Dynamic supernodes in sparse Cholesky update/downdate and triangular solves *

TIMOTHY A. DAVIS[†]and WILLIAM W. HAGER[‡]

September 8, 2006

Technical report TR-2006-004, CISE Dept, Univ. of Florida, Gainesville, FL

Abstract

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.

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 L. 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

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

[†]Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, USA. email: davis@cise.ufl.edu. http://www.cise.ufl.edu/∼davis.

[‡]Dept. of Mathematics, Univ. of Florida, Gainesville, FL, USA. email: hager@math.ufl.edu. http://www.math.ufl.edu/∼hager.

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 [20]. The sparse rank-1 update when L is not a supernodal Cholesky factorization is discussed in [7], while the multiple rank case is in [8]. 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 paper [6].

The *supernodal* Cholesky factorization method [2, 11, 21, 26, 28, 29] 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 paper, 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 paper 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.

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 [22, 14, 13]) 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 right-hand 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 [23, 24, 30]. The parent of node j in the tree is given by the first off-diagonal nonzero entry l_{ij} in column j,

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

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. That is, $\mathcal{L}_j = \{i \,|\, l_{ij} \neq 0\}$. 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 $\mathcal{L}_j = \mathcal{L}_{j-1} \setminus \{j\}$. That is, the set of columns in a supernode forms 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 [25]. First, the elimination tree of A is computed in nearly $O(|A|)$ time [23, 24].[1] More precisely the time is $\mathrm{O}(|A|\alpha(|A|, n))$ 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`) using CHOLMOD.

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 [16]. In MATLAB, this is computed by the routine `symbfact`, also using CHOLMOD. 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 [15],

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

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

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

$$\left[\begin{array}{ccc} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{array}\right] \left[\begin{array}{ccc} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{array}\right] = \left[\begin{array}{ccc} A_{11} & A_{21}^T & A_{31}^T \\ A_{21} & A_{22} & A_{32}^T \\ A_{31} & A_{32} & A_{33} \end{array}\right], \tag{3}$$

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,

$$\left[\begin{array}{c} S_1 \\ S_2 \end{array}\right] = \left[\begin{array}{c} A_{22} \\ A_{32} \end{array}\right] - \left[\begin{array}{c} L_{21} \\ L_{31} \end{array}\right] L_{21}^T. \tag{4}$$

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.

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.

In MATLAB 7.2, `x=A\b` and `chol` use the above supernodal Cholesky factorization method, as implemented in CHOLMOD. The performance of CHOLMOD and many other sparse Cholesky factorization packages is discussed in [19].

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 $L_{22}x_2 = \overline{b}_2$ where $\overline{b}_2 = b_2 - L_{21}x_1$ 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 [19].

3 Updating a sparse Cholesky factorization

3.1 Non-supernodal update

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 [7]. 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) [3, 4, 17, 31, 32, 27]. 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 [6]; that algorithm is a mere 35 lines of C.

The rank-1 update/downdate is discussed in [7]. 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 driving motivation is an active-set linear programming method [10, 9], 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 multi-set, with multiplicities for each entry in the set. This extra information is costly for a general application, and is thus it is often more efficient to retain all the nonzeros during a downdate rather try to determine which nonzero should be zero after the downdate.

Instead, assume that either an update or downdate can add entries to $\overline{A}$ and thus also to its updated/downdated Cholesky factor $\overline{L}$. Neither the update nor the downdate are assumed to remove any entries. 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 [8]. 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.

3.2 Supernodal update

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 [7, 8].

It would be simpler to assume, as in [6], that the nonzero pattern of 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 non-supernodal update/downdate. The structure of L, and its supernodes, would be static. However, this requirement is too

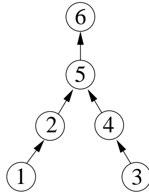

Figure 1: Example elimination tree

limiting for many applications. In particular, it would not be suitable in our motivating application, the LP Dual Active Set Algorithm, LPDASA.

Thus, 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 non-supernodal 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, gaps between the columns of unused memory space are permitted in our data structure, and the columns need not appear in monotonic 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 [8], except that multiplicities are not kept. The run time of the symbolic phase is always 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.

3.3 Dense Cholesky update

Consider the method in Algorithm 1 for updating/downdating a dense LDL^T factorization (a modified version of Method C1 in [17]). 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 [17], for example). They all include an innermost loop in which W and a column of L are used to modify each other.

ALGORITHM 1 (Dense rank-k update/downdate).

for $r = 1$ **to** k **do**
    $\alpha_r = 1$
for $j = 1$ **to** n **do**
    for $r = 1$ **to** k **do**
        $\overline{\alpha} = \alpha_r + \sigma w_{jr}^2 / d_j$
        $d_j = d_j \overline{\alpha}$
        $\gamma_r = -\sigma w_{jr} / d_j$
        $d_j = d_j / \alpha_r$
        $\alpha_r = \overline{\alpha}$
    for $i = j + 1$ **to** n **do**
        for $r = 1$ **to** k **do**
            $w_{ir} = w_{ir} - w_{jr} l_{ij}$
            $l_{ij} = l_{ij} - \gamma_r w_{ir}$

An update of the LL^T factorization is nearly identical. For example, the `cs_updown` function in CSparse [6] is based on Bischof, Pan, and Tang's [3] combination of Carlson's update [4] and Pan's downdate [27]. All of these methods modify the LL^T factorization instead, as does Stewart's method in LINPACK [12, 31, 32] (the method used by `cholupdate` in MATLAB). To update LL^T factorization, the innermost loop requires 5 floating-point operations instead of 4 for the LDL^T case. The memory traffic is identical, however, since the extra term is a scalar which would be held in registers or cache.

3.4 Dynamic supernodal update

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. Each column L_{*j} along that path is modified by W, and likewise modifies the matrix W.

The algorithm exploits three cases: supernodes consisting of one, two, or four columns of L. 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.

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).

Consider the update of a single column of L. This method corresponds to the rank-k update given in [8], and a single iteration of the **for** j loop in ALGORITHM 1. The loop across the rows i of the column iterate 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 loop is unchanged.

Let $s = |\mathcal{L}_j|$ be the number of entries in the jth column of L. The 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). The ratio of floating point operations per memory reference is given below.

Non-supernodal update								
rank-k update:	$k = 1$	2	3	4	5	6	7	8
flops / memory access:	0.80	1.14	1.33	1.45	1.53	1.60	1.65	1.68

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 sub-diagonal 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 the algorithm below. It 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.

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:
$l_{i_1,j} = x_{11}$
$l_{i_2,j} = x_{21}$
$l_{i_1,j+1} = x_{22}$
$l_{i_2,j+1} = x_{22}$

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.

Assume W is stored in row-major order, in scattered form. That is, 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 non-supernodal 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 non-supernodal 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 cached variables (t, x, γ, w_{jr} and $w_{j+1,r}$), four entries of L are accessed for each $s/2$ outer iteration, and two entries of W are accessed 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 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. The flops per memory access for each possible value of k is given in the table below.

Dynamic supernodal update with 2-column supernodes								
rank-k update:	$k=1$	2	3	4	5	6	7	8
flops / memory access:	1.14	1.78	2.18	2.46	2.67	2.82	2.95	3.05

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, above, in a 2-column update.

This idea can 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. The flops per memory access ratio becomes $16sk/(2sk+9s)$; these ratios are shown below for each value of k:

Dynamic supernodal update with 4-column supernodes								
rank-k update:	$k=1$	2	3	4	5	6	7	8
flops / memory access:	1.45	2.46	3.20	3.76	4.21	4.57	4.87	5.12

Its peak ratio 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.

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 right-hand sides (b can be an n-by-4 dense matrix). The solution x is stored in row-major form.

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 3. Each inner iteration multiplies a dense 1-by-3 vector with a 3-by-4 matrix.

ALGORITHM 3 Supernodal solve (3 columns of L and 4 right-hand 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 $\mathcal{L}_j \setminus \{j, j+1, j+2\}$
$X_{i,*} = X_{i,*} - L_{i,j:j+2}y$

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 X we store 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 is similar.

With a single right-hand 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.

By comparison, a simple lower triangular solver (one right-hand 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.

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 non-supernodal multiple-rank update in [8] with our new dynamic supernodal update in CHOLMOD. 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, 4 GB RAM (DDR 333 Mhz), 512 KB cache, an 800 MHz memory bus, the Goto BLAS v1.05 [18], and running Linux). The theoretical peak performance of the computer is 6.4 GFlops. The `gcc` compiler was used (version 3.3.5, with `-O3` optimization).

5.1 Dynamic supernodal update/downdate

Three matrices were used for the tests presented below.

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 non-supernodal 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

The update $LDL^T + CC^T$ was selected so that the nonzero pattern of L did not change, to compare the results with CSparse. It was computed 16 different ways: in steps of 1 to 8 columns at a time, and both with and without dynamic supernodes. The results in the table below show that using dynamic supernodes increases the performance of the sparse Cholesky update/downdate by about 50% when the rank of C is 5 or greater.

	non-supernodal		supernodal		
rank	time	Mflops	time	Mflops	speedup
1	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%

CSparse includes a sparse rank-1 Cholesky update/downdate method that updates the LL^T factorization. It performs about 25% more floating-point work, but requires the same memory traffic as an LDL^T update. It only applies to problems where the nonzero pattern of L does not change. Thus, for a rank-1 update, its performance in terms of run time and MFlops can exceed CHOLMOD. Note that the 3.29 seconds in CSparse excludes the MATLAB interface, which must copy L (a MATLAB function should not modify its inputs).

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 [10, 9]. The method cannot be compared with CSparse, since the pattern of L is changing.

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 non-supernodal 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

	non-supernodal		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%

If the update is followed by another one with the same C, then the nonzero pattern of L remains unchanged. Almost the same amount of floating-point work is required. For this experiment, CSparse takes 4.65 seconds plus an additional 0.15 seconds in its MATLAB interface. The results for CHOLMOD are listed below.

	non-supernodal		supernodal		
rank	time	Mflops	time	Mflops	speedup
1	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%

The final matrix we tested was a randomly generated dense symmetric positive definite matrix of dimension $n = 3000$. The `chol` function in MATLAB takes 2.3 seconds to factorize it, a rate of 3.9 GFlops using LAPACK [1]. If that same matrix is converted into a sparse matrix, CHOLMOD requires 3.1 seconds to factorize it (2.9 GFlops), excluding the ordering time but including both symbolic and numeric factorization. A rank-1 dense update using `cholupdate` in MATLAB (based on LINPACK [12, 31]), 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 is the time to copy the matrix to and from the MATLAB workspace.

The dense matrix-vector multiply (DGEMV) in the Goto BLAS has a peak performance of 2.7 GFlops 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 (637 MFlops for $n = 500$, and 597 MFlops 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. 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.

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 non-unit diagonal.

1. a simple method in CSparse (`cs_lsolve`) that solves $Lx = b$ with a single right-hand side ($k = 1$),

2. a non-supernodal method, but with loop-unrolling and the ability to handle multiple right-hand 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 right-hand side and DTRSM and DGEMM for multiple right-hand 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. For this matrix, CSparse obtains a performance of 421 Mflops when solving $Lx = b$. The results for the other three methods are shown below, for various values of k. The performance for $k > 32$ is essentially the same for $k = 32$ for all three methods.

k	non-supernodal time	non-supernodal Mflops	dynamic supernodal time	dynamic supernodal Mflops	conventional supernodal time	conventional supernodal Mflops
1	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

The dynamic supernodal method is nearly twice as fast as the non-supernodal method and conventional BLAS-based supernodal method for $k = 2$. For other values of k, it is about 50% faster than the non-supernodal 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 BLAS-based supernodal solver only for $k \geq 7$. For $k \geq 32$ the BLAS-based 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.

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 methods are faster than both the non-supernodal methods and conventional supernodal methods, except when solving a lower triangular system with 8 or more simultaneous right-hand sides. The CHOLMOD package that includes these methods is described in a companion paper, [5].

References

[1] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK*

Users' Guide. SIAM, Philadelphia, 3rd edition, 1999.

[2] C. C. Ashcraft and R. G. Grimes. SPOOLES: an object-oriented sparse matrix library. In *Proc. 1999 SIAM Conf. Parallel Processing for Scientific Computing*, Mar. 1999.

[3] C. H. Bischof, C.-T. Pan, and P. T. P. Tang. A Cholesky up- and downdating algorithm for systolic and SIMD architectures. *SIAM J. Sci. Comput.*, 14(3):670–676, 1993.

[4] N. A. Carlson. Fast triangular factorization of the square root filter. *AIIA Journal*, 11:1259–1265, 1973.

[5] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 8xx: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Software*, 2006. (submitted).

[6] T. A. Davis. *Direct Methods for Sparse Linear Systems.* SIAM, Philadelphia, PA, 2006.

[7] T. A. Davis and W. W. Hager. Modifying a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.*, 20(3):606–627, 1999.

[8] T. A. Davis and W. W. Hager. Multiple-rank modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.*, 22:997–1013, 2001.

[9] T. A. Davis and W. W. Hager. Dual multilevel optimization. *Math. Program.*, page to appear, 2006.

[10] T. A. Davis and W. W. Hager. A sparse proximal implementation of the LP Dual Active Set Algorithm. *Math. Program.*, pages http://dx.doi.org/10.1007/s10107–006–0017–0, 2006.

[11] F. Dobrian, G. K. Kumfert, and A. Pothen. The design of sparse direct solvers using object oriented techniques. In *Adv. in Software Tools in Sci. Computing*, pages 89–131. Springer-Verlag, 2000.

[12] J. J. Dongarra, J. R. Bunch, C. Moler, and G. W. Stewart. *LINPACK Users Guide.* SIAM, Philadelphia, 1978.

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

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Trans. Math. Software*, 14:18–32, 1988.

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

[16] J. R. Gilbert, E. G. Ng, and B. W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.*, 15(4):1075–1091, 1994.

[17] P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders. Methods for modifying matrix factorizations. *Math. Comp.*, 28(126):505–535, 1974.

[18] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. TR-2002-55, Univ. Texas at Austin, Dept. of Computer Sciences, 2002.

[19] N. I. M. Gould, Y. Hu, and J. A. Scott. A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Software*, 200x. (to appear).

[20] W. W. Hager. Updating the inverse of a matrix. *SIAM Review*, 31(2):221–239, 1989.

[21] P. Hénon, P. Ramet, and J. Roman. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing*, 28(2):301–321, 2002.

[22] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:308–323, 1979.

[23] J. W. H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Trans. Math. Software*, 12(2):127–148, 1986.

[24] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11(1):134–172, 1990.

[25] J. W. H. Liu, E. G. Ng, and B. W. Peyton. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 14(1):242–252, 1993.

[26] E. G. Ng and B. W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14(5):1034–1056, 1993.

[27] C.-T. Pan. A modification to the LINPACK downdating algorithm. *BIT*, 30:707–722, 1990.

[28] E. Rothberg and A. Gupta. Efficient sparse matrix factorization on high-performance workstations - Exploiting the memory hierarchy. *ACM Trans. Math. Software*, 17(3):313–334, 1991.

[29] V. Rotkin and S. Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Trans. Math. Software*, 30(1):19–46, 2004.

[30] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Trans. Math. Software*, 8(3):256–276, 1982.

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

[32] G. W. Stewart. *Matrix algorithms, Volume 1: Basic decompositions*. SIAM, Philadelphia, 1998.