduction. In effect, we use the modified path condition

$$i \xrightarrow{A^T} \overset{R}{\Longrightarrow} j$$

on the symmetrically-reduced supernodal structure R of $L(:,J)^T$.

4.3 Detecting supernodes

Since supernodes consist of contiguous columns of L, we can decide at the end of each symbolic factorization step whether the new column j belongs to the same supernode as column $j-1$.

For T2 supernodes, the test is straightforward. During symbolic factorization, we test whether $L(:,j) \subseteq L(:,j-1)$ (where the containment applies to the set of nonzero indices). At the end of the symbolic factorization step, we test whether $nnz(L(:,j)) = nnz(L(:,j-1)) - 1$. Column j joins column $j-1$'s supernode if and only if both tests are passed.

T3 supernodes also require the diagonal block of U to be full. To check this, it suffices to check whether the single element u_{rj} is nonzero, where r is the first column index of the supernode. This follows from Corollary 2: $u_{rj} \neq 0$ implies that $u_{ij} \neq 0$ for all $r \leq i \leq j$. Indeed, we can even omit the test $L(:,j) \subseteq L(:,j-1)$ for T3 supernodes. If $u_{rj} \neq 0$, then $u_{j-1,j} \neq 0$, which means that column $j-1$ updates column j, which implies $L(:,j) \subseteq L(:,j-1)$. Thus we have proved the following.

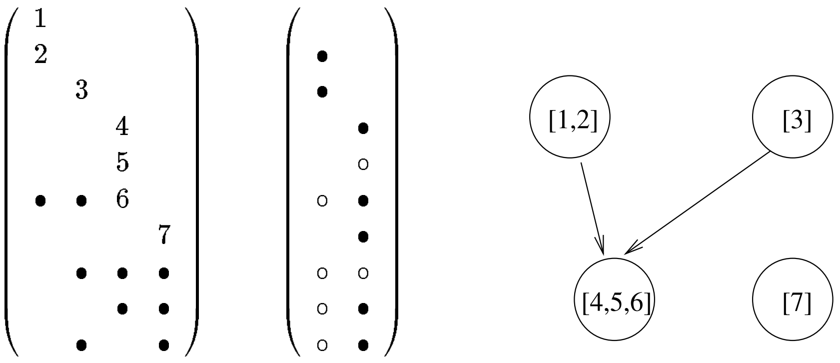
Theorem 4 *Suppose a T3 supernode begins with column r and extends at least through column $j-1$. Column j belongs to this supernode if and only if $u_{rj} \neq 0$ and $nnz(L(:,j)) = nnz(L(:,j-1)) - 1$.*

For either T2 or T3 supernodes, it is straightforward to implement the relaxed versions discussed in Section 2.4. Also, since the main benefits of supernodes come when they fit into the cache, we impose a maximum size for a supernode.

4.4 Panel depth-first search

The supernode-panel algorithm consists of an outer factorization (applying updates from supernodes to the active panel) and an inner factorization (applying supernode-column updates within the active panel). Each has its own symbolic factorization. The outer symbolic factorization happens once per panel, and determines two things: (1) a single column structure, which is the union of the structures of the panel columns, and (2) which supernodes update each column of the panel, and in what order. This is the information that the supernode-panel update loop in Figure 7 needs.

The inner symbolic factorization happens once for each column of the panel, interleaved column by column with the inner numeric factorization. In addition to determining the nonzero structure of the active column and which supernodes within the panel will update the active column, the inner symbolic factorization is also responsible for forming supernodes (that is, for deciding whether the active column will start a new supernode) and for symmetric structural pruning. The inner symbolic factorization is, therefore, exactly the supernode-column symbolic factorization described above.



Reduced supernodal R $\quad$ $A(:, 8\colon 9)$

Figure 12: The supernodal directed graph corresponding to $L(1\colon 7, 1\colon 7)^T$.

The outer symbolic factorization must determine the structures of columns j to $j+w-1$, i.e., the structure of the whole panel, and also a topological order for $U(1\colon j, j\colon j+w-1)$ *en masse.* To this end, we developed an efficient panel depth-first search scheme, which is a slight modification of the column DFS. The panel depth-first search algorithm maintains a single postorder DFS list for all w columns of the panel. Let us call this the PO list, which is initially empty. The algorithm invokes the column depth-search procedure for each column from j to $j+w-1$. In the column DFS, each time the search backtracks from a vertex, that vertex is appended to the PO list. In the panel DFS, however, the vertex is appended to the PO list *only if it is not already on the list.* This gives a single list that includes every position that is nonzero in any panel column, just once; and the entire list is in (reverse) topological order. Thus the order of updates specified by the list is acceptable for each of the w individual panel columns.

We illustrate the idea in Figure 12, using the sample matrix from Figure 11 and a panel of width two. The first seven columns have been factored, and the current panel consists of columns 8 and 9. In the panel, nonzeros of A are shown as "$\bullet$" and fill in F is shown as "$\circ$". The depth-first search for column 8 starts from vertices 2 and 3. After that search is finished, the panel postorder list is $PO = (6,2,3)$. Now the depth-first search for column 9 starts from vertices 6 and 7 (not 4, since 6 is the representative vertex for the supernode containing column 4). This DFS only appends 7 to the PO list, because 6 is already on the list. Thus, the final list for this panel is $PO = (6,2,3,7)$. The postorder list of column 8 is $(6,2,3)$ and the postorder list of column 9 is $(6,7)$, which are simply two sublists of the panel PO list. The topological order is the reverse of PO, or $(7,3,2,6)$. In the loop of line 3 of Figure 7, we follow this topological order to schedule the updating supernodes and perform numeric updates to columns of the panel.

5 Evaluation

In this section, we apply our algorithms to practical matrices from various sources. We will compare the performance of SuperLU, our supernode-panel code, with its predecessors, and with other approaches to sparse LU factorization.

Matrix	Name	s	Rows	Nonzeros	Nonzeros/row
1	MEMPLUS	.983	17758	99147	5.6
2	GEMAT11	.002	4929	33185	6.7
3	RDIST1	.062	4134	9408	2.3
4	MCFE	.709	765	24382	31.8
5	SHERMAN5	.780	3312	20793	6.3
6	LNSP3937	.869	3937	25407	6.5
7	LNS3937	.869	3937	25407	6.5
8	SHERMAN3	1.000	5005	20033	4.0
9	JPWH991	.947	991	6027	6.1
10	ORANI678	.073	2529	90158	35.6
11	ORSREG1	1.000	2205	14133	6.4
12	SAYLR4	1.000	3564	22316	6.3
13	SHYY161	.769	76480	329762	4.3
14	VENKAT01	1.000	62424	1717792	27.5
15	GOODWIN	.642	7320	324772	44.4
16	INACCURA	1.000	16146	1015156	62.9
17	DENSE1000	1.000	1000	1000000	1000
18	RAEFSKY3	1.000	21200	1488768	70.2
19	WANG3	1.000	26064	177168	6.8
20	RAEFSKY4	1.000	19779	1316789	66.6
21	VAVASIS3	.001	41092	1683902	41.0

Table 2: Benchmark matrices. Structural symmetry s is defined to be the fraction of the nonzeros matched by nonzeros in symmetric locations. None of the matrices are numerically symmetric.

5.1 Experimental setup

Table 2 lists 21 matrices with some characteristics of their nonzero structures. The matrices are sorted in increasing order of $flops/nnz(F)$, the ratio of the number of floating point operations to the number of nonzeros $nnz(F)$ in the factored matrix $F = U + L - I$. The reason for this order will be described in more detail in section 5.5. Some of the matrices are from the Harwell-Boeing collection [9]. Most of the larger matrices are from the ftp site maintained by Tim Davis of the University of Florida.[3] Those matrices are as follows. GOODWIN is a finite element matrix in a nonlinear solver for a fluid mechanics problem, provided by Ralph Goodwin of the University of Illinois at Urbana-Champaign. MEMPLUS is a circuit simulation matrix from Steve Hamm of Motorola. INACCURA, RAEFSKY3/4, and VENKAT01 were provided by Horst Simon of Silicon Graphics. RAEFSKY3 is from a fluid structure interaction turbulence problem. RAEFSKY4 is from a buckling problem for a container model. VENKAT01 comes from an implicit 2-D Euler solver for an unstructured grid in a flow simulation. WANG3 is from solving a coupled nonlinear PDE system in a 3-D ($30 \times 30 \times 30$ uniform mesh) semiconductor device simulation, as provided by Song Wang of the University of New South Wales, Sydney. SHYY161 is derived from a direct, fully-coupled method for solving the Navier-Stokes equations for viscous flow calculations, provided by Wei Shyy of the University of Florida. VAVASIS3 is an unsymmetric augmented matrix for a 2-D PDE with highly varying coefficients [33]. DENSE1000 is a dense 1000×1000 matrix.

In this paper, we do not address the performance of preordering for sparsity. Matrices 1, 14 and 19 were symmetrically permuted by Matlab's symmetric minimum degree ordering on $A + A^T$. For

[3]ftp.cis.ufl.edu, in pub/umfpack/matrices

IBM RS/6000-590							
Matrix	$nnz(F)$	$\frac{nnz(F)}{nnz(A)}$	#flops (10^6)	Seconds	MFLOPS	Fraction of time in numeric	Fraction of flops in DGEMV
1	140388	1.4	1.8	0.59	2.98	19%	78%
2	93370	2.8	1.6	0.27	5.97	34%	81%
3	338624	36.0	12.9	0.98	13.04	41%	85%
4	118717	4.8	11.7	0.44	24.91	58%	95%
5	281876	13.6	30.4	0.94	32.30	55%	92%
6	534229	21.0	59.6	2.01	28.77	55%	95%
7	559043	22.0	66.4	2.19	30.04	55%	96%
8	433376	21.6	61.7	1.41	44.41	51%	87%
9	161334	26.7	23.5	0.62	37.34	56%	94%
10	623806	6.9	103.9	4.47	22.97	63%	97%
11	453112	32.1	76.6	1.46	52.81	57%	89%
12	774310	34.7	144.7	2.78	51.67	56%	90%
13	7635773	23.2	1578.5	28.89	54.58	52%	91%
14	12785320	7.4	2730.9	40.06	66.62	59%	92%
15	3109585	9.6	665.6	12.66	51.75	65%	92%
16	10016266	9.9	4126.1	68.65	60.17	62%	97%
17	1000000	1.0	666.7	5.74	116.06	70%	95%
18	17631651	11.8	12128.7	112.27	109.03	74%	96%
19	13287108	74.9	14559.4	116.94	124.50	78%	98%
20	26714111	20.3	31307.1	257.88	120.47	79%	98%
21	49687658	29.5	89865.1	789.65	113.81	80%	98%

Table 3: Performance of SUPERLU on an IBM RS/6000-590.

all other matrices, the columns were permuted by Matlab's minimum degree ordering of $A^T A$ [18].

We performed the numerical experiments on an IBM RS/6000-590. The CPU clock rate is 66.5 MHz. The processor has two branch units, two fixed-point units, and two floating-point units, which can all operate in parallel if there are no dependencies. In particular, each FPU can perform two operations (a multiply and an add or subtract) at every cycle. Thus, the peak floating-point performance is 266 MFLOPS. The data cache is of size 256 KB with 256-byte lines, and is 4-way set-associative with LRU replacement policy. There is a separate 32 KB instruction cache. The size of the main memory is 768 MB. The SUPERLU algorithm is implemented in C; we use the AIX xlc compiler with -O3 optimization.

5.2 Performance of the code

Table 3 presents the performance of the SUPERLU code on this system. All floating point computations are in double precision.

In the inner loops of our sparse code, we call the two dense BLAS-2 routines DTRSV (triangular solve) and DGEMV (matrix-vector multiply) provided in the IBM ESSL library [23], whose BLAS-3 matrix-matrix multiply routine (DGEMM) achieves about 250 MFLOPS when dimension of the matrix is larger than 60 [1]. In our sparse algorithm, we find that DGEMV typically accounts for more than 80% of the floating-point operations. As shown in the last column of Table 3, this percentage is higher than 95% for many matrices. Our measurements reveal that for typical dimensions arising from the benchmark matrices, DGEMV achieves roughly 235 MFLOPS if the data

IBM RS/6000-590					
	GP	GP-Mod	SupCol-F	SupCol-C	SuperLU
Matrix	xlf -O3	xlf -O3	xlf -O3	xlc -O3	xlc -O3
1	1.00	1.48	1.18	1.05	.68
2	1.00	1.69	1.23	1.29	1.00
3	1.00	2.75	2.60	2.24	1.94
4	1.00	3.44	4.31	3.52	3.52
5	1.00	3.43	5.04	4.57	4.23
6	1.00	3.39	4.21	3.86	3.54
7	1.00	3.39	4.29	3.85	3.55
8	1.00	3.54	6.19	5.99	5.27
9	1.00	3.61	4.71	4.21	4.48
10	1.00	3.55	3.88	2.98	3.10
11	1.00	3.64	5.98	5.86	5.98
12	1.00	3.67	6.39	5.99	6.30
13	1.00	3.65	6.71	6.46	5.67
14	1.00	3.86	8.49	8.33	8.87
15	1.00	3.84	6.91	6.46	7.16
16	1.00	4.17	7.55	7.24	7.94
17	1.00	4.21	10.22	9.78	14.54
18	1.00	4.30	11.70	11.54	14.00
19	1.00	4.34	12.32	12.23	15.75
20	1.00	4.35	12.18	11.89	15.39
21	1.00	4.79	13.11	13.12	15.63
Geometric Mean	1.00	3.46	5.56	5.19	5.29

Table 4: Speedups achieved by each enhancement over the GP column-column code, on the RS/6000.

are from cache. If the data are fetched from main memory, this rate can drop by a factor of 2 or 3.

The Blas speed is clearly an upper bound on the overall factorization rate. However, because symbolic manipulation usually takes a nontrivial amount of time, this bound could be much higher than the actual sparse code performance. Table 3 also presents the percentage of the total execution time spent in numeric computation. For matrices 1 and 2, the program spent less than 35% of its time in the numeric part. Compared to the others, these two matrices are sparser, have less fill, and have smaller supernodes, so our supernodal techniques are less applicable. Matrix 2 is also highly unsymmetric, which makes the symmetric structural reduction technique less effective. However, it is important to note that the execution times for these two matrices are small.

For larger and denser matrices such as 17–21, we achieve between 110 and 125 Mflops, which is about half of the machine peak. These matrices take much longer to factor, which could be a serious bottleneck in an iterative simulation process. Our techniques are successful in reducing the solution times for this type of problem.

For a dense 1000×1000 matrix, our code achieves 116 Mflops. This compares with 168 Mflops reported in the Lapack manual [2] on a matrix of this size, and 236 Mflops reported in the online Linpack benchmark files [25].

Sparc 20					
	GP	GP-MOD	SUPCOL-F	SUPCOL-C	SUPERLU
Matrix	f77 -O3	f77 -O3	f77 -O3	cc -xO3	cc -xO3
1	1.00	1.19	.98	1.25	.75
2	1.00	1.32	1.26	1.71	1.09
3	1.00	1.64	1.75	1.65	1.58
4	1.00	1.80	2.36	2.16	2.32
5	1.00	1.82	2.74	2.74	2.81
6	1.00	1.82	2.49	2.36	2.33
7	1.00	1.84	2.58	2.47	2.27
8	1.00	1.90	3.11	3.19	3.09
9	1.00	1.85	2.41	2.39	2.58
10	1.00	1.86	2.08	1.78	1.81
11	1.00	1.89	3.02	3.09	3.20
12	1.00	1.95	3.03	3.09	3.32
13	1.00	2.08	3.48	3.47	3.55
15	1.00	1.89	3.05	3.02	3.91
17	1.00	1.96	3.56	3.13	4.89
Geometric Mean	1.00	1.77	2.38	2.40	2.39

Table 5: Speedups achieved by each enhancement over the GP column-column code, on a Sparc 20. The CPU is rated at 60.0 MHz, and there is a 1 MB external cache. This system does not provide a BLAS library, so we use our own C BLAS routines. Some large problems could not be tested because of physical memory constraints.

5.3 Comparison with previous row or column LU algorithms

In this section, we compare the performance of SUPERLU with several of its predecessors, including the partial pivoting code by Gilbert and Peierls [20] (referred to as GP), Eisenstat and Liu's improved GP code that incorporates symmetric reduction [13] (referred to as GP-MOD), and two versions of our supernode-column code (referred to as SUPCOL-F and SUPCOL-C). GP, GP-MOD, and SUPCOL-F are written in Fortran; SUPCOL-C and SUPERLU are written in C. We translated SUPCOL-F literally into C to produce SUPCOL-C; no changes in algorithms or data structures were made. SUPCOL-F, SUPCOL-C and SUPERLU use ESSL BLAS. (Matlab contains a C implementation of GP [18], which we did not test here.)

Tables 4 through 6 present the speedups achieved by various enhancements over the original GP column-column code on high-end workstations from three vendors. Thus, for example, a speedup of 2 means that the running time was half that of GP. The numbers in the last row of each table are obtained by averaging the speedups in the corresponding column. We make the following observations about the results on the IBM RS/6000:

- Symmetric structure pruning (GP-MOD) is very effective in reducing the graph search time. This significantly decreases the symbolic time in the GP code. It achieves speedup in all problems.

- Supernodes (SUPCOL) restrict the search to the supernodal graph, and allow the numeric kernels to employ dense BLAS-2 operations. The effects are not as dramatic as the pruning technique. For some matrices, such as 1–3, the results are not as good as GP-MOD. This is

DEC Alpha					
Matrix	GP f77 -O2	GP-MOD f77 -O2	SUPCOL-F f77 -O2	SUPCOL-C cc -O2	SUPERLU cc -O2
1	1.00	1.19	.96	.98	.55
2	1.00	1.31	1.10	1.10	.78
3	1.00	1.65	1.76	1.37	1.27
4	1.00	1.90	2.20	2.12	2.09
5	1.00	1.81	2.60	2.63	2.79
6	1.00	1.84	2.38	2.25	2.35
7	1.00	1.81	2.32	2.24	2.33
8	1.00	1.84	2.33	3.23	3.54
9	1.00	1.92	2.65	2.46	2.79
10	1.00	1.80	2.05	1.61	1.85
11	1.00	1.82	3.04	3.09	2.64
12	1.00	1.78	3.15	3.13	4.19
13	1.00	1.80	3.33	3.43	3.84
15	1.00	1.77	2.84	2.82	4.19
17	1.00	1.83	4.47	3.60	6.45
Geometric Mean	1.00	1.72	2.38	2.24	2.40

Table 6: Speedups achieved by each enhancement over the GP column-column code, on a DEC Alpha. The CPU is rated at 200 MHz, and there is a 512 KB external cache. We use the BLAS routines from DEC's DXML library. Some large problems could not be tested because of physical memory constraints.

because the supernodes are often small, especially for sparser problems.

- Supernode-panel updates (SUPERLU) reduce the cache miss rate and exploit dense substructures in the factor F. For problems without much structure, the gain is often offset by various overheads. However, the benefits become evident for larger or denser problems.

- The Fortran compiler produces slightly faster code than the C compiler (for SUPCOL) on both the IBM and DEC machines. The difference is about 5% to 15% for small problems, and is less for large problems where most of the time is spent in BLAS routines. We have seen smaller runtime differences between the codes generated by the SunOS 5.4 Fortran and C compilers on the Sun Sparc 20.

  Matrix 10 is an exception: the Fortran code is about 25% faster on the IBM RS/6000, 17% faster on the Sun Sparc 20, and 27% faster on the DEC Alpha.

As more and more sophisticated techniques are introduced, the overhead cost of the code is also increased to some extent. This overhead can show up prominently in small problems. For example, on the IBM RS/6000, GP-MOD works better than any of the subsequent codes for problems 1–3.

5.4 Comparison with other approaches to LU factorization

In this section we compare our supernode-panel algorithm with other popular algorithms to solve the unsymmetric linear system $Ax = b$, using the matrices from our benchmark suite. The right-hand side vector b is constructed so that the solution is $x_i = 1 + i/n$.

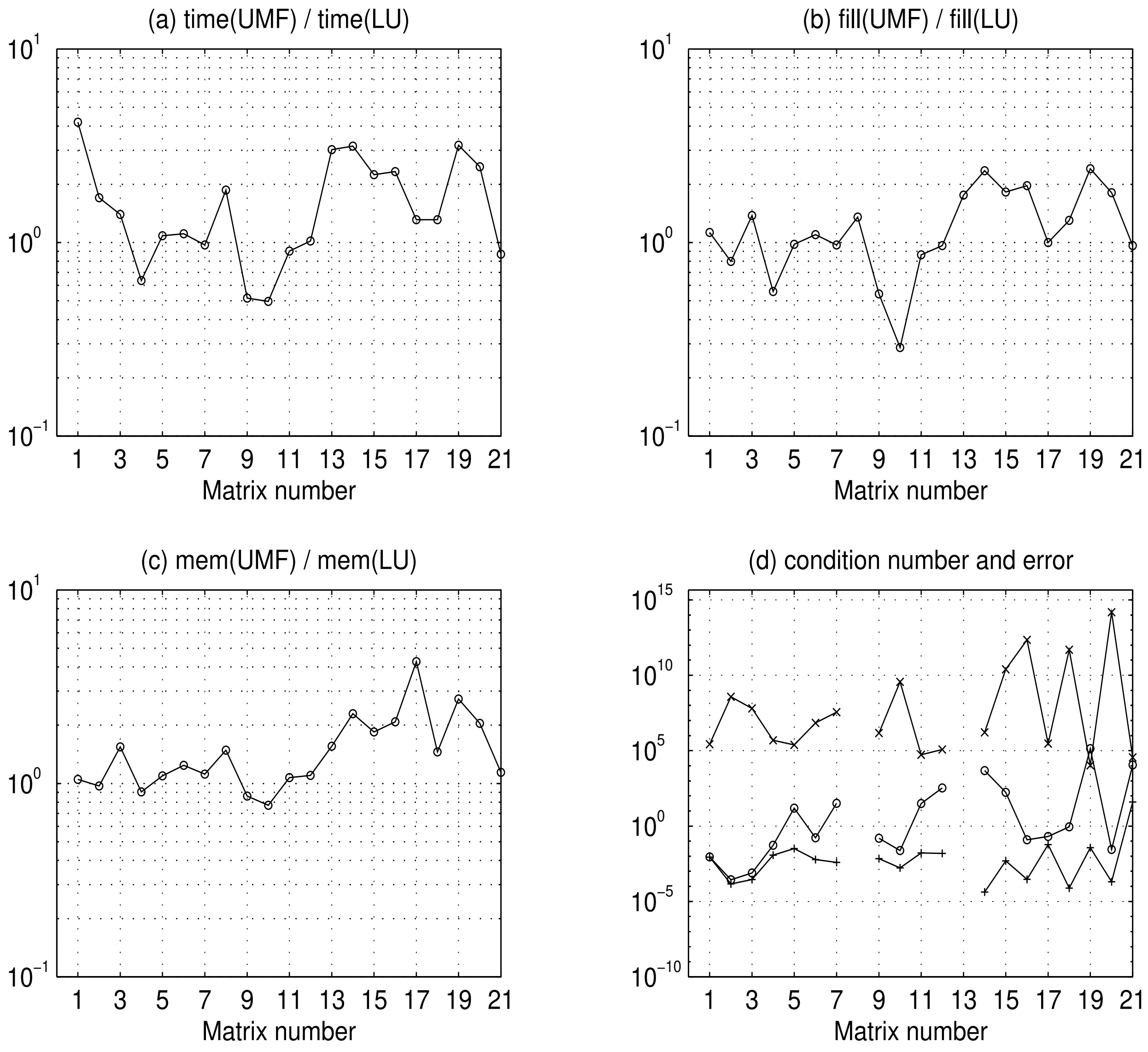


Figure 13: Comparison of SuperLU algorithm with unsymmetric multifrontal algorithm implemented in Umfpack on an IBM RS/6000-590, with Blas routines from the ESSL library. In (d), we plot the estimated condition number $\kappa_\infty(A)$ labeled with "x", $err(\text{Umfpack})/(\kappa_\infty(A)\cdot\epsilon)$ labeled with "o", and $err(\text{SuperLU})/(\kappa_\infty(A)\cdot\epsilon)$ labeled with "+". These plots do not show Matrix 8 and 13, because their condition numbers are larger than $1/\epsilon$.

In particular, we consider the unsymmetric multifrontal algorithm implemented in UMFPACK version 1.0 [7]. UMFPACK is implemented in Fortran, so we use the AIX xlf compiler with -O3 optimization, and link with the IBM ESSL library for BLAS calls. We use the parameter settings for UMFPACK suggested by its authors [6].

Figure 13 compares several aspects of the two implementations. We plot the ratio of each individual measure from UMFPACK to that of our SUPERLU code. The memory requirement only counts the amount of memory actually used, excluding any external fragmentation.

Neither code is always faster than the other. For six problems, UMFPACK is faster than SUPERLU, by at most a factor of two. For the rest of problems, SUPERLU is faster than UMFPACK. For seven out of the 21 matrices, SUPERLU runs more than twice as fast as UMFPACK.

Figure 13(d) compares the solution accuracy and stability of the two approaches, without using iterative refinement. SUPERLU consistently delivers more accurate solutions, because UMFPACK uses a threshold pivoting strategy to trade off stability and sparsity. In particular, for stable algorithms we expect the normalized error to be

$$\frac{\|x - \tilde{x}\|_\infty}{\|\tilde{x}\|_\infty} \cdot \frac{1}{\kappa_\infty(A) \cdot \epsilon} = \Theta(1) \ .$$

But very frequently, this error from UMFPACK is much larger than $\Theta(1)$. For Matrix 19, the backward error is as large as $10^5\ \epsilon$. To mitigate the instability, we experimented with working-precision iterative refinement in UMFPACK, and found that in many cases one iteration can reduce the backward error to a level comparable to that of SUPERLU. The refinement process normally takes less than 2% of the factorization time. We recommend that UMFPACK incorporate this inexpensive technique to guarantee backward stability. The iteration can be terminated when the scaled residual $\|\tilde{r}\|_\infty / \|A\|_\infty \|\tilde{x}\|_\infty$ is acceptable, say less than machine precision ϵ.

In our SUPERLU software, equilibration and refinement are options for the user. We compute the componentwise relative backward error $\omega = \max_i(|\tilde{r}|_i / (|A||\tilde{x}| + |b|)_i)$, and stop refining when $\omega \leq \epsilon$ or ω does not decrease by at least a factor of two. See Arioli, Demmel, and Duff [3] for details. In practice, most of the test matrices take only one or two refinement steps to meet this criterion.

The time for SUPERLU does not include the time to reorder the columns. UMFPACK is less sensitive to the initial column ordering, because it dynamically permutes the columns for sparsity. Surprisingly, Figure 13(b) seems to suggest that for large matrices the dynamic fill-reducing approach used in UMFPACK is less effective than the minimum degree ordering algorithms.

5.5 Understanding cache behavior and parameters

In this subsection, we analyze the behavior of SUPERLU in detail. We wish to understand when our algorithm is significantly faster than other algorithms. We would like performance-predicting variable(s) that depend on "intrinsic" properties of the problem, such as the sparsity structure, rather than algorithmic details and machine characteristics. We begin by analyzing the speedups of our enhanced codes over the base GP implementation. Figures 14, 15 and 16 depict the speedups and the characteristics of the matrices, with panel size $w = 8$.

5.5.1 How much cache reuse can we expect?

As discussed in Section 3.2, the supernode-panel algorithm gets its primary gains from improved data locality, by reusing a cached supernode several times. To understand how much cache reuse

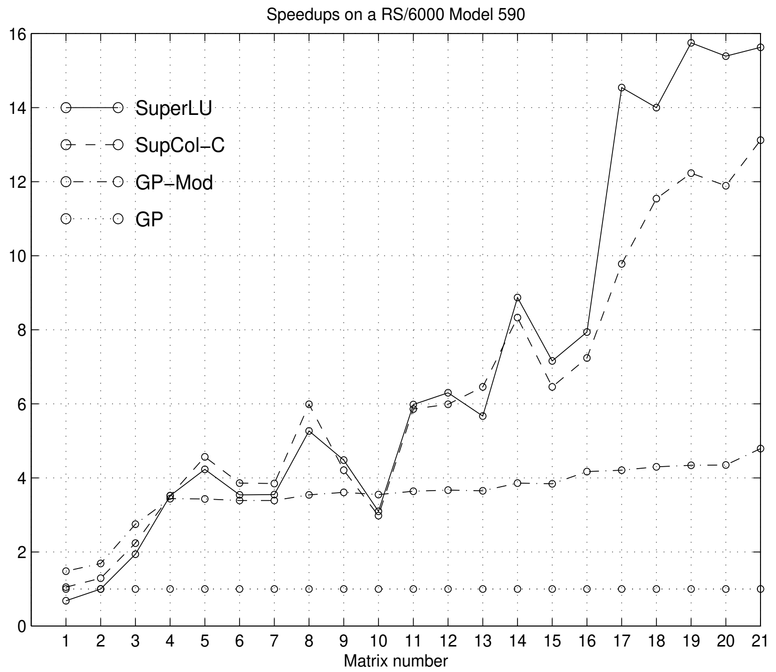


Figure 14: Speedups of each enhancement over GP code, on the RS/6000

we can hope for, we computed two statistics: *ops-per-nz* and *ops-per-ref*. After defining these statistics carefully, we discuss which more successfully measures reuse.

Ops-per-nz is simply calculated as $\#flops/nnz(F)$, the total number of floating point operations per nonzero in the filled matrix F. If there were perfect cache behavior, i.e., one cache miss per data item (ignoring the effect of cache line size), then *ops-per-nz* would exactly measure the amount of work per cache miss. In reality, *ops-per-nz* is an upper bound on the reuse. Note that *ops-per-nz* depends only on the fact that we are performing Gaussian elimination with partial pivoting, not on algorithmic or machine details. *Ops-per-nz* is a natural measure of potential data reuse, because it has long been used to distinguish among the different levels of BLAS.

In contrast, *ops-per-ref* provides a lower bound on cache reuse, and does depend on the details of the SuperLU algorithm. *Ops-per-ref* looks at each supernode-panel update separately, and assumes that all the associated data is outside the cache before beginning the update. This pessimistic assumption limits the potential reuse to twice the panel size, $2w$.

Now we define *ops-per-ref* more carefully. Consider a single update from supernode $(r\colon s)$ to panel $(j\colon j+w-1)$. Depending on the panel's nonzero structure, each entry in the updating supernode is used to update from 0 to w panel columns. Thus each entry in the updating supernode participates in between 0 and $2w$ floating point operations during a sup-panel update. We assume that the supernode entry is brought into cache from main memory exactly once for the entire sup-panel update, if it is used at all. Thus, during a single sup-panel update, each entry accessed in the updating supernode accounts for between 2 and $2w$ *operations per reference*. The *ops-per-ref* statistic is the average of this number over all entries in all sup-panel updates. It measures how many times the average supernode entry is used each time it is brought into cache from main memory. *Ops-per-ref* ranges from 2 to $2w$, with larger values indicating better cache use. If there is little correlation between the row structures of the columns in each panel, *ops-per-ref* will be small; if there is perfect correlation, as in a dense matrix, it will be close to $2w$.

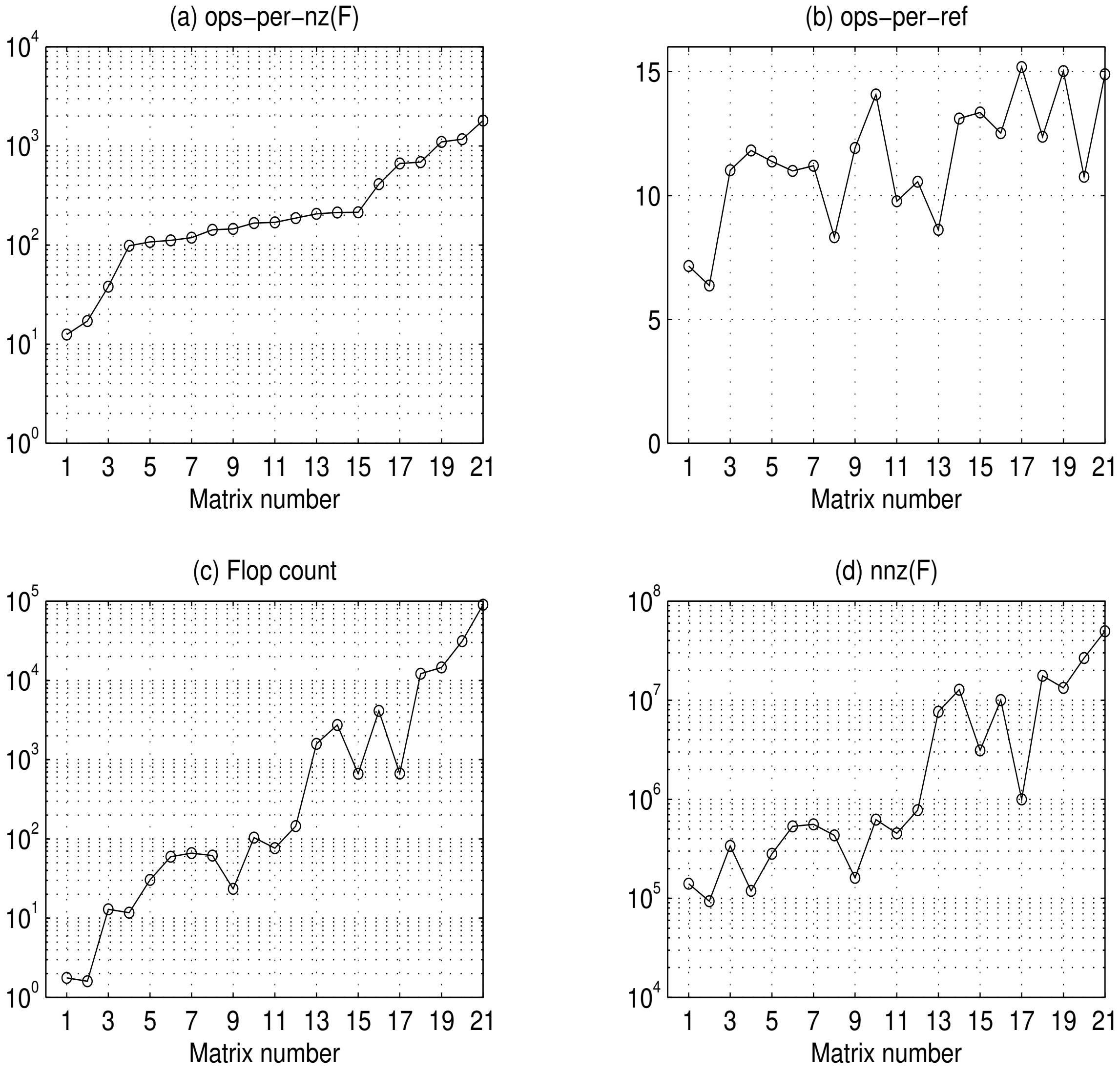


Figure 15: Some characteristics of the matrices.

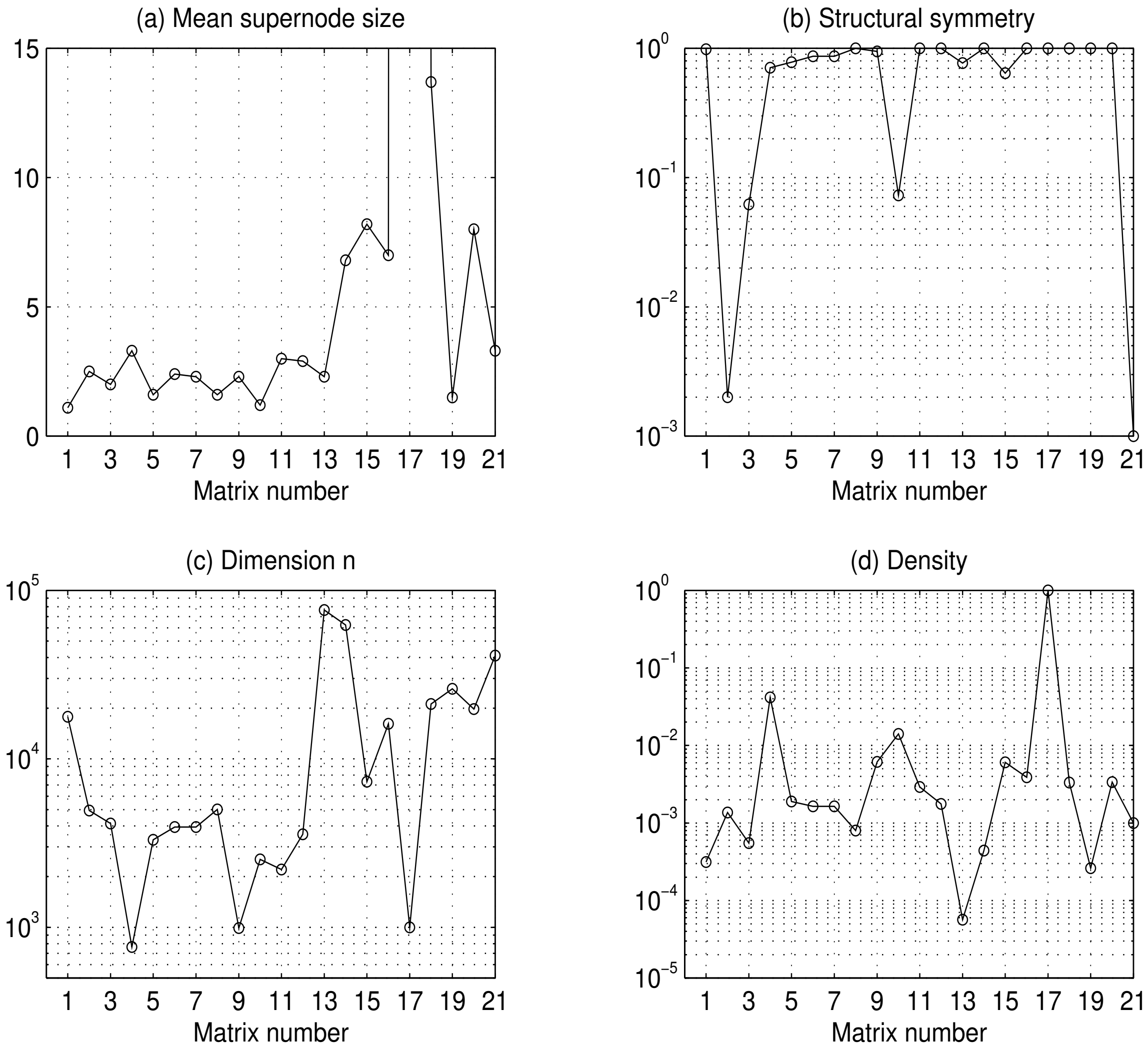


Figure 16: Some intrinsic properties of the matrices.

Now we describe how we compute the average *ops-per-ref* for the entire factorization. For each updating supernode $(r\colon s)$ and each panel $(j\colon j+w-1)$ (see Figure 7), define

$$ksmin = \min_{j \le jj < j+w, r \le i \le s} \{i | A(i, jj) \neq 0\}.$$

Then $nnz(L(r\colon n, ksmin\colon s))$ entries of the supernode are referenced in the sup-panel update. The dense triangular solve in column jj of the update takes $(s - ks + 1)(s - ks)$ flops, where $ks = min_{r \le i \le s}\{i | A(i, jj) \neq 0\}$. The matrix-vector multiply uses $2(s - ks + 1)nnz(L(s + 1\colon n, s))$ flops. We count both additions and multiplications. For all panel updates, we sum the memory reference counts and the flop counts, then divide the second sum by the first to arrive at an average *ops-per-ref*.

Now we compare the predictive powers of *ops-per-nz* (Figure 15 (a)) and *ops-per-ref* (Figure 15 (b)) in predicting speedup (Figure 14). The superiority of *ops-per-nz* is evident; it is much more strongly correlated with the speedup of SuperLU than *ops-per-ref*. This is good news, because *ops-per-nz* measures the best case reuse, and *ops-per-ref* the worst case. But neither statistic captures all the variation in the performance. In future work, we hope to use a hardware monitor to measure the exact cache reuse rate. (This data could also be obtained from a simulator, but the matrices we are interested in are much too large for a simulator to be viable.)

5.5.2 How large are the supernodes?

The supernode size determines the size of the matrix to be passed to matrix-vector multiply and other BLAS-2 routines in our algorithm. Figure 16(a) shows the average number of columns in the supernodes of the matrices, after amalgamating the relaxed supernodes at the bottom of the column etree. The average size is usually quite small.

More important than average size is the distribution of supernode sizes. In sparse Gaussian elimination, more fill tends to occur in the later stages. Usually there is a large percentage of small supernodes corresponding to the fringe of the column etree, even after amalgamation. Larger supernodes appear at the higher levels of the tree. In Figure 17 we plot the histograms of the supernode size for four matrices chosen to exhibit a wide range of behavior. Matrix 1 has 16378 supernodes, all but one of which have less than 12 columns; the single large supernode, with 115 columns, is the dense submatrix at the bottom right corner of F. Matrix 15 has more supernodes distributed over a wider spectrum; it has 13 supernodes with 54 to 59 columns. This matrix gives greater speedups over the non-supernodal codes.

Figure 16 also plots three other properties of each matrix: structural symmetry, dimension, and density. None of them have any significant correlation with the performance. The effectiveness of symmetric reduction depends on F being structurally symmetric, which depends on the choice of pivots. So, structural symmetry of A does not gives any useful information.

We note that the speedups achieved by the dense 1000×1000 problem (matrix 17) show the best performance gains, because this matrix has large supernodes and exhibits ideal data reuse. It achieves speedups of 49% to 79% on the three platforms. The gains for any sparse matrix should be smaller than this.

5.5.3 Blocking parameters

In our hybrid blocking algorithm (Figure 9), we need to select appropriate values for the parameters that describe the two-dimensional data blocking: panel width w, maximum supernode size t, and row block size b. The key considerations are that the active data we access in the inner loop (the

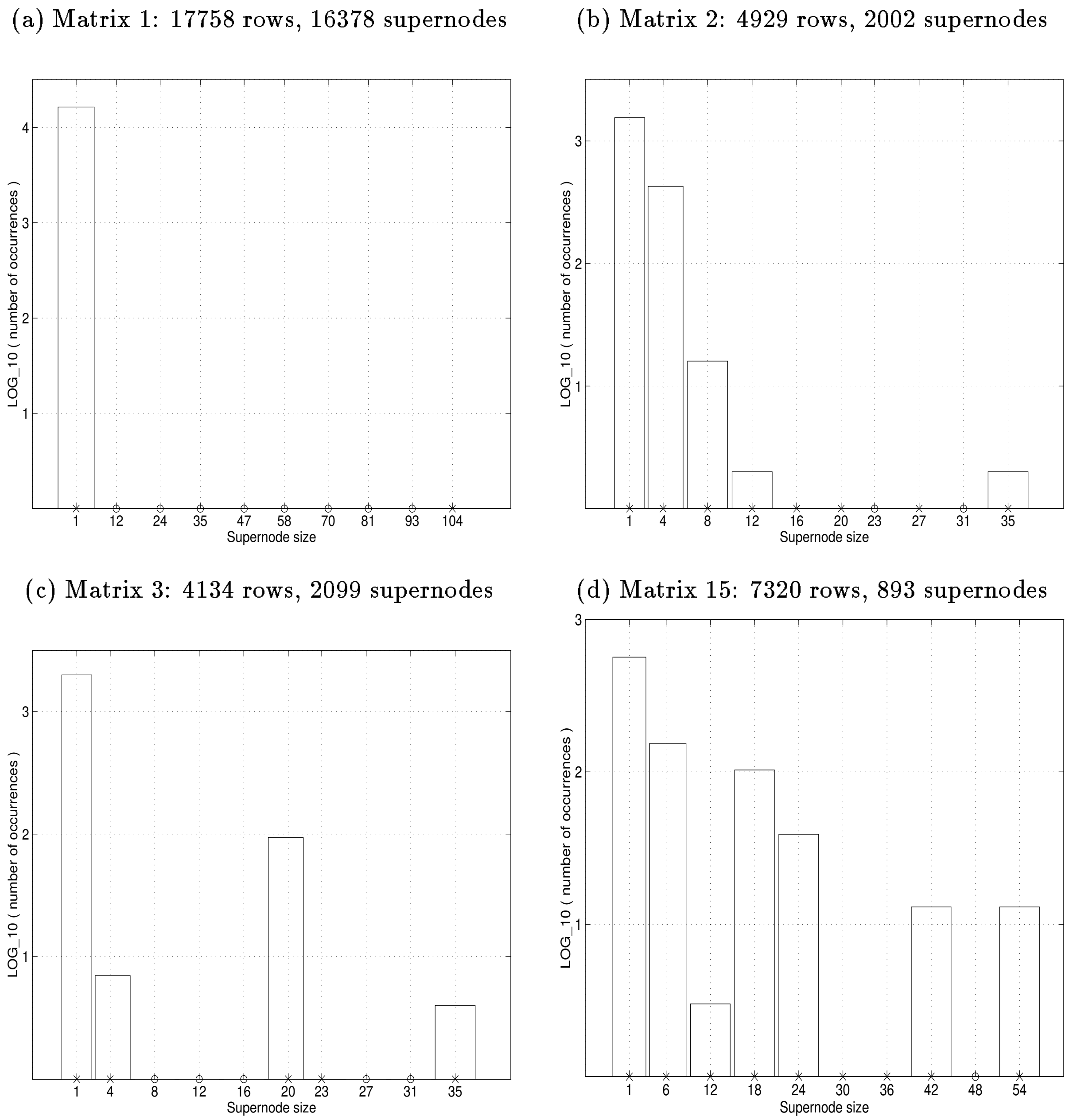


Figure 17: Distribution of supernode size for four matrices. The number at the bottom of each bar is the smallest supernode size in that bin. The mark "o" at the bottom of a bin indicates zero occurrences. Artificial supernodes of granularity $r = 4$ are used (see Section 2.4).

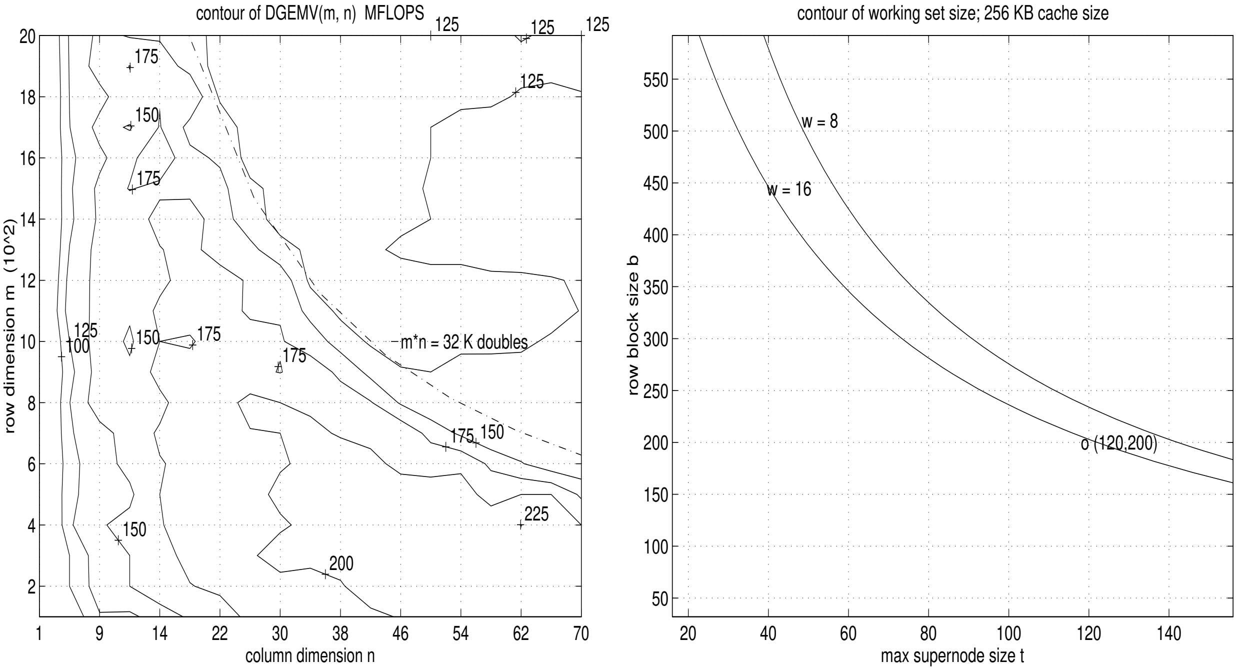


Figure 18: (a) Contour plot of DGEMV performance. (b) Contour plot of working set in 2-D algorithm.

working set) should fit into the cache, and that the matrices presented to the BLAS-2 routine DGEMV should be the sizes and shapes for which that routine is optimized. Here we describe in detail the methodology we used to choose parameters for the IBM RS/6000.

- DGEMV **optimization.** As indicated in the last column of Table 3, the majority of the floating-point operations are in the matrix-vector multiply. The dimensions (m, n) of the matrices in calls to DGEMV vary greatly depending on the supernode dimensions. Very often, the supernode is a tall and skinny matrix, that is, $m \gg n$. We measured the DGEMV MFLOPS rate for various m and n, and present a contour plot in the (m, n) plane in Figure 18(a). Each contour represents a constant MFLOPS rate. The dashed curve represents $mn = 32\text{K}$ double floats, or a cache capacity of 256 KB. In the optimum region, we achieve more than 200 MFLOPS; outside this region, performance drops either because the matrices exceed the cache capacity, or because the column dimension n is too small.

- **Working set.** By studying the data access pattern in the inner loop of the 2-D algorithm, lines (7–9) in Figure 9, we find that the working set size is the following function of w, t, and b, as shown in Figure 19:

$$WS = \underbrace{b \times t}_{\text{row block from supernode}} + \underbrace{(t+b) \times w}_{\text{vectors in matrix-vector multiply}} + \underbrace{b \times w}_{\text{part of SPA structure}} .$$

  In Figure 18(b), we fix two w values, and plot the contour lines for $WS = 32\text{K}$ in the (t, b) plane. If the point (t, b) is below the contour curve, then the working set can fit in a cache of 32K double floats, or 256 kilobytes.

Based on this analysis, we use the following rules to set the parameters.

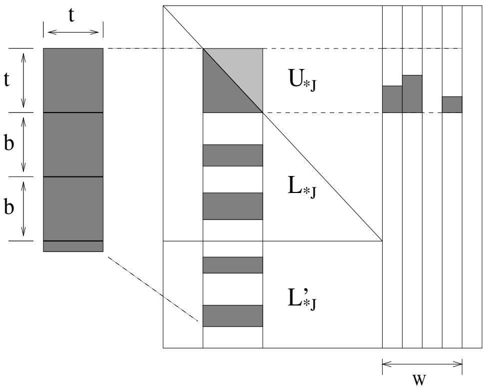


Figure 19: Parameters of the working set in the 2-D algorithm.

First we choose w, the width of the panel in columns. Larger panels mean more reuse of cached data in the outer factorization, but also mean that the inner factorization (by the sup-col algorithm) must be applied to larger matrices. We find empirically that the best choice for w is between 8 and 16. Performance tends to degrade for larger w.

Next we choose b, the number of rows per block, and t, the maximum number of columns in a supernode. Recall that b and t are upper bounds on the row and column dimensions of the call to DGEMV. We choose $t = 120$ and $b \approx 200$, which guarantees that the working set fits in cache (per Figure 18(b)), and that we can hope to be near the optimum region of DGEMV performance (per Figure 18(a)).

Recall that b is relevant only when we use row-wise blocking, that is, when the test "**if** $(r\colon s)$ is large" succeeds at line 4 of Figure 9. This implies that the 2-D scheme adds overhead only if the updating supernode is small. In the actual code, the test for a large supernode is

if $ncol > 40$ **and** $nrow > b$ **then** the supernode is large,

where $nrow$ is the number of dense rows below the diagonal block of the supernode, $ncol$ is the number of actual dense columns of the supernode updating the panel. In practice, this choice usually gives the best performance.

The best choice of the parameters w, t, and b depends on the machine architecture and on the BLAS implementation, but it is largely independent of the matrix structure. Thus we do not expect each user of SUPERLU to choose values for these parameters. Instead, our library code provides an inquiry function that returns the parameter values, much in the spirit of the LAPACK environment routine ILAENV. The machine-independent defaults will often give satisfactory performance. The methodology we have described here for the RS/6000 can serve as a guide for users who want to modify the inquiry function to give optimal performance for particular computer systems.

6 Remarks

6.1 The rest of the package

In addition to the LU factorization kernel described in this paper, we have developed a suite of supporting routines to solve linear systems. The complete SUPERLU package includes condition number estimation, iterative refinement of solutions, and componentwise error bounds for the refined solutions. These are all based on the dense matrix routines in LAPACK [2]. In addition, SuperLU includes a Matlab mex-file interface, so that our factor and solve routines can be called as alternatives to those built into Matlab.

6.2 Effect of the matrix on performance

The supernodal approach reduces both symbolic and numeric computation time. But unsymmetric supernodes tend to be smaller than supernodes in symmetric matrices. The supernode-panel method is most effective for large problems with enough dense submatrices to use dense block operations and exploit data locality. In this regard, the dense 1000×1000 example illustrates the largest possible gains. Dense blocks are necessary for top performance in all modern factorization algorithms, whether left-looking, right-looking, multifrontal, or any other style.

Our goal has been to develop sparse LU software that works well for problems with a wide range of characteristics. It is harder to achieve high flop rates on problems that are very sparse and have no structure to exploit; it is easier on problems that are denser or become so during elimination. Fortunately, the "hard" matrices by this definition generally take many fewer floating point operations than the "easy" ones, and hence take much less time to factor. Our combination of 1-D and 2-D blocking techniques gives a good performance compromise for all the problems we have studied, and with particularly good performance on the largest problems.

6.3 Effect of the computer system on performance

We have studied several characteristics of the computing platform that can affect the overall performance, including the Blas-2 speed and the cache size. Based on these factors, we can systematically make a good choice of the blocking parameters in the code so as to maximize the speed of the numeric kernel. Although we have empirical evidence only for the IBM RS/6000, we expect this methodology to be applicable to other systems (and Blas implementations) as well.

6.4 Possible enhancements

We are considering several possible enhancements to the SuperLU code. One is to switch to a dense LU code at a late stage of the factorization. It would be difficult to implement this in a supernode-column code, because that code is strictly left-looking, and only one column of the matrix is factored at a time. However, this would be much easier in the supernode-panel code. At the time we decide to switch, we simply treat the rest of the matrix columns (say d of them) as one panel, and perform the panel update to $A(1\colon n, n-d+1\colon n)$. (One might want to split this panel up for better cache behavior.) Then the reduced matrix at the bottom right corner can be factored by calling an efficient dense code, for example, from Lapack [2]. The dense code does not spend time on symbolic structure prediction and pruning, thus streamlining the numeric computation. We believe that, for large problems, the final dense submatrix will be big enough to make the switch beneficial. For example, for a 2-D $k \times k$ square grid problem ordered by nested dissection, the dimension of the final dense submatrix is $\frac{3}{2}k \times \frac{3}{2}k$; for a 3-D $k \times k \times k$ cubic grid, it is $\frac{3}{2}k^2 \times \frac{3}{2}k^2$, if pivots come from the diagonal.

To enhance SuperLU's performance on small problems, it would be possible to make a choice at runtime whether to use supernode-panel, supernode-column, or column-column updates. The choice would depend on the size of the matrix A and the expected properties of its supernodes; it might be based on an efficient symbolic computation of the density and supernode distribution of the Cholesky factor of $A^T A$ [21].

Could we make supernode-panel panel updates more effective by improving the similarity between the row structures of the columns in a panel? We believe this could be accomplished with a more sophisticated column permutation strategy. We could partition the nodes of the column etree into connected subtrees, grouping together nodes that have common descendants (and therefore the potential for updates from the same supernodes). Then the overall column order would be a two-level postorder, first within the subtrees (panels) and then among them. Again, it might be possible to use information about the Cholesky supernodes of $A^T A$ to guide this grouping.

We are also developing a parallel sparse LU algorithm based on SuperLU. In this context, we target large problems, especially those too big to be solved on a uniprocessor system. Therefore, we plan to parallelize the 2-D blocked supernode-panel algorithm, which has very good asymptotic behavior for large problems. The 2-D block-oriented layout has been shown to scale well for parallel sparse Cholesky factorization [22, 29].

Acknowledgements

We thank Rob Schreiber and Ed Rothberg for very helpful discussions during the development of SuperLU. Rob pushed us to find a way to do the supernode-panel update; Ed suggested the *ops-per-ref* statistic. We are also grateful to Tim Davis and Steve Vavasis for making their test problems available to us. The Institute for Mathematics and Its Applications at the University of Minnesota provided the fertile environment in which this work began.

References

[1] R.C. Agarwal, F.G. Gustavson, P. Palkar, and M. Zubair. A performance analysis of the subroutines in the ESSL/LAPACK call conversion interface (CCI). IBM T.J. Watson Research Center, Yorktown Heights, 1994.

[2] E. Anderson et al. *LAPACK User's Guide, Second Edition.* SIAM, Philadelphia, 1995.

[3] M. Arioli, J. W. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.*, 10(2):165–190, April 1989.

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

[5] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in sparse matrix methods for large sparse linear systems on vector supercomputers. *Intern. J. of Supercomputer Applications*, 1:10–30, 1987.

[6] T. A. Davis. User's guide for the unsymmetric-pattern multifrontal package (UMFPACK). Technical Report TR-93-020, Computer and Information Sciences Department, University of Florida, June 1993.

[7] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. Technical Report RAL 93-036, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, 1994.

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of basic linear algebra subroutines. *ACM Trans. Mathematical Software*, 14:1–17, 18–32, 1988.

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

[10] I. S. Duff and J. K. Reid. MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations. Technical Report RAL–93–072, Rutherford Appleton Laboratory, Oxon, UK, 1993.

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

[12] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in sparse unsymmetric symbolic factorization. *SIAM J. Matrix Analysis and Applications*, 13:202–211, 1992.

[13] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in a sparse partial pivoting code. *SIAM J. Scientific and Statistical Computing*, 14:253–257, 1993.

[14] J. A. George and E. Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM J. Scientific and Statistical Computing*, 6:390–409, 1985.

[15] J. A. George and E. Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM J. Scientific and Statistical Computing*, 8:877–898, 1987.

[16] J. R. Gilbert. Predicting structure in sparse matrix computations. *SIAM J. Matrix Analysis and Applications*, 15:62–79, 1994.

[17] J. R. Gilbert and J. W. H. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM J. Matrix Analysis and Applications*, 14:334–352, 1993.

[18] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in Matlab: Design and implementation. *SIAM J. Matrix Analysis and Applications*, 13:333–356, 1992.

[19] J. R. Gilbert and E. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, 1993.

[20] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Scientific and Statistical Computing*, 9:862–874, 1988.

[21] John R. Gilbert, Esmond G. Ng, and Barry W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Analysis and Applications*, 15:1075–1091, 1994.

[22] A. Gupta and V. Kumar. Optimally scalable parallel sparse cholesky factorization. In *The 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 442–447, 1995.

[23] *International Business Machines Corporation Engineering and Scientific Subroutine Library, Guide and Reference*. Version 2 Release 2, Order No. SC23-0526-01, 1994.

[24] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Analysis and Applications*, 11:134–172, 1990.

[25] PDS: The performance database server, http://performance.netlib.org/performance/, May 1995.

[26] E. G. Ng and B. W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Scientific and Statistical Computing*, 14:1034–1056, 1993.

[27] E. Rothberg and A. Gupta. Efficient sparse matrix factorization on high-performance workstations – exploiting the memory hierarchy. *ACM Trans. Mathematical Software*, 17:313–334, 1991.

[28] E. Rothberg and A. Gupta. An evaluation of left-looking, right-looking and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines. *Int. J. High Speed Computing*, 5:537–593, 1993.

[29] E. E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse cholesky factorization. In *Supercomputing*, pages 503–512, November 1993.

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

[31] A. H. Sherman. Algorithm 533: NSPIV, a FORTRAN subroutine for sparse Gaussian elimination with partial pivoting. *ACM Trans. Mathematical Software*, 4:391–398, 1978.

[32] H. Simon, P. Vu, and C. Yang. Performance of a supernodal general sparse solver on the CRAY Y-MP: 1.68 GFLOPS with autotasking. Technical Report TR SCA-TR-117, Boeing Computer Services, 1989.

[33] S. A. Vavasis. Stable finite elements for problems with wild coefficients. Technical Report 93–1364, Department of Computer Science, Cornell University, Ithaca, NY, 1993. To appear in *SIAM J. Numerical Analysis*.