Dynamic Supernodes in Sparse Cholesky Update/Downdate and Triangular Solves

TIMOTHY A. DAVIS and WILLIAM W. HAGER
University of Florida

The supernodal method for sparse Cholesky factorization represents the factor L as a set of supernodes, each consisting of a contiguous set of columns of L with identical nonzero pattern. A conventional supernode is stored as a dense submatrix. While this is suitable for sparse Cholesky factorization where the nonzero pattern of L does not change, it is not suitable for methods that modify a sparse Cholesky factorization after a low-rank change to A (an update/downdate, $\overline{A} = A \pm WW^T$). Supernodes merge and split apart during an update/downdate. Dynamic supernodes are introduced which allow a sparse Cholesky update/downdate to obtain performance competitive with conventional supernodal methods. A dynamic supernodal solver is shown to exceed the performance of the conventional (BLAS-based) supernodal method for solving triangular systems. These methods are incorporated into CHOLMOD, a sparse Cholesky factorization and update/downdate package which forms the basis of `x=A\b` in MATLAB when `A` is sparse and symmetric positive definite.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Linear systems (direct methods); sparse and very large systems*; G.4 [**Mathematics of Computing**]: Mathematical Software—*Algorithm analysis; efficiency*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Cholesky factorization, linear equations, sparse matrices

ACM Reference Format:
Davis, T. A. and Hager, W. W. 2009. Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. ACM Trans. Math. Softw. 35, 4, Article 27 (February 2009), 23 pages. DOI = 10.1145/1462173.1462176. http://doi.acm.org/10.1145/1462173.1462176.

This work was supported by the National Science Foundation, under grants 0203270, 0620286, and 0619080.

Authors' addresses: T. A. Davis, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, U.S.A.; email: davis@cise.ufl.edu; W. W. Hager, Department of Mathematics, University of Florida, Gainesville, FL, U.S.A.; email: hager@math.ufl.edu.

1. INTRODUCTION

Given a sparse, symmetric positive definite matrix A with a Cholesky factorization $A = LL^T$ or $A = LDL^T$ and a low-rank modification $\overline{A} = A + WW^T$ (update) or $\overline{A} = A - WW^T$ (downdate), we consider the problem of computing the Cholesky factorization of $\overline{A}$ while exploiting the supernodal structure of its Cholesky factor. Since an update operation changes the supernodal structure, it is not easy to take advantage of the original supernodes. This leads us to develop a new *dynamic supernodal update algorithm*. In the dynamic algorithm, the supernodes are detected and exploited as the update progresses. In a similar fashion, we obtain a *dynamic supernodal solve* in which the supernodes are detected as the solve progresses. Update/downdate problems such as these arise in optimization algorithms, sensitivity analysis, and many other areas [Hager 1989]. The sparse rank-1 update when L is not a supernodal Cholesky factorization is discussed in Davis and Hager [1999], while the multiple rank case is in Davis and Hager [2001]. It is assumed that A has already been permuted by a fill-reducing ordering; this is a large and critical topic in itself which is beyond the scope of this article [Davis 2006].

The *supernodal* Cholesky factorization method [Ashcraft and Grimes 1999; Dobrian et al. 2000; Hénon et al. 2002; Ng and Peyton 1993; Rothberg and Gupta 1991; Rotkin and Toledo 2004] exploits dense matrix kernels during the factorization and solution of the resulting triangular systems. It is based on *supernodes*, which are adjacent columns of L with identical nonzero pattern stored as a single dense submatrix of L. In this article, the supernodal structure of L is exploited in the initial factorization, the low-rank update/downdate, and in the solution of the triangular systems required to solve $Ax = b$ after the Cholesky factorization $A = LL^T$ is computed. As the matrix is updated or downdated, supernodes can merge and split apart. Conventional supernodes cannot be adapted to this problem. In this article we show how these *dynamic* supernodes can be effectively exploited to obtain high performance.

Section 2 provides a brief overview of supernodes in sparse Cholesky factorization and in the solution of triangular systems. Sections 3 and 4 present our update/downdate method and triangular solvers, both of which exploit dynamic supernodes. The performance of our new methods is illustrated in Section 5.

A note regarding notation: When we need to make a distinction between the original matrices A and L and their updated/downdated versions, we use $\overline{A}$ and $\overline{L}$ for the modified versions. Our update/downdate methods modify L and b in place. In those contexts when we refer to L changing, it should be clear that we are dynamically changing L to obtain $\overline{L}$. Thus, L can sometimes refer to both the original and modified Cholesky factorization. The usage should be clear in context.

2. THE SUPERNODAL METHOD

The primary purpose of the supernodal method is to obtain high performance on modern computer architectures with memory hierarchy by exploiting dense

submatrices in the sparse factorization. Improved locality also enables its use on parallel computers, but only sequential algorithms are considered here.

The use of dense matrix kernels (the BLAS [Lawson et al. 1979; Dongarra et al. 1990; 1988]) is a common technique for improving the performance of sparse matrix factorization and the solution of the subsequent triangular systems that are required to solve $Ax = b$ for a general matrix A. Supernodal and multifrontal methods both exploit the nearly-identical sparsity structure often shared by adjacent columns of L (and rows of U in the unsymmetric case), for either LU or Cholesky factorization ($A = LU$ or $A = LL^T$). These methods are able to use the BLAS to obtain a level of performance that is a significant fraction of the computer's theoretical peak performance.

2.1 Finding Supernodes

Informally, a *supernode* is a set of adjacent columns of L that have an identical nonzero pattern. They are typically stored in a way that exploits this structure, such as a dense matrix of dimension s-by-z, where s is the number of nonzeros in the leftmost column in the supernode and z is the number of columns in the supernode. Dense matrix kernels are used to compute each supernode, and to apply each supernode to the righthand side when solving a triangular system.

More precisely, a supernode is defined by a chain of nodes in the elimination tree, and the sparsity pattern of the corresponding columns of L. The *elimination tree* of an n-by-n matrix A is a tree of n nodes [Liu 1990; 1986; Schreiber 1982]. The parent of node j in the tree is given by the first off-diagonal nonzero entry l_{ij} in column j,

$$\text{parent}(j) = \min\{i \mid i > j \text{ and } l_{ij} \neq 0\}. \tag{1}$$

If this set is empty, then node j is a root of the elimination tree. The tree may actually be a forest with more than one root, but it is still conventionally called an elimination tree. Numerical cancellation is ignored, so the term "$l_{ij} \neq 0$" in (1) should be understood to be true for any entry l_{ij} that must be computed during Cholesky factorization; it may occasionally be zero numerically. Otherwise, the definition of the elimination tree breaks down, as do many theorems regarding sparse Cholesky factorization.

Let $\mathcal{L}_j$ denote the nonzero pattern of column j of L. In other words, $\mathcal{L}_j = \{i \mid l_{ij} \neq 0\}$. A *fundamental supernode* is a maximal sequence of z columns $f, f+1, \ldots, f+z-1$ such that for any successive pair of columns $j-1$ and j in the list, $j-1$ is the only child of j, and $\mathcal{L}_j = \mathcal{L}_{j-1} \setminus \{j\}$. In other words, columns in a supernode form a chain in the elimination tree, and have identical nonzero pattern (excluding entries in the upper triangular part). Column f is the first, or leading, column in the supernode. It may have any number of children in the elimination tree. For a *relaxed supernode*, some of the constraints of this definition are loosened; two columns may be placed in the same supernode if their nonzero patterns are similar but not identical, and $j-1$ need not be the only child of j, for example.

Supernodes can be found without constructing the nonzero pattern of L, in time that is essentially linear in the number of nonzeros of A [Liu et al.

1993]. First, the elimination tree of A is computed in nearly $O(|A|)$ time [Liu 1990; 1986].[1] More precisely, the time is $O(|A|\alpha(|A|, n))$ where α is the inverse Ackerman function, a function that grows extremely slowly. Thus in practical terms the time is $O(|A|)$.

Next, the elimination tree is typically reordered via a depth-first postordering, taking $O(n)$ time. In a depth-first postordering, the d descendants of a node j in the elimination tree are all numbered $j - d$ through $j - 1$. The postordering ensures that a node with only one child c is always numbered $c + 1$. This maximizes the sizes of fundamental supernodes in the matrix L. Postordering also improves memory locality during numerical factorization. The postordering is a permutation of A, but has no effect on the the number of nonzeros in L. It thus has no effect on the fill-reducing ordering. In MATLAB, `[parent,q]=etree(A)` computes both the elimination tree (`parent`) and its postordering (`q`).

Once the tree is found and postordered, the number of entries in each column of L is found, using an algorithm that takes nearly $O(|A|)$ time [Gilbert et al. 1994]. In MATLAB, this is computed by the routine `symbfact`. If `count=symbfact(A)`, then `count(j)` = $|\mathcal{L}_j|$. The column counts and the elimination tree are then used to find the fundamental supernodes. Consider the jth column of L. Its nonzero pattern is related to the nonzero patterns of the children of node j in the elimination tree [George and Liu 1981],

$$\mathcal{L}_j = \mathcal{A}_j \cup \{j\} \cup \left(\bigcup_{j=\text{parent}(c)} \mathcal{L}_c \setminus \{c\} \right), \tag{2}$$

where $\mathcal{A}_j$ is the nonzero pattern of the jth column of the strictly lower triangular part of A. A lower bound on the column count of j is thus $|\mathcal{L}_j| \geq |\mathcal{L}_c| - 1$. The following condition defines a fundamental supernode.

CONDITION 2.1. Columns $j-1$ and j are members of the same fundamental supernode if and only if $|\mathcal{L}_j| = |\mathcal{L}_{j-1}| - 1$ and $j - 1$ is the only child of j in the elimination tree.

Thus, supernodes can be found in nearly $O(|A|)$ time without accessing or computing the nonzero pattern of L. Only the elimination tree and the column counts are required. This observation is essential to the dynamic supernodal routines described in Sections 3 and 4. The restriction on $j - 1$ being the only child of j can be relaxed, resulting in larger supernodes.

CONDITION 2.2. Columns $j-1$ and j can be members of the same supernode if $|\mathcal{L}_j| = |\mathcal{L}_{j-1}| - 1$ and $j - 1$ is a child of j in the elimination tree.

2.2 Supernodal Factorization

In the nonsupernodal left-looking Cholesky factorization algorithm, the kth step of factorization computes the kth column of L, accessing columns 1

[1]The number of nonzeros in matrix or vector x, or the size of a set x, is denoted as $|x|$.

through $k-1$ of L and column k of A. Each step consists of a sparse-matrix-vector multiply (in MATLAB notation, `A(k:n,k)-L(k:n,1:k-1)*L(k,1:k-1)'`) followed by a square root and scaling of the kth column of L.

Supernodal Cholesky factorization is a blocked version of the left-looking method, where each block is a supernode. The method can be derived from the expression

$$\begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{bmatrix} = \begin{bmatrix} A_{11} & A_{21}^T & A_{31}^T \\ A_{21} & A_{22} & A_{32}^T \\ A_{31} & A_{32} & A_{33} \end{bmatrix}, \tag{3}$$

where the middle block row and column of each matrix are rows and columns corresponding to the kth supernode. If the columns of L corresponding to the first $k-1$ supernodes are known (L_{11}, L_{21}, and L_{31}), then the kth supernode can be computed, using the following algorithm.

First, a sparse matrix product is performed to initialize the kth supernode.

$$\begin{bmatrix} S_1 \\ S_2 \end{bmatrix} = \begin{bmatrix} A_{22} \\ A_{32} \end{bmatrix} - \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} L_{21}^T \tag{4}$$

The L_{21} and L_{31} matrices split into a set of supernodes. The sparse matrix multiplication is performed one supernode at a time, using a dense matrix multiplication for each supernode. The subtraction in (4) does not use dense matrix operations, since the nonzero patterns of each supernode are different. Instead, a scatter operation is used. Fortunately, most of the floating-point operations are performed in the dense matrix multiply.

Next, the dense Cholesky factorization $S_1 = L_{22}L_{22}^T$ is computed. This is a dense submatrix factorization, since L_{22} is the diagonal block of a single supernode. The nonzero patterns of these columns are all the same, and the subdiagonal of L_{22} is all nonzero in these columns (the columns form a chain in the elimination tree). Thus L_{22} is a dense matrix.

Finally, the triangular system $L_{32}L_{22}^T = S_2$ is solved for L_{32}. The L_{32} matrix is sparse, but each column has the same nonzero pattern, and thus a dense triangular solver is used for this step, with multiple dense right-hand sides (DTRSM when the matrix is real).

Since S_1 and S_2 have the same nonzero pattern that is a subset of the kth supernode, they can be stored in the same place as L_{22} and L_{32}, respectively. The supernode

$$\begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix}$$

is stored in a s-by-z dense matrix, where $s \geq z$ is the number of nonzeros in the first column of the supernode.

CHOLMOD is used by `chol`, `symbfact`, `etree`, `x=A\b` when `A` is sparse symmetric positive definite in MATLAB 7.2. The performance of CHOLMOD and 10 other sparse factorization packages is discussed by Gould et al. [2007]:

> It appears that the newest code CHOLMOD offers the best balance between the three solution phases and so gives the best overall performance. . . . CHOLMOD generally requires the least memory.

The three phases consist of: (1) ordering and analysis, (2) factorization, and (3) forward and backsolve. Scott and Hu also discuss the design considerations for a sparse direct code [Scott and Hu 2007], and give details on the user interface and design of CHOLMOD.

2.3 Supernodal Solve

Consider the triangular system $Lx = b$,

$$\begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \tag{5}$$

where L is partitioned the same as in (3), and x is a dense vector. In the forward solve, the kth step requires the solution of a dense lower triangular system $L_{22}x_2 = b_2 - L_{21}x_1$, where $b_2 - L_{21}x_1$ is computed first. Next, $L_{32}x_2$ is subtracted from the right-hand side, requiring a dense matrix-vector multiplication and a sparse gather/scatter operation (since L_{32} is the subdiagonal part of the k supernode). Most of the work is thus performed with dense matrix kernels (the level-2 BLAS). Matrix-matrix operations are used if x is a dense matrix rather than a vector.

This method is used in `x=A\b` in MATLAB 7.2 when `A` is sparse and symmetric positive definite, as implemented in CHOLMOD. It is not used in `x=L\b` when `L` is lower triangular, since `L` is not stored in supernodal form (even if `L` comes from a supernodal sparse Cholesky factorization, `L=chol(A)`). Performance comparisons with other triangular solvers are given in Gould et al. [2007].

3. UPDATING A SPARSE CHOLESKY FACTORIZATION

We first review our nonsupernodal update/downdate method in Section 3.1 from our prior work [Davis and Hager 2001; 1999]. Next, Section 3.2 discusses the difficulties encountered in updating/downdating a conventional supernodal factorization, particularly when the nonzero pattern of L changes. These difficulties are resolved in Section 3.3, which presents a dynamic supernodal update/downdate that allows the nonzero pattern of L to change, while still exploiting supernodes. Note that while MATLAB 7.2 includes CHOLMOD's supernodal factorization, it does not include the dynamic supernodal update/downdate described next.

3.1 Nonsupernodal Update/Downdate

Consider the rank-1 update/downdate, $\overline{A} = A \pm ww^T$ where w is a sparse column vector. If the Cholesky factorization LL^T of A is known (or $A = LDL^T$, where D is diagonal), the factorization of the modified matrix $\overline{A}$ can be found in time proportional to the number of entries in L that change [Davis and Hager 1999]. This includes the time required to modify the nonzero pattern of L, if the pattern needs to change. For additional background on the update/downdate problem, see (for example) Bischof et al. [1993], Carlson [1973], Gill et al. [1974], Stewart [1998; 1979], Pan [1990]. A simple rank-1

update/downdate of a sparse LL^T factorization that does not change the nonzero pattern of L is discussed in detail in Davis [2006]; that algorithm is a mere 35 lines of C.

The rank-1 update/downdate that allows for changes in the nonzero pattern of L is discussed in Davis and Hager [1999]. During an update, $\overline{A} = A + ww^T$, no entries are removed from L; entries are only added. During a downdate, $\overline{A} = A - ww^T$, entries could be dropped (but not added) if $A = CC^T$ and w is one of the columns of C.

The time taken for our rank-1 update is exactly proportional to the number of entries in L that change, which can be very small ($O(1)$ even). This time includes the time to update/downdate the pattern and numerical values of L, and the time to update/downdate the elimination tree. For a rank-k update, the time is bounded by the time for k rank-1 updates/downdates, except that we save time and reduce memory traffic by reading and writing just once each column of L that changes. Thus, our update/downdate methods are asymptotically optimal, even when new entries get added in an update, or old entries are removed in a downdate. However, we do not exploit supernodes in our prior work [Davis and Hager 2001; 1999].

For either an update or a downdate, the entries that change in L correspond to a single path in the elimination tree of the modified matrix $\overline{A}$. The path starts from the node i corresponding to the smallest row index of nonzero entries in w, and proceeds upwards, ending at the root of the tree. For a multiple rank update/downdate ($\overline{A} = A \pm WW^T$, where W is n-by-k), the columns that change correspond to a set of paths in the elimination tree of $\overline{A}$, each starting at the node corresponding to the smallest row index of nonzero entries in each column of W [Davis and Hager 2001]. Paths that merge as they proceed upwards toward the root result in a rank-k update of the columns along that path, where in this case k is the number of paths from columns of W that have merged.

The driving motivation of our supernodal update/downdate method presented in this article is an active-set linear programming method (the LP Dual Active Set Algorithm, LPDASA [Davis and Hager 2008a; 2008b]), where C is the matrix comprised of the columns corresponding to the active variables. Dropping entries requires the nonzero pattern of L to be held as a multiset, with multiplicities for each entry in the set. Downdating the nonzero pattern of L can be done in perfect asymptotic time, but it requires additional memory to hold each column of L as a multiset. This extra information is costly for a general application, and is not need in LPDASA. It is more memory-efficient to retain all the nonzeros during a downdate rather than try to determine which nonzero should be zero after the downdate.

Thus, in this article (and in CHOLMOD), the storage structure for L is only modified to account for the creation of new nonzeros during the update/downdate. We assume that either an update or downdate can add entries to $\overline{A}$ and thus also to its updated/downdated Cholesky factor. Neither the update nor the downdate are assumed to remove any entries. Entries which should be numerically zero after a downdate are retained in storage. These entries can be dropped later by redoing the symbolic factorization, if desired, in $O(|L|)$

time (CHOLMOD provides a function to perform this operation, which we refer to as *symbolic refactorization*).

3.2 Difficulties in a Static Supernodal Update/Downdate

If supernodes are exploited, better performance could be obtained, in much the same way as supernodes can improve the performance of sparse Cholesky factorization and solves. However, adding entries to a supernodal factorization is problematic. A single update/downdate can cause some supernodes to merge and others to split apart. If two adjacent columns $j-1$ and j of L are not in the same supernode, an update/downdate could add entries to $j-1$ so that now these two columns have the same nonzero pattern, causing them to merge into one fundamental supernode. Similarly, if the two columns are identical, an update/downdate could add entries to column j but not to $j-1$, causing the supernode to split apart.

Supernodes are conventionally stored in a dense s-by-z matrix. Modifying this structure would lead to an algorithm whose time complexity could be far from optimal. Suppose an update/downdate modifies only the last column of the supernode, causing the supernode to split. The time required to modify the numerical values in the supernode would be $O(s-z)$, but $O(sz)$ time would be required for the data movement that splits the supernode in two. This is not a viable solution. Thus, supernodes were not considered in Davis and Hager [2001; 1999].

It would be simpler to assume, as in CSparse (a concise sparse matrix package written for a textbook [Davis 2006]), that the nonzero pattern of a supernodal factor L does not change. The matrix L could be kept in its supernodal form, and the update/downdate could then exploit this structure to obtain higher performance than a nonsupernodal update/downdate. The structure of L, and its supernodes, would be static. However, this requirement is too limiting for many applications. In particular, it would not be suitable in our motivating application, LPDASA.

3.3 Dynamic Supernodal Update/Downdate

Our goal is a rank-k update/downdate method that simultaneously exploits supernodes and allows the nonzero pattern of L to change. The solution is to not store supernodes in their conventional form.

Instead, the matrix L is stored in a conventional nonsupernodal compressed sparse column form, where each column of L is stored as a list of numerical values of the nonzero entries, and an integer list of the corresponding row indices. MATLAB uses a similar data structure, except that to allow for columns to grow and shrink, we allow for gaps between the columns of unused memory space in our data structure. We also allow columns to appear in any order in memory. Supernodes are detected dynamically as the update/downdate progresses through the matrix.

The update/downdate is split into two phases: symbolic and numeric. The symbolic update is identical to that in Davis and Hager [2001], except that multiplicities are not kept. The runtime of the symbolic phase is always

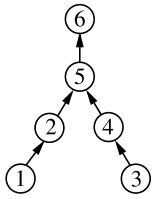

Fig. 1. Example elimination tree.

asymptotically dominated by the numeric update; it is much less if the pattern does not change or changes only very little. The numeric update proceeds along a series of disjoint subpaths, each of which computes an update of rank anywhere from 1 to k. As it proceeds, it detects supernodes dynamically. If columns j_1 and j_2 are adjacent columns on the same subpath, then j_1 is a child of j_2. If in addition, the number of nonzeros in column j_1 is one more than the number of nonzeros in column j_2, then j_1 and j_2 lie in the same dynamic supernode. This test takes $O(1)$ time, and can be done without examining the nonzero patterns of the two columns. This is a relaxation of the restriction that $j-1$ and j be adjacent in the matrix. Consider the small elimination tree in Figure 1. Suppose the update path starts at node 1, and that columns 3 and 4 are not updated. Columns 1, 2, 5, and 6 could all be part of the same dynamic supernode.

CONDITION 3.1. Columns c and j can be members of the same dynamic supernode if $|\mathcal{L}_j| = |\mathcal{L}_c| - 1$, c is a child of j in the elimination tree, and c and j are adjacent nodes in a disjoint subpath of the subtree of columns modified by the update/downdate.

Columns can be added to a dynamic supernode by looking ahead along the path and stopping when Condition 3.1 is broken. For the triangular solve of $L^T x = b$ and $Lx = b$, respectively, Condition 2.2 is used when b is dense, because all columns of L take part in the solve.

To understand how the update/downdate algorithm operates on a supernode, we first examine how the update/downdate algorithm operates on a dense matrix.

Consider the method in Algorithm 1 for updating/downdating a dense LDL^T factorization (a modified version of Method C1 in Gill et al. [1974]). It computes the new factorization $\overline{LDL}^T = LDL^T \pm WW^T$, where W is n-by-k. The σ term is equal to +1 for an update or -1 for a downdate. Many other methods are possible (see Gill et al. [1974], for example). They all include an innermost loop in which W and a column of L are used to modify each other.

An update of the LL^T factorization is nearly identical. For example, the `cs_updown` function in CSparse [Davis 2006] is based on Bischof, Pan, and Tang's [Bischof et al. 1993] combination of Carlson's update [Carlson 1973] and Pan's downdate [Pan 1990]. All of these methods modify the LL^T factorization instead, as does Stewart's method in LINPACK [Dongarra et al. 1978; Stewart 1998; 1979] (the method used by `cholupdate` in MATLAB). To update

Algorithm 1. (Dense Rank-k Update/Downdate).

```
for r = 1 to k do
    α_r = 1
for j = 1 to n do
    for r = 1 to k do
        ᾱ = α_r + σ w_jr^2 / d_j
        d_j = d_j ᾱ
        γ_r = −σ w_jr / d_j
        d_j = d_j / α_r
        α_r = ᾱ
    for i = j + 1 to n do
        for r = 1 to k do
            w_ir = w_ir − w_jr l_ij
            l_ij = l_ij − γ_r w_ir
```

LL^T factorization, the innermost loop requires 5 floating-point operations instead of 4 for the LDL^T case in Algorithm 1. The memory traffic is identical, however, since the extra term is a scalar which would be held in registers or cache.

In the sparse update, the **for** j loop in Algorithm 1 is replaced by a loop that iterates over a single disjoint subpath in the elimination tree [Davis and Hager 2001]. Each column L_{*j} along this path is modified by W, and likewise modifies the matrix W. The row index i is restricted to the nonzeros in column L_{*j}, so that the **for** $i = j + 1$ **to** n loop is replaced with a loop in the form **for** $i \in \mathcal{L}_j$.

It is important to note that the innermost loop accesses an entire row of W; it is thus most efficient to store W in row-major form, as a scattered (full) matrix. This of course limits the maximum rank update by the available memory space. In CHOLMOD, the sparse W is scattered into a full workspace matrix with n rows and at most 8 columns. We can thus perform up to rank-8 updates at a time; larger rank updates are done in groups of at most 8.

For a rank-1 update/downdate, there is just one path. Let $t_1 = \min \mathcal{W}$ be the smallest row index of nonzero entries in the column w, where $\mathcal{W}$ denotes the nonzero pattern of w. The columns modified by the rank-1 update consist of all nodes in the path from node t_1 to the root in the elimination tree of $\overline{L}$.

For the rank-k case, each rank-1 update/downdate results in a single path. Together, these paths form a subtree of the elimination tree, the leaves of which are t_1 through t_k. We split this subtree into a set of disjoint subpaths.

Now consider the dynamic supernodal case. We wish to group together adjacent columns along one of these disjoint subpaths, and thus we apply Algorithm 1 to all columns j along each subpath of the subtree of columns to update.

The dynamic supernodal algorithm exploits three cases: supernodes consisting of one, two, or four columns of L. Our algorithm easily extends to supernodes of any size, but we limited the actual size of supernodes for performance reasons. A "supernode" of size one is just a single column of L. We attempted to exploit dynamic supernodes of size three, but found that we obtained better

performance by treating a three-column supernode as one supernode of two columns and one supernode of one column. Best performance is obtained in a supernode of four columns. Supernodes of five columns or greater also did not improve performance any further, and only led to lengthier code to exploit them. Thus, a large supernode of five or more columns is handled as a sequence of four-column supernodes, followed by a "cleanup" step with a supernode of one and/or two columns.

Two critical factors impact the performance of any numerical method on a high-performance computer: (1) the number of floating-point operations per memory reference, and (2) how regular or irregular the memory references are. Next we present each of the three different updates (1-column, 2-column, and 4-column), and analyze their flop count and memory access behavior.

For simplicity, this discussion assumes j, $j+1$, $j+2$, and $j+3$ are the successive columns along an update/downdate subpath. In general, they need not be adjacent in L to be considered part of same dynamic supernode (see Condition 3.1).

3.3.1 *Nonsupernodal Rank-k Update.* Suppose the algorithm is at column j. If column j and $j+1$ are not in the same dynamic supernode, the single column j is updated. If columns j through $j+3$ all lie within the same supernode, then a dynamic supernode of four columns is updated. Otherwise, if j and $j+1$ reside in the same dynamic supernode, then a dynamic supernode of two columns is updated.

Consider the update of a single column of L. This method corresponds to the rank-k update given in Davis and Hager [2001], and a single iteration of the **for** j loop in Algorithm 1. The loop across the rows i of the column iterates instead over the rows i corresponding to nonzero entries in column j. This loop is blocked, so that every iteration updates four nonzeros in column j at a time. The r loops are unchanged.

Let $s = |\mathcal{L}_j|$ be the number of entries in the jth column of L. The single-column rank-k update performs $4sk$ floating-point operations, and $2sk + 3s$ memory references. For the matrix W, sk entries are read, modified, and written back, accounting for $2sk$ memory references. The numerical values and nonzero pattern of L_{*j} are read ($2s$ references). The numerical values of L_{*j} are then written back (another s references).

3.3.2 2-*Column Supernodal Rank-k Update.* Consider an update of columns j and $j+1$. In this case, the nonzero pattern of column $j+1$ need not be accessed. First, the r loop computes the α and γ terms for the jth column, and modifies the diagonal d_j. Next, the single subdiagonal entry $L_{j+1,j}$ is updated. The r loop can now proceed for column $j+1$, computing the α and γ terms for that column, and updating the $j+1$st diagonal entry.

The 2-column update is given in Algorithm 2, which is the 2-column supernodal version of the last four statements of Algorithm 1. Let $s = |\mathcal{L}_j|$. Algorithm 2 is used when the condition $|\mathcal{L}_{j+1}| = s - 1$ is true, and either $|\mathcal{L}_{j+2}| = s - 2$ or $|\mathcal{L}_{j+3}| = s - 3$ are false (if all four conditions are true, columns j through $j+3$ form a 4-column dynamic supernode). Algorithm 2 as presented

Algorithm 2. (Supernodal Rank-k Update/Downdate of Columns j and $j+1$ of L).

for each adjacent pair i_1, i_2 in $\mathcal{L}_j \setminus \{j, j+1\}$ **do**
  copy entries of L into a 2-by-2 dense matrix:
  $x_{11} = l_{i_1,j}$
  $x_{21} = l_{i_2,j}$
  $x_{12} = l_{i_1,j+1}$
  $x_{22} = l_{i_2,j+1}$
  rank-k update of x:
  for $r = 1$ **to** k **do**
    gather two entries of W into a 2-by-1 vector t:
    $t_1 = w_{i_1,r}$
    $t_2 = w_{i_2,r}$
    update two entries in column j:
    $t_1 = t_1 - w_{jr} x_{11}$
    $t_2 = t_2 - w_{jr} x_{21}$
    $x_{11} = x_{11} - \gamma_{r1} t_1$
    $x_{21} = x_{21} - \gamma_{r1} t_2$
    update two entries in column $j+1$:
    $t_1 = t_1 - w_{j+1,r} x_{12}$
    $t_2 = t_2 - w_{j+1,r} x_{22}$
    $x_{12} = x_{12} - \gamma_{r2} t_1$
    $x_{22} = x_{22} - \gamma_{r2} t_2$
    scatter t back into W:
    $w_{i_1,r} = t_1$
    $w_{i_2,r} = t_2$
  copy x back into L:
  $l_{i_1,j} = x_{11}$
  $l_{i_2,j} = x_{21}$
  $l_{i_1,j+1} = x_{22}$
  $l_{i_2,j+1} = x_{22}$

assumes that there are an even number of nonzero entries below the diagonal in column $j+1$. If there are an odd number, the outermost **for** loop is proceed by an iteration that handles the first off-diagonal entry i in column $j+1$ of L, which is not shown.

Assume W is stored in row-major order, in scattered form. Specifically, it is stored as a dense row-major n-by-k matrix. This limits the algorithm to handling updates with a modest number of columns k. Since W is stored in row-major order, the access of w_{jr} and $w_{j+1,r}$ as r varies is very efficient. These entries will be cached, since there are only $2r$ of them. The arrays t and x can be held in registers. The access of the 2-by-2 block of L is efficient, for two reasons. First, the terms are accessed only once, regardless of k. Second, since the columns of L are kept sorted (with row indices in strictly ascending order), entries in rows i_1 and i_2 of L_{*j} are adjacent in memory.

The two columns j and $j+1$ of L are not stored in a single dense submatrix, as they would be in a true supernodal factor. However, in both the nonsupernodal data structure for L and in the supernodal L, the two entries $l_{i,j}$ and

$l_{i,j+1}$ would not be adjacent anyway. In the former case, it is likely that they will be nearby. In the latter case, they will reside a distance of exactly $s = |\mathcal{L}_f|$ entries away, in the same dense submatrix, where f is the first column in the supernode. The impact on performance of using a nonsupernodal data structure instead of a conventional supernodal data structure is thus slight. The nonzero pattern of $\mathcal{L}_j$ needs to be read only once, not twice, and its access is also independent of k.

Performing a rank-k update of two columns of L requires about $8sk$ floating-point operations ($s/2$ outer iterations, with $16k$ operations each), excluding the $16k$ flops to compute the α and γ coefficients. Excluding cached variables (t, x, γ, w_{jr}, and $w_{j+1,r}$), four entries of L are read and written for each $s/2$ outer iteration, and two entries of W are read and written for each $k \times s/2$ inner iteration. The integer pattern of $\mathcal{L}$ is read in just once. The total number of memory reads is $s(k + 3)$, and there are $s(k + 2)$ writes.

Most memory traffic is regular, since W is stored in row-major order and since L is stored by column. The only irregular access is the gather of the first $w_{i_1,1}$ and $w_{i_2,1}$ entries; for subsequent r, the memory traffic is regular. Likewise, only the access to the first entry in each column of L is irregular; the subsequent ones are all stride-1 accesses. This is essentially the same as accessing two columns of a dense matrix, which would be the case if L is stored in supernodal form.

The number of floating-point operations per memory reference in Algorithm 2 is thus $8sk/(2sk + 5s)$. Our code handles the case for $k = 1$ to $k = 8$; larger rank updates are split into updates where W has 8 columns or less. Since k is limited to a small constant, the r loop is completely unrolled, and eight different versions of the function are created by the compiler.

By comparison, a level-2 BLAS operation (n-by-n dense matrix times dense vector) has a flops-per-memory-access ratio of about 2, or about the same as a rank-3 2-column update.

The memory traffic to update column j and $j+1$ separately is almost double, since sk entries of W must be accessed twice. These entries are accessed only once in a 2-column update in Algorithm 2.

3.3.3 4-*Column Supernodal Rank-k Update.* This idea can be extended to more than two columns of L since supernodes are usually much larger. With four columns, an analogous algorithm that updates a 1-by-4 block of L in the innermost loop performs $16sk$ floating-point operations, $s(k + 5)$ memory reads, and $s(k + 4)$ memory writes.

Just as in the two-column rank-k update/downdate, the first step is to update the triangular part of the four-column supernode. This step computes the four α coefficients and the k-by-4 γ coefficients for the rank-k update of the 4-by-4 triangular part of the supernode, using Algorithm 1. It then performs the following update for all rows $i > j + 3$ in each of the four columns, using Algorithm 3.

Assuming again that the α and γ coefficients are always kept in cache, the flops-per-memory-access ratio becomes $16sk/(2sk + 9s)$. The flop-per-memory ratios for nonsupernodal, 2-column supernodal, and 4-column supernodal

Algorithm 3. (Supernodal Rank-k Update/Downdate of Four Columns j to $j+3$ of L).

for i in $\mathcal{L}_j \setminus \{j : j+3\}$ **do**
  copy entries of L into a 1-by-4 dense matrix x:
  $x = l_{i,j:j+3}$
  copy entries of W into a 4-by-k dense matrix z:
  $z = w_{j:j+3,1:k}$
  rank-k update of x:
  for $r = 1$ **to** k **do**
    update one entry in column j:
    $w_{ir} = w_{ir} - z_{1r}x_1$
    $x_1 = x_1 - \gamma_{r1}w_{ir}$
    update one entry in column $j+1$:
    $w_{ir} = w_{ir} - z_{2r}x_2$
    $x_2 = x_2 - \gamma_{r2}w_{ir}$
    update one entry in column $j+2$:
    $w_{ir} = w_{ir} - z_{3r}x_3$
    $x_3 = x_3 - \gamma_{r3}w_{ir}$
    update one entry in column $j+3$:
    $w_{ir} = w_{ir} - z_{4r}x_4$
    $x_4 = x_4 - \gamma_{r4}w_{ir}$
  copy x back into L:
  $l_{i,j:j+3} = x$

Table I. Flops per Memory Access for Rank-k Update

rank-k update:	$k=1$	2	3	4	5	6	7	8
nonsupernodal	4/5	8/7	12/9	16/11	20/13	24/15	28/17	32/19
$(4sk/(2sk+3s))$	0.80	1.14	1.33	1.45	1.53	1.60	1.65	1.68
2-column supernode	8/7	16/9	24/11	32/13	40/15	48/17	56/19	64/21
$(8sk/(2sk+5s))$	1.14	1.78	2.18	2.46	2.67	2.82	2.95	3.05
4-column supernode	16/11	32/13	48/15	64/17	80/19	96/21	112/23	128/25
$(16sk/(2sk+9s))$	1.45	2.46	3.20	3.76	4.21	4.57	4.87	5.12

updates are shown in Table I for each value of k. Each is listed as both a ratio and a decimal value.

The peak ratio for a 4-column rank-8 update is 128/25, or 5.12. In a matrix with many large supernodes, most of the work will be done in updates of rank-k to dynamic supernodes of size 4. Our method thus uses one of three updates: single-column with 4-by-1 updates in the innermost loop, 2-column with 2-by-2 updates of L, and 4-column with 1-by-4 updates of L. We can thus expect a rank-8 update of a matrix with many large supernodes to rival or exceed the performance of a BLAS matrix-vector multiply (which has a flops/memory reference ratio of 2).

3.3.4 *Gap Strategy.* During an update, the number of entries in a column can increase. In a MATLAB-style compressed column data structure, there are no gaps between the columns to allow for this growth. In our data structure, we allow for gaps to appear. It would be possible to use the C memory allocator

(`malloc`) to allocate and realllocte (`realloc`) each column, but this would be slow and memory-inefficient. Instead, we allocate one block of memory for all the columns and manage the gaps ourselves.

These gaps arise naturally when a supernodal factorization is converted into a "simplicial" one (where the nonzero pattern of each column of L is stored separately), because we do not move the numerical values. A supernode of size s-by-z is a lower triangular dense matrix. The upper triangular part is not used. Thus, when converted into a simplicial L, the first column in the supernode is followed by a gap of one entry. The second has a gap of two. The next-to-last column has a gap of $z - 1$. The last column has no gap, since the next entry in L is the first entry of the next supernode.

These gaps are used by our dynamic update to hold new entries that appear. If additional entries appear, then a column is removed from its place and stored at the tail end of the memory space used for L. If the new column j requires c entries after the update, space of $\min(\lfloor 1.2c \rfloor + 5, n - j)$ entries is given to it to allow for potential future growth.

4. DYNAMIC SUPERNODAL SOLVE

The dynamic supernodal update/downdate has a similar structure as the algorithm for solving a sparse lower triangular system $Lx = b$. The latter is simpler, since we only consider the case where b is a dense vector or matrix. It accesses columns 1 through n, and can exploit the same dynamic supernode strategy. Our method looks for dynamic supernodes of size one to three columns, and can handle one to four righthand sides (b can be an n-by-4 dense matrix). The solution x is stored in row-major form.

We limit the size of the dynamic supernode to three columns for performance reasons. We found that attempting to exploit supernodes of size four columns or larger resulted in lower performance in our dynamic supernodal triangular solve. This is analogous to the innermost blocking used in the BLAS, which typically relies on completely unrolled loops that operate on 2-by-2, 2-by-4, or other small submatrices of similar sizes.

With four right-hand sides, and a dynamic supernode of three columns of L, the triangular solve for these three columns is given in Algorithm 4. Each inner iteration multiplies a dense 1-by-3 vector with a 3-by-4 matrix.

Just as in the dynamic supernodal update/downdate, the nonzero patterns of columns $j + 1$ and $j + 2$ are not accessed. The access of the first entry x_{ij} is irregular, but access to entries in subsequent columns is regular, since we store X in row-major form. The matrix y is only of size 3-by-4 and can be stored in cache or registers. Thus, each inner iteration performs 24 floating-point operations, reads 3 entries of L, one entry of $\mathcal{L}$, and reads/writes 4 entries of X. The flops-per-memory-access ratio is thus 24/12, or 2. A conventional supernodal solve has a similar ratio.

With a single righthand side and 3 columns in a dynamic supernode, each inner iteration performs 6 floating-point operations, reads 3 entries of L, one entry of $\mathcal{L}$, and reads/writes one entry of x. In this case the ratio is 1 (6 flops and 6 memory references).

Algorithm 4. (Supernodal Solve (3 Columns of L and 4 Righthand Sides).

solve $L_{j:j+2,j:j+2}y = X_{j:j+2,*}$ for y
$X_{j:j+2,*} = y$
for each i in $\mathcal{L}_j \setminus \{j, j+1, j+2\}$
$X_{i,*} = X_{i,*} - L_{i,j:j+2}y$

By comparison, a simple lower triangular solver for a dense matrix (one righthand side, and no supernodes) performs 2 floating-point operations in its innermost loop, reads one entry of L, and reads/writes one entry of x. The flops-per-memory ratio is 2/3. In the sparse case, the ratio drops to 2/4 because $\mathcal{L}$ must also be read.

For a dense L and a single right-hand side, most of the work can be done in a matrix-vector multiplication, which has a flops-per-memory-access ratio of about 2.

5. RESULTS

To test our methods, we compared the nonsupernodal multiple-rank update in Davis and Hager [2001] with our new dynamic supernodal update in CHOLMOD. The nonsupernodal multiple-rank update/downdate was done in a modified version of CHOLMOD in which the dynamic detection of supernodes was disabled. The CSparse package [Davis 2006] includes a simple rank-1 update/downdate method that does not check for, nor allow, any changes in the nonzero pattern of L.

We also compared our dynamic supernodal triangular solvers with a simple sparse triangular solver and a conventional supernodal triangular solver.

These results were obtained on a Intel Pentium 4 (3.2GHz clock frequency, 4GB RAM (DDR 333 Mhz), 512KB cache, an 800 MHz memory bus, and running Linux). The theoretical peak performance of the computer is 6.4GFlops. The `gcc` compiler was used (version 3.3.5, with `-O3` optimization). All timings are in seconds of elapsed time.

5.1 Dynamic Supernodal Update/Downdate

Four matrices were used for the tests presented next.

5.1.1 *The ND/ND3k Matrix.* For this matrix, the update $LDL^T + WW^T$ was selected so that the nonzero pattern of L did not change, to allow comparisons with CSparse. The update was computed in 17 different ways. We used CHOLMOD in steps of 1 to 8 columns of W at a time, and both with and without dynamic supernodes. We also used CSparse, but with only one column of W at a time since CSparse can only perform rank-1 updates/downdates. The matrix statistics and results are shown in Table II.

The rank-1 Cholesky update/downdate method in CSparse uses a sparse LL^T factorization and thus performs about 25% more floating-point work as an LDL^T update. It requires the same memory traffic. Since it only applies to problems where the nonzero pattern of L does not change, it does not need to check for possible changes in the pattern. Thus, for a rank-1 update, its

Table II. The ND/ND3k Matrix, Statistics and Results

Matrix name:	ND/ND3K
source:	3D discretization of a PDE
n:	9000
$\|A\|$, lower triangular part:	1.6×10^6
ordering method:	CHOLMOD nested dissection
ordering time:	2.0 seconds
CHOLMOD symbolic Cholesky factorization:	0.14 seconds
CHOLMOD numeric Cholesky factorization:	6.75 seconds
time to convert to nonsupernodal LDL^T:	0.25 seconds
$\|L\|$:	12.6×10^6
Cholesky factorization flop count:	22.15×10^9
CHOLMOD Cholesky factorization Mflops:	3281
update/downdate rank:	128
# of entries modified in L:	12.1×10^6
CSparse update time:	3.29 seconds
CSparse update flop count:	2.2×10^9
CSparse update Mflops:	667
CHOLMOD update flop count:	1.81×10^9

Table III. ND/nd3k: CSparse and CHOLMOD Results (fixed nonzero pattern)

	CSparse nonsupernodal		CHOLMOD nonsupernodal		CHOLMOD supernodal		
rank	time	Mflops	time	Mflops	time	Mflops	speedup
1	3.29	667	3.95	458	3.32	554	19%
2	-	-	3.09	585	2.53	715	22%
3	-	-	2.88	628	2.23	811	29%
4	-	-	2.64	685	2.02	895	31%
5	-	-	2.65	682	1.82	993	46%
6	-	-	2.46	735	1.65	1096	49%
7	-	-	2.34	769	1.54	1174	52%
8	-	-	2.24	807	1.52	1189	47%

performance in terms of runtime and MFlops can exceed CHOLMOD when the pattern does not change.

The results in Table III show that using dynamic supernodes increases the performance of CHOLMOD's sparse Cholesky update/downdate by about 50% (the speedup, in the rightmost column) when the rank of W is 5 or greater.

5.1.2 *The Qaplib/lp_nug15 Matrix.* The second problem was selected to test a changing nonzero pattern. The matrix A is 6330-by-22275; 6330 columns were chosen at random to obtain A_F. The matrix $S = A_F A_F^T + \beta I$ was factorized, and then a rank-128 update was selected at random from the columns in A but not in A_F. This procedure mimics the use of CHOLMOD in a linear programming solver, LPDASA [Davis and Hager 2008a; 2008b]. The matrix statistics and basic results are shown in Table IV.

In Table V, we present the results with just CHOLMOD (both nonsupernodal and dynamic supernodal) where the nonzero pattern is changing. Of the 67,372 times that a column was modified in the 16 rank-8 updates (the last row of Table V), the nonzero pattern of a column was modified 15,396 times (23%, all of which grew in size since entries are not dropped). Although a

Table IV. The Qaplib/lp_nug15 Matrix, Statistics and Results

Matrix name:	QAPLIB/LP_NUG15
source:	linear programming problem
n:	6330
$\|S\|$, lower triangular part:	129×10^3
ordering method:	CHOLMOD nested dissection
ordering time:	0.58 seconds
CHOLMOD symbolic Cholesky factorization:	0.02 seconds
CHOLMOD numeric Cholesky factorization:	5.89 seconds
time to convert to nonsupernodal LDL^T:	0.14 seconds
initial $\|L\|$:	7.57×10^6
Cholesky factorization flop count:	16.4×10^9
Cholesky factorization Mflops:	2684
update/downdate rank:	128
# of entries modified in L:	7.56×10^6
final $\|L\|$:	7.61×10^6
CHOLMOD update flop count:	2.5×10^9

Table V. Qaplib/lp_nug15: CHOLMOD Results (changing nonzero pattern)

	CHOLMOD nonsupernodal		CHOLMOD supernodal		
rank	time	Mflops	time	Mflops	speedup
1	6.81	372	6.08	416	12%
2	5.14	492	4.40	575	17%
3	4.78	529	3.97	637	20%
4	4.47	566	3.69	686	21%
5	4.44	570	3.47	729	28%
6	4.25	596	3.30	767	29%
7	4.11	616	3.24	781	27%
8	4.05	625	3.24	781	25%

column grew in length 23% of the time, the gap we create is typically sufficient to accommodate the growth. When the original gaps between the columns, that arose when the supernodal factor was converted in place to a simplicial factor, were left in place (the default), 1,904 columns had to be shifted (3%). When all gaps were removed initially, 2,320 columns had to be shifted, incurring an additional runtime cost of only 0.22 seconds above the default method, an increase in runtime of only 7%. These results show that our strategy of creating gaps between columns is effective.

The results in Table V cannot be directly compared with CSparse, since the pattern of L is changing. However, if the update is followed by another one with the same W, then the nonzero pattern of L remains unchanged. Almost the same amount of floating-point work is required. The results for both CHOLMOD (with and without supernodes) and CSparse are listed in Table VI.

The differences between Tables V and VI reflect the work required to modify the nonzero pattern $\mathcal{L}$. Modifying the nonzero pattern in a supernodal rank-8 update takes 3.24 seconds, as shown in Table V; the time drops to 1.88 seconds in Table VI. Thus, the time taken to modify the pattern is about 1.36 seconds to the total runtime.

Table VI. Qaplib/lp_nug15: CSparse and CHOLMOD Results (fixed nonzero pattern)

	CSparse nonsupernodal		CHOLMOD nonsupernodal		CHOLMOD supernodal		
rank	time	Mflops	time	Mflops	time	Mflops	speedup
1	4.65	680	5.32	476	4.57	555	16%
2	-	-	3.69	687	2.94	863	25%
3	-	-	3.33	762	2.54	999	31%
4	-	-	3.03	837	2.23	1137	36%
5	-	-	3.03	837	2.05	1237	48%
6	-	-	2.85	890	1.92	1321	48%
7	-	-	2.75	922	1.87	1357	47%
8	-	-	2.70	939	1.88	1349	44%

5.1.3 *A Dense Matrix.* The next matrix we tested was a randomly generated dense symmetric positive definite matrix of dimension $n = 3000$, to test how close CHOLMOD comes to its theoretical performance limit.

The `chol` function in MATLAB takes 2.3 seconds to factorize this matrix, a rate of 3.9GFlops using LAPACK [Anderson et al. 1999] and the Goto BLAS [Goto and van de Geijn 2002]. If that same matrix is converted into a sparse matrix, CHOLMOD requires 3.1 seconds to factorize it (2.9GFlops), excluding the ordering time but including both symbolic and numeric factorization. A rank-1 dense update using `cholupdate` in MATLAB (based on LINPACK [Dongarra et al. 1978; Stewart 1979]) takes 0.2 seconds. The same update using CSparse takes 0.12 seconds (0.03 seconds for the update, and 0.10 seconds to copy the result back to MATLAB). A rank-8 LDL^T update in CHOLMOD takes 0.2 seconds, half of which is the actual update, and the other half being the time to copy the matrix to and from the MATLAB workspace. The copy is required simply because a MATLAB function cannot modify its inputs. Even with the copy (which `cholupdate` must also perform), our sparse update/downdate methods (CSparse and CHOLMOD) are faster than the rank-1 dense update/downdate in MATLAB when operating on the same matrix.

For comparison, the dense matrix-vector multiply (DGEMV) in the Goto BLAS has a peak performance of 2.7GFlops for computing $y = Ax + y$ when A is n-by-n and assumed to already be stored in cache if it is small enough to fit. DGEMV reaches this peak when $n = 160$, but drops when n is about 256 or larger because of cache effects (637MFlops for $n = 500$, and 597MFlops when $n = 1000$, for example). This range of performance is comparable to the sparse Cholesky update/downdate, because L itself is normally too large to fit into cache. The n-by-k workspace W for the update/downdate will typically be too large to fit into cache. The peak performance of the rank-8 update is 1349, which is 2.3 times the performance of the dense matrix-vector multiply for large n. This is very close to the expected ratio of 2.56, based on the difference in flops per memory reference (2 in DGEMV, 5.12 in CHOLMOD's rank-8 update). The entries in the matrix L are read and written just once by our method. Additional performance can only be obtained by an algorithm that keeps more of W in cache than our method does.

Table VII. Banded Matrix, CSparse and CHOLMOD Results (fixed nonzero pattern)

	CSparse nonsupernodal		CHOLMOD nonsupernodal		CHOLMOD supernodal		
rank	time	Mflops	time	Mflops	time	Mflops	speedup
1	0.77	456	0.75	468	0.65	540	15%
2	-	-	0.54	650	0.54	650	0%
3	-	-	0.45	780	0.41	856	10%
4	-	-	0.41	856	0.36	957	14%
5	-	-	0.44	797	0.32	1097	38%
6	-	-	0.40	878	0.31	1132	29%
7	-	-	0.39	900	0.29	1211	34%
8	-	-	0.36	975	0.26	1350	38%

5.1.4 *A Banded Matrix.* Finally, we consider a banded matrix of dimension n and with a half-bandwidth of t using a natural ordering. If factorized in a supernodal Cholesky factorization with exact supernodes, the only supernodes (either static or dynamic) that would appear would be in the last t columns of L. However, as typical with supernodal Cholesky factorization methods, CHOLMOD relaxes the restriction that two adjacent columns must be identical in their pattern (excluding the diagonal) to merge into a single supernode. A few extra entries are added to create larger supernodes. These entries remain in L, and are exploited by our dynamic supernodal update/downdate.

For example, consider a matrix with dimension $n = 3000$ and half-bandwidth $t = 500$. A concise L would have 500 entries in each column, except for the last 500 columns. Columns in the factor L with relaxed supernodes have up to 644 nonzeros, using the default supernode relaxation parameters in CHOLMOD (most columns of L have less than 540 entries). With 128 successive rank-1 updates with a randomly chosen W (but with no change in pattern), 196,754 columns were updated, as 2,330 individual columns, 60 supernodes of two columns, and 48,576 supernodes of four columns each. The 3000-by-128 sparse matrix W for the update $LDL^T + WW^T$ was created by choosing the pattern of W from 128 randomly chosen columns of L. The results are shown in Table VII.

Next, we added entries to W to make the last row of W completely nonzero, to force the nonzero pattern of L to change. The patterns of nearly all columns of L do change (2,331 of 3,000). This provides a worst-case scenario for our method. There are few natural supernodes (only those found via relaxed amalgamation) and almost all columns of L increase in length. The results for this experiment are shown in Table VIII. Comparing these results with Table VII, it is clear the performance of our method suffers in this worst-case example. However, it is still worthwhile to exploit dynamic supernodes. Consider a rank-128 update done in steps of rank-8 updates (the last row of each of these two tables). The performance of this update is 1350Mflops when the pattern does not change, and drops to 1099Mflops when the patterns of most columns of L change. This is a significant decrease, but still higher than a nonsupernodal rank-8 update, and much higher than any rank-1 update, supernodal or otherwise.

Table VIII. Banded Matrix, CHOLMOD Results (changing nonzero pattern)

	CHOLMOD nonsupernodal		CHOLMOD supernodal		
rank	time	Mflops	time	Mflops	speedup
1	0.73	481	0.67	524	9%
2	0.55	639	0.58	606	−5%
3	0.48	732	0.48	732	0%
4	0.46	764	0.43	818	7%
5	0.46	764	0.38	925	21%
6	0.44	799	0.37	950	19%
7	0.43	817	0.35	1004	23%
8	0.42	837	0.32	1099	31%

5.2 Dynamic Supernodal Solve

In this experiment, we compare four different methods for solving $LX = B$, where B is an n-by-k matrix and L is lower triangular with a nonunit diagonal:

(1) a simple method in CSparse (`cs_lsolve`) that solves $Lx = b$ with a single righthand side ($k = 1$);
(2) a nonsupernodal method, but with loop-unrolling and the ability to handle multiple righthand sides;
(3) the dynamic supernodal method, which is the same as (2) except that it detects and exploits dynamic supernodes; and
(4) a conventional supernodal method, using DTRSV and DGEMV for one righthand side and DTRSM and DGEMM for multiple righthand sides.

The last two methods are in CHOLMOD. Method (2) is the same as (3) except that the dynamic supernode detection in (3) was disabled. The ND/ND3K matrix was used, with the same ordering as discussed in the previous section. The results for all four methods are shown in Table IX, for various values of k. The performance for $k > 32$ is essentially the same for $k = 32$ for all methods.

The dynamic supernodal method (method (3)) is nearly twice as fast as *both* the nonsupernodal method and conventional BLAS-based supernodal method for $k = 2$. We did not expect our dynamic supernodal solve to outperform a conventional BLAS-based supernodal solve, for any value of k. For other values of k, it is about 50% faster than the nonsupernodal method. This is the same speedup obtained by the dynamic update/downdate method discussed in the previous section. The dynamic solver is slower than the conventional supernodal solver only for $k \geq 7$. For $k \geq 32$ the conventional supernodal solver is about 80% faster than the dynamic supernodal solver.

These results are significant in applications such as the LP Dual Active Set Algorithm that require many solutions to triangular systems. For many linear programming problems, the update/downdate and triangular solve time dominate the time required by the initial Cholesky factorization.

Table IX. Performance of Simple, Nonsupernodal, Dynamic Supernodal, and Conventional Supernodal Solvers

	(1) CSparse		(2) non-supernodal		(3) dynamic supernodal		(4) conventional supernodal	
k	time	Mflops	time	Mflops	time	Mflops	time	Mflops
1	0.060	421	0.058	437	0.041	619	0.048	525
2	-	-	0.092	550	0.050	1010	0.098	514
3	-	-	0.117	655	0.072	1050	0.107	708
4	-	-	0.108	935	0.078	1300	0.113	891
5	-	-	0.167	757	0.118	1072	0.123	1023
6	-	-	0.193	783	0.128	1188	0.128	1188
7	-	-	0.220	803	0.147	1201	0.136	1297
8	-	-	0.208	971	0.151	1334	0.143	1414
9	-	-	0.263	865	0.193	1175	0.151	1500
10	-	-	0.298	849	0.202	1250	0.157	1606
11	-	-	0.325	854	0.226	1229	0.167	1666
12	-	-	0.313	969	0.230	1317	0.173	1748
16	-	-	0.410	985	0.303	1335	0.206	1961
20	-	-	0.515	980	0.380	1329	0.240	2104
24	-	-	0.615	985	0.460	1317	0.270	2244
28	-	-	0.720	982	0.535	1321	0.303	2336
32	-	-	0.825	979	0.605	1335	0.340	2376

6. SUMMARY

We have shown how dynamic supernodes can be exploited to obtain efficient methods for updating or downdating a sparse Cholesky factorization and for solving the resulting triangular systems. The dynamic supernodal update/downdate and triangular solve methods are faster than the corresponding nonsupernodal methods. Our dynamic supernodal solve is faster than a conventional supernodal solve, except when solving with a righthand side with 6 or more columns. The CHOLMOD package is written in C, with a MATLAB interface, and is described in a companion paper [Chen et al. 2008].

REFERENCES

ANDERSON, E., BAI, Z., BISCHOF, C. H., BLACKFORD, S., DEMMEL, J. W., DONGARRA, J. J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. C. 1999. *LAPACK Users' Guide*, 3rd ed. SIAM, Philadelphia, PA.

ASHCRAFT, C. C. AND GRIMES, R. G. 1999. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*.

BISCHOF, C. H., PAN, C.-T., AND TANG, P. T. P. 1993. A Cholesky up- and downdating algorithm for systolic and SIMD architectures. *SIAM J. Sci. Comput. 14*, 3, 670–676.

CARLSON, N. A. 1973. Fast triangular factorization of the square root filter. *AIIA J. 11*, 1259–1265.

CHEN, Y., DAVIS, T. A., HAGER, W. W., AND RAJAMANICKAM, S. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Softw.* 35, 3.

DAVIS, T. A. 2006. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA.

DAVIS, T. A. AND HAGER, W. W. 1999. Modifying a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl. 20*, 3, 606–627.

DAVIS, T. A. AND HAGER, W. W. 2001. Multiple-Rank modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl. 22*, 997–1013.

DAVIS, T. A. AND HAGER, W. W. 2008a. Dual multilevel optimization. *Math. Program. 112*, 2, 403–425.

DAVIS, T. A. AND HAGER, W. W. 2008b. A sparse proximal implementation of the LP Dual Active Set Algorithm. *Math. Program. 112*, 2, 275-301.

DOBRIAN, F., KUMFERT, G. K., AND POTHEN, A. 2000. The design of sparse direct solvers using object oriented techniques. In *Advances in Software Tools in Scientific Computing*. Springer, 89–131.

DONGARRA, J. J., BUNCH, J. R., MOLER, C., AND STEWART, G. W. 1978. *LINPACK Users Guide*. SIAM, Philadelphia, PA.

DONGARRA, J. J., DU CROZ, J., DUFF, I. S., AND HAMMARLING, S. 1990. A set of level-3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1, 1–17.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND HANSON, R. J. 1988. An extended set of Fortran basic linear algebra subprograms. *ACM Trans. Math. Softw. 14*, 18–32.

GEORGE, A. AND LIU, J. W. H. 1981. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ.

GILBERT, J. R., NG, E. G., AND PEYTON, B. W. 1994. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl. 15*, 4, 1075–1091.

GILL, P. E., GOLUB, G. H., MURRAY, W., AND SAUNDERS, M. A. 1974. Methods for modifying matrix factorizations. *Math. Comput. 28*, 126, 505–535.

GOTO, K. AND VAN DE GEIJN, R. 2002. On reducing TLB misses in matrix multiplication. TR-2002-55, University Texas at Austin, Department of Computer Sciences.

GOULD, N. I. M., HU, Y., AND SCOTT, J. A. 2007. A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Softw. 33*, 2, 32 pages.

HAGER, W. W. 1989. Updating the inverse of a matrix. *SIAM Rev. 31*, 2, 221–239.

HÉNON, P., RAMET, P., AND ROMAN, J. 2002. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Comput. 28*, 2, 301–321.

LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw. 5*, 308–323.

LIU, J. W. H. 1986. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Trans. Math. Softw. 12*, 2, 127–148.

LIU, J. W. H. 1990. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl. 11*, 1, 134–172.

LIU, J. W. H., NG, E. G., AND PEYTON, B. W. 1993. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl. 14*, 1, 242–252.

NG, E. G. AND PEYTON, B. W. 1993. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput. 14*, 5, 1034–1056.

PAN, C.-T. 1990. A modification to the LINPACK downdating algorithm. *BIT 30*, 707–722.

ROTHBERG, E. AND GUPTA, A. 1991. Efficient sparse matrix factorization on high-performance workstations - Exploiting the memory hierarchy. *ACM Trans. Math. Softw. 17*, 3, 313–334.

ROTKIN, V. AND TOLEDO, S. 2004. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Trans. Math. Softw. 30*, 1, 19–46.

SCHREIBER, R. 1982. A new implementation of sparse Gaussian elimination. *ACM Trans. Math. Softw. 8*, 3, 256–276.

SCOTT, J. A. AND HU, Y. 2007. Experiences of sparse direct symmetric solvers. *ACM Trans. Math. Softw. 33*, 3, 28 pages.

STEWART, G. W. 1979. The effects of rounding error on an algorithm for downdating a Cholesky factorization. *J. Inst. Math. Appl. 23*, 203–213.

STEWART, G. W. 1998. *Matrix Algorithms, Volume 1: Basic Decompositions*. SIAM, Philadelphia, PA.

Received September 2006; revised November 2007; accepted May 2008