An Out-of-Core Sparse Symmetric-Indefinite Factorization Method

OMER MESHAR, DROR IRONY, and SIVAN TOLEDO
Tel-Aviv University

We present a new out-of-core sparse symmetric-indefinite factorization algorithm. The most significant innovation of the new algorithm is a dynamic partitioning method for the sparse factor. This partitioning method results in very low I/O traffic and allows the algorithm to run at high computational rates, even though the factor is stored on a slow disk. Our implementation of the new code compares well with both high-performance in-core sparse symmetric-indefinite codes and a high-performance out-of-core sparse Cholesky code.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Linear systems (direct and iterative methods)*; *sparse, structured, and very large systems (direct and iterative methods)*; G.4 [**Mathematical Software**]: Algorithm design and analysis; G.4 [**Mathematics of Computing**]: Mathematical Software—*Efficiency*; E.5 [**Data**]: Files—*Organization/structure*

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Out-of-Core, symmetric-indefinite

1. INTRODUCTION

We present a method for factoring a large sparse symmetric-indefinite matrix A. By storing the triangular factor of A on-disk, the method can handle large matrices whose factors do not fit within the main memory of the computer. A dynamic I/O-aware partitioning of the matrix ensures that the method performs little disk I/O, even when the factor is much larger than main memory. Our experiments indicate that the method can factor finite-element matrices with factors larger than 10 GB on an ordinary 32-bit workstation (a 2.4 GHz Intel-based PC) in less than an hour.

This method allows us to solve linear systems $Ax = b$ with a single righthand-side and linear systems $AX = B$ with multiple righthand-sides

efficiently and accurately. Linear systems with a symmetric-indefinite coefficient matrix arise in optimization, finite-element analysis, and shift-invert eigensolvers (even when the matrix whose eigen decomposition is sought-after is definite).

Linear solvers that factor the coefficient matrix into a product of permutation, as well as triangular, diagonal, and orthogonal factors, are called *direct methods*. Our method is direct, and it decomposes A into permutation, triangular, and block-diagonal factors (the block-diagonal factor has 1-by-1 and 2-by-2 blocks). Compared to iterative linear solvers, direct solvers tend to be more reliable and accurate, but they sometimes require significantly more time and memory. In general, direct solvers are preferred when the user has little expertise in iterative methods, when iterative methods fail or converge too slowly, or when many linear systems with the same coefficient matrix must be solved. In many applications, such as finite-element analysis and shift-invert eigensolvers, many linear systems with the same coefficient matrix are indeed solved, and in such cases a direct solver is often the most appropriate.

The size of the triangular factor of a symmetric matrix and the amount of work required to compute the factorization are sensitive to the ordering of rows and columns of the matrix. Therefore, matrices are normally symmetrically permuted to reduce the fill in the factors. Although the problem of finding a minimal-fill ordering is NP-complete, there exist effective heuristics that work well in practice (as well as provable approximations that have not been shown to work well in practice [Natanzon et al. 1998]). Even when the matrix has been permuted using a fill-reducing permutation, the factor is often much larger (denser) than the matrix and may be too large to fit in memory, even when the matrix itself fits comfortably. When the factor does not fit within main memory, the user has three choices: to resort to a so-called *out-of-core* algorithm, which stores the factors on-disk, or to switch to either an iterative algorithm or a machine with a larger memory. Since machines with more than a few gigabytes of main memory are still beyond the reach of most users, and since iterative solvers are not always appropriate, there are cases when an out-of-core method is the best solution.

The main challenge in designing an out-of-core algorithm is ensuring that it does not perform too much disk input/output (I/O). The disk-to-memory bandwidth is usually about two orders of magnitude lower than memory-to-processor bandwidth. Therefore, to achieve a high computational rate, an out-of-core algorithm must access data structures on-disk infrequently; most data accesses should be to data that is stored, perhaps temporarily, in main memory. Algorithms in numerical linear algebra achieve this goal by partitioning matrices into blocks of rows and columns. When matrices are dense, relatively simple one- and two-dimensional partitions into blocks of consecutive rows and columns work well; when matrices are sparse, the partitioning algorithm must consider the nonzero structure of the matrices. Essentially the same partitioning strategies are used whether the I/O is performed automatically by the virtual-memory system (or by cache policies higher in the memory hierarchy) or by explicitly using system calls. In general, explicit I/O tends to work better than virtual memory when data structures on disk are significantly larger than

memory. Explicit I/O is the only choice when data structures on disk are too large to fit into the virtual address-space of the program (larger than 2–4 GB on 32-bit processors, depending on the operating system).

To the best of our knowledge, our algorithm is the first out-of-core sparse symmetric-indefinite factorization method to be described in the literature. At least two proprietary out-of-core symmetric-indefinite codes do exist (BSCLIB-EXT and MSC-NASTRAN), but they have not been presented in the literature. Another algorithm, by Dobrian and Pothen [2006], has been recently presented at a conference, but it appears that the associated code is not yet complete. Our algorithm is also the first *left-looking* out-of-core sparse symmetric-indefinite factorization method; earlier methods have all been multifrontal. Several out-of-core methods have been proposed for the somewhat easier problem of factoring a symmetric positive-definite matrix, most recently by Rothberg and Schreiber [1999] and Rotkin and Toledo [2004]. Gilbert and Toledo [1999] proposed a method for the more general problem of factoring a general sparse unsymmetric matrix [1999]. This algorithm is more widely applicable than the one we present here, but is also significantly slower. For earlier sparse out-of-core methods, see the references in the aforementioned articles.

Our new method is based on a sparse left-looking formulation of the LDL^T factorization. Our code is not the first left-looking LDL^T code [Ashcraft and Grimes 1999], but to the best of our knowledge, a left-looking formulation has never been described in the literature (Ashcraft and Grimes' article [1999] documents the software, but not the algorithm). We partition the matrix into blocks called *compulsory subtrees* [Rotkin and Toledo 2004] to achieve I/O efficiency, but the matrix is partitioned dynamically during numeric factorization to allow for pivoting (the method of Rotkin and Toledo [2004] partitions the matrix statically before numeric factorization begins). To achieve a high computational rate, we have implemented a partitioned *dense LDL^T* factorization to factor large dense diagonal blocks; the corresponding LAPACK routine cannot be used in sparse codes.

Our implementation of the new algorithm is reliable and performs well. On a 2.4 GHz PC, it factors an indefinite finite-element matrix with about a million rows and columns in less than an hour (wallclock time, including all input/output), producing a factor with about 1.3×10^9 nonzeros (more than 10 GB). A larger matrix, whose factor contained about 3.3×10^9 nonzeros, took about 9.5 hours to factor. On this machine, the factorization runs at a rate of 1–2 billion floating-point operations per second, *including* the disk I/O-time. The user can specify where to store the factor, which is broken into files of smaller than 2 GB, to allow the code to run on-file systems where the individual files are limited to 2 GB.

The article is organized as follows. The next section presents the background to sparse symmetric-indefinite factorizations. The one that follows presents our left-looking formulation of the factorization; we use this formulation in both in-core and out-of-core codes. Section 5 presents our new out-of-core algorithm and its implementation. Section 6 presents our experimental results, and Section 7 concludes.

2. BACKGROUND TO SPARSE SYMMETRIC-INDEFINITE FACTORIZATIONS

This section describes the basics of sparse symmetric-indefinite factorizations. For additional details and references, see the monograph of Duff et al. [1986] and the articles cited in the following.

2.1 Symmetric-Indefinite Factorizations

A symmetric-indefinite n-by-n matrix can be factored into a product $A = PLDL^T P^T$, where P is a permutation matrix, L is unit lower triangular (has 1's on the diagonal), and D is block diagonal with 1-by-1 and 2-by-2 blocks. The permutation P is computed during factorization to ensure numerical stability. This factorization can be used to quickly solve linear systems $AX = B$ and compute the inertia of A [Bunch and Kaufman 1977]. If A is dense, the amount of floating-point arithmetic required is only slightly larger than that required for the Cholesky factorization of $A + \sigma I$, where $\sigma > -\lambda_{\min}(A)$, the smallest eigenvalue of A. The amount of work involved in pivot searches, to construct P so that the growth in L is controlled, is usually small when a partial pivoting strategy is used, like that of Bunch and Kaufman [1977]. When complete [Bunch et al. 1976] or rook pivoting is used [Ashcraft et al. 1998], the cost of pivot searches can be significant.

When A is sparse, the permutation P has a dramatic effect on the sparsity of the triangular factor L. There are cases wherein one choice of P would lead to dense Schur complement after one elimination step (and to a dense triangular factor), whereas another choice, equally good from a numerical point of view, would keep the reduced matrices and the factor as sparse as A. This issue is addressed in the following way. First, a fill-reducing permutation Q for the Cholesky factor C of $A + \sigma I$ is found. The rows and columns of A are symmetrically permuted according to Q, and a symmetric-indefinite factorization is applied to $Q^T A Q = PLDL^T P^T$. If the choice $P = I$ is numerically sound for the factorization of $Q^T A Q$, then the amount of fill in L is roughly the same as that in the Cholesky factor C. (The fill is exactly the same if D has only 1-by-1 blocks; otherwise, full 2-by-2 diagonal blocks cause more fill in the first column of the block, but this fill does not generate additional fill in the trailing submatrix. Diagonal zeros in the diagonal block cause slightly less fill in both the second column of the block and in the trailing submatrix.) In general, however, $P = I$ is not a valid choice. An arbitrary choice of P can destroy the sparsity in L completely, so most of the sparse symmetric-indefinite factorization methods attempt to constrain the pivot search such that the resulting permutation QP is not too different from Q alone. We explain how the pivot search is constrained next.

2.2 The Elimination Tree and Assembly Tree

A combinatorial structure called the *elimination tree of A* [Schreiber 1982] (etree) plays a key role in virtually all symmetric factorization methods, both definite and indefinite [Liu 1990]. When A is definite, the etree is used to predict the structure of the factor, represent data dependencies, and schedule the factorization. In symmetric-indefinite factorizations, the etree is used to
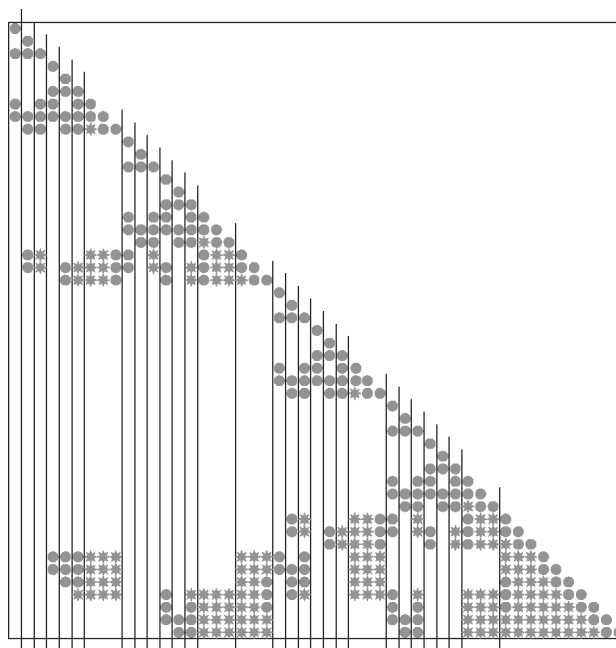
Fig. 1. A fundamental supernodal decomposition of the factor of a matrix corresponding to a 7-by-7 grid problem, ordered with nested dissection. The circles correspond to elements that are nonzero in the coefficient matrix and the stars represent fill elements.

constrain P so that L does not fill too much. The etree is also used for indefinite factorization in which P is thusly constrained to represent data dependencies, schedule the factorization, and estimate the structure of the factor (but not to predict it exactly).

The elimination tree is a rooted forest (tree unless A can be permuted to a nontrivial block-diagonal form) with n vertices. The parent $\pi(j)$ of vertex j in the etree is defined to be $\pi(j) = \min_{i>j}\{C_{ij} \neq 0\}$, where C is the Cholesky factor of $A + \sigma I$. An equivalent definition is the transitive reduction of the underlying directed graph of A. This alternative definition is harder to visualize, but does not reference a Cholesky factorization. The etree can be computed directly from the nonzero structure of A in time that is essentially linear in the number of nonzeros in A.

Virtually all state-of-the-art sparse indefinite factorization algorithms use a *supernodal partitioning* of the factor L, illustrated in Figure 1 [Duff and Reid 1983a; Ng and Peyton 1993; Rothberg and Gupta 1991]. The factor is decomposed into dense diagonal blocks and corresponding subdiagonal blocks such that the rows in subdiagonal blocks are either entirely zero or almost completely dense. In an indefinite factorization, the algorithm computes a preliminary supernodal partitioning for the Cholesky factor C before the numeric factorization begins. The partitioning is refined during numeric factorization such that it is a correct partitioning of the actual factor L.

j, k, ℓ	Indices of supernodes
p, q	Indices of columns
$\pi(j)$	Parent of j in the elimination tree
Ω_j	Indices of columns in supernode j of $A + \sigma I$
Ξ_j	Indices of rows in the subdiagonal block of supernode j of $A + \sigma I$
$\tilde{\Omega}_j$	Indices of the columns that have been eliminated during the processing of supernode j of A
$\tilde{\Xi}_j$	Indices of the rows in the subdiagonal block of supernode j of A
Δ_j	Indices of columns that were delayed *from* supernode j to its parent $\pi(j)$
Ψ_j	Indices of columns that were delayed *to* supernode j from its children
$A^{(j)}$	The reduced matrix immediately after the elimination of supernode j
$F^{(j)}$	Contributions from the subtree rooted at supernode j (including contributions to columns in the subtree)
$L^{(j)}$	Representation of columns $\Omega_j \cup \Psi_j$ in $A^{(j)}$, which are transformed into columns $\tilde{\Omega}_j$ in L and columns Δ_j in $A^{(j+1)}$
$\bar{A}$	The reduced matrix during the processing of supernode j
Φ_j	The columns that have already been eliminated from supernode j during its processing

Fig. 2. The notation used in Sections 2–4. The key to the notation is the fact that all matrices are *n*-by-*n*, even though implementations discard eliminated rows and columns.

Supernodal partitioning is represented by a *supernodal elimination tree*, or *assembly tree*, denoted T. In the supernodal etree, each tree vertex represents a supernode. The vertices are labeled 1 to s using a postorder traversal, where s is the number of supernodes. We denote by T_j the vertices in the subtree rooted at j, including j itself. The pivoting permutation P is chosen so that the supernodal elimination trees of C and L coincide, although some of the supernodes of L might be empty. We associate with supernode j the ordered set Ω_j of the indices of columns in the supernode in C, and the unordered set Ξ_j of indices of nonzero row indices in the subdiagonal block of C. We denote by $\tilde{\Omega}_j$ and $\tilde{\Xi}_j$ the same sets, respectively, in L. The ordering of indices in Ω_j is some ordering consistent with a postorder traversal of the nonsupernodal etree of A. For example, the sets of supernode 29, the next-to-rightmost supernode in Figure 1, are $\Omega_{29} = (37, 38, 39)$ and $\Xi_{29} = \{40, 41, 42, 46, 47, 48, 49\}$.

If A is positive definite, the factorization algorithm eliminates all columns in Ω_j during the processing of vertex j of the etree; this requires a rank-$|\Omega_j|$ update to the remaining equations, and more specifically, to rows and columns with indices in Ξ_j. When A is indefinite, however, the algorithm may be unable to eliminate all the columns in Ω_j during the processing of vertex j. The columns that are not eliminated are *delayed* to j's parent $\pi(j)$. The parent tries to eliminate both the columns in $\Omega_{\pi(j)}$ and delayed columns. The columns in this set that the parent fails to eliminate are delayed to $\pi(\pi(j))$, and so on. At the root, all remaining columns are eliminated [Duff and Reid 1983b]. In essence, a column is delayed when all admissible pivot rows are numerically unstable; delaying provides new admissible pivot rows. We denote the set of columns that were delayed from j to its parent by Δ_j. This set includes all the columns in T_j that were not eliminated within the subtree.

2.3 Sparse Factorizations

Sparse factorization codes differ in how they represent and apply the Schur-complement modification to the trailing submatrix. Denoting the matrix after the elimination of columns in supernode j by $A^{(j)}$ (and using $A^{(0)} = A$), the equations that govern the factorization process are

$$A^{(j)} = A^{(j-1)} - L_{*,\tilde{\Omega}_j} D_{\tilde{\Omega}_j,\tilde{\Omega}_j} L_{\tilde{\Omega}_j,*}^T$$
$$= A - \sum_{k \leq j} L_{*,\tilde{\Omega}_k} D_{\tilde{\Omega}_k,\tilde{\Omega}_k} L_{\tilde{\Omega}_k,*}^T .$$

Restricted to a single column q, these equations become

$$A_{*,q}^{(j)} = A_{*,q} - \sum_{\substack{k \leq j \\ q \in \tilde{\Xi}_k}} L_{*,\tilde{\Omega}_k} D_{\tilde{\Omega}_k,\tilde{\Omega}_k} L_{\tilde{\Omega}_k,q}^T . \tag{1}$$

To keep the representation reasonably simple, $A^{(j)}$ always denotes an n-by-n matrix; however, factorization codes usually only represent the unelimated rows and columns of $A^{(j)}$. Codes that generate an explicit representation of $A^{(j)}$ after every supernodal elimination step are called *right-looking* codes. Due to the difficulty of efficiently adding two sparse matrices, there are no state-of-the-art right-looking codes. Codes that represent $A^{(j)}$ completely implicitly, storing only A and the columns of L and D that have been computed thus far, are called *left-looking* [Ng and Peyton 1993; Rothberg and Gupta 1991]. *Multifrontal* codes [Duff and Reid 1983a; Liu 1992] represent $A^{(j)}$ using A and matrices that represent partial sums of the form $F^{(\ell)} = \sum_{k \in T_\ell} L_{*,\tilde{\Omega}_k} D_{\tilde{\Omega}_k,\tilde{\Omega}_k} L_{\tilde{\Omega}_k,*}^T$. Here again, $F^{(\ell)}$ denotes an n-by-n matrix, but implementations only store the rows and columns of $F^{(\ell)}$ that have not yet been eliminated. Each partial sum accumulates contributions from supernodes in one connected component of the subgraph of the supernodal elimination tree induced by vertices 1 through j; the root of the subgraph is ℓ. The partial sums are all zero, except in rows and columns $\tilde{\Xi}_\ell \cup \tilde{\Omega}_\ell$. Since the rows and columns in $\tilde{\Omega}_\ell$ have already been eliminated, they are never included in the actual representation of $F^{(\ell)}$. Therefore, the sums can be stored as small dense matrices. *Frontal* codes [Duff and Scott 1999; Irons 1970] are similar, except that they represent $A^{(j)}$ using A in addition to just one other matrix, which represents the total sum $\sum_{k \leq j} L_{*,\tilde{\Omega}_k} D_{\tilde{\Omega}_k,\tilde{\Omega}_k} L_{\tilde{\Omega}_k,*}^T$. Although frontal codes sometimes perform well, and are relatively easy to implement out-of-core, they generally tend to introduce many explicit zeros into the representation of the factor. Frontal codes are outside the scope of this article.

In the indefinite case, the multifrontal approach is the most common and well-documented in literature [Ashcraft et al. 1998; Duff and Reid 1983a]. However, we use a left-looking approach that we describe in the next section.

3. LEFT-LOOKING FACTORIZATION

Previous research on sparse out-of-core factorization methods for symmetric positive-definite matrices suggests that left-looking methods are more efficient

than multifrontal ones [Rothberg and Schreiber 1999; Rotkin and Toledo 2004]. The difference between multifrontal and the left-looking approaches is in the way in which Schur-complement modifications to the reduced matrix are represented. In the multifrontal algorithm, the matrix $F^{(j)}$ is computed when supernode j is factored. Before $\pi(j)$ is factored, the uneliminated columns in $F^{(j)}$ are appended to $F^{(\pi(j))}$, and $F^{(j)}$ is discarded. By contrast, in the left-looking approach, outer products of the form $L_{*,\tilde{\Omega}_k} D_{\tilde{\Omega}_k,\tilde{\Omega}_k} L^T_{\tilde{\Omega}_k,*}$ are not represented explicitly until they are actually subtracted from columns of the reduced matrix. The disadvantage of the multifrontal approach is that it often simultaneously represents multiple contributions to the same nonzero in L. The representation of these contributions (which are not part of the data structure that represents the factor L) uses up memory and causes additional I/O activity. We note that some out-of-core codes, such as BSCLIB-EXT, do use a multifrontal approach. However, experiments performed by Rothberg and Schreiber [1999] using their own codes indicated that a BSCLIB-EXT-like approach performs more I/O than a left-looking approach.

Unfortunately, left-looking sparse indefinite factorizations have not been described in the literature. There is actually one in-core code that uses such a method, SPOOLES [Ashcraft and Grimes 1999], but the algorithm it uses is not described explicitly anywhere. We describe here the formulation of the in-core algorithm, and the next section explains how we implemented it out-of-core.

The left-looking algorithm traverses the etree and factors the matrix in postorder. Let $\Psi_j = \cup_{k,j=\pi(k)}\Delta_k$ be the set of columns that are delayed to supernode j. To process supernode j, the algorithm creates a matrix $L^{(j)}$ that will represent columns $\Omega_j \cup \Psi_j$ in the reduced matrix $A^{(j)}$, and later, in the factor L. In the presentation of the algorithm that follows, we treat all matrices as if they are n-by-n and symmetric (except L, which is lower triangular). However, our code only stores the lower part of symmetric matrices. In addition, during the processing of supernode j, the only nonzero columns in $L^{(j)}$ are $\Omega_j \cup \Psi_j$; the rest are identically zero and not represented at all. We begin the processing of supernode j by initializing $L^{(j)}$ to zero. We then copy columns Ω_j of A to $L^{(j)}$,

$$L^{(j)}_{*,\Omega_j} = A_{*,\Omega_j} .$$

The algorithm now recursively traverses T_j and updates $L^{(j)}_{*,\Omega_j}$,

$$L^{(j)}_{*,\Omega_j} = L^{(j)}_{*,\Omega_j} - \sum_{k\in T_j} L_{*,\tilde{\Omega}_k} D_{\tilde{\Omega}_k,\tilde{\Omega}_k} L^T_{\tilde{\Omega}_k,\Omega_j} .$$

Note that each term in the sum updates only the columns $\tilde{\Xi}_k \cap \Omega_j$, not all of Ω_j. The contributions $L_{*,\tilde{\Omega}_k} D_{\tilde{\Omega}_k,\tilde{\Omega}_k} L^T_{\tilde{\Omega}_k,\tilde{\Xi}_k\cap\Omega_j}$ are first computed using a dense matrix-matrix multiplication routine, and then scatter-added to $L^{(j)}$. The recursive traversal continues to the children of k only if $\tilde{\Xi}_k \cap \Omega_j \neq \emptyset$; otherwise, it returns to k's parent without searching the subtree rooted at k for supernodes that update $L^{(j)}$; there are none [Liu 1990].

Next, the delayed columns from each child ℓ of j are copied into $L^{(j)}$,

$$
\begin{aligned}
L^{(j)}_{*,\Delta_\ell} &= L^{(j)}_{*,\Delta_\ell} + L^{(\ell)}_{*,\Delta_\ell} \\
&= L^{(j)}_{*,\Delta_\ell} + \left(A_{*,\Delta_\ell} - \sum_{i \in T_\ell} L_{*,\tilde{\Omega}_i} D_{\tilde{\Omega}_i,\tilde{\Omega}_i} L^T_{\tilde{\Omega}_i,\Delta_\ell} \right) .
\end{aligned}
$$

Note that the columns in Ψ_j are not updated during the processing of vertex j. Since these columns were delayed from one of j's children, say ℓ, all the updates from columns in the subtree rooted at ℓ have already been applied to these columns, and columns in subtrees rooted at other children of j can never update these columns. Therefore, these columns are fully updated.

Now that $L^{(j)}_{*,\Omega_j \cup \Psi_j} = A^{(j)}_{*,\Omega_j \cup \Psi_j}$ (i.e., columns $\Omega_j \cup \Psi_j$ in $L^{(j)}$ are now columns of the reduced matrix), the algorithm tries to eliminate the columns in $\Omega_j \cup \Psi_j$. The process repeatedly searches for a column or pair of columns that can be eliminated. We explain the search strategy and admissibility criteria to follow. Once one or two columns have been eliminated, the remaining uneliminated columns in $\Omega_j \cup \Psi_j$ are updated to allow for future pivoting decisions. The factorization of supernode j may fail to eliminate some of the columns in $\Omega_j \cup \Psi_j$ (essentially, those with relatively large elements in rows outside of $\Omega_j \cup \Psi_j$). These columns are put into Δ_j and delayed to $\pi(j)$. The set of columns $\tilde{\Omega}_j$ that has been successfully factored is added to the factor matrices L and D,

$$
\begin{aligned}
L_{*,\tilde{\Omega}_j} &= L_{*,\tilde{\Omega}_j} + L^{(j)}_{*,\tilde{\Omega}_j} , \\
D_{*,\tilde{\Omega}_j} &= D_{*,\tilde{\Omega}_j} + D^{(j)}_{*,\tilde{\Omega}_j} .
\end{aligned}
$$

There is a simpler, but less efficient, way to handle column delays. The algorithm can simply propagate to $\pi(j)$ the index set Δ_j, but discard the columns $L^{(j)}_{*,\Delta_j}$ themselves. The parent $\pi(j)$ would then read these columns from A and update them as it does to columns in $\Omega_{\pi(j)}$. This is simpler, since all the columns of $L^{(j)}$ now receive exactly the same treatment, whereas our strategy treats columns in $\Omega_{\pi(j)}$ differently than those in $\Psi_{\pi(j)}$. However, in the former strategy a contribution from supernode i to a column in Ω_j is computed and added to the column during the processing of supernode j, then again during the processing of $\pi(j)$ if the column is delayed, and so on, until we reach the supernode where the column is eliminated. At each delay, the contributions to the column are discarded and recomputed at the parent supernode. Due to this increased cost, we decided to use the previous strategy.

4. PIVOT-ADMISSIBILITY CRITERIA AND PIVOT-SEARCH STRATEGIES

This section describes the pivot-admissibility criteria and pivot-search strategies that our codes use. A 1-by-1 or 2-by-2 pivot that is small compared to the rest of the elements in its column (or pair of columns) may cause growth in the reduced matrix. This growth is what causes instability in the factorization, and why columns may have to be delayed.

This principle implies that in both multifrontal and left-looking factorizations, the pivots for supernode j can only come from $\Omega_j \cup \Psi_j$ because

if $\gamma_q = 0$
   then the column q is already factored
else if $\left|\bar{A}_{q,q}\right| \geq \hat{\alpha}\gamma_q$
   then use q as a 1-by-1 pivot
else if $\left|\bar{A}_{p,p}\right| \geq \hat{\alpha}\gamma_p$
   then use p as a 1-by-1 pivot
else if

$$\left|\bar{A}_{p,p}\bar{A}_{q,q} - \bar{A}_{p,q}\bar{A}_{p,q}\right| \geq \hat{\alpha} \ \max \ \{\left|\bar{A}_{p,p}\right|\gamma_q + \left|\bar{A}_{p,q}\right|\gamma_p \, ,$$
$$\left|\bar{A}_{q,q}\right|\gamma_p + \left|\bar{A}_{p,q}\right|\gamma_q\}$$

   then use $\bar{A}_{\{q,p\},\{q,p\}}$ as a 2-by-2 pivot
else
   no pivot found; repeat search using next column
end if

Fig. 3.   Our pivot-admissibility test. This strategy is from MA27 [Ashcraft et al. 1998], see Figure 3.3. The scalar $\hat{\alpha}$ is a threshold that controls growth. A high value prevents growth in the factor and hence enhances numerical stability, but may cause many columns to be delayed. We use the value $\hat{\alpha} = 0.001$.

uneliminated columns outside this set have not yet been assembled. That is, if $q \notin \Omega_j \cup \Psi_j$, then the terms in Equation (1) have not yet been summed, and some of them may be unavailable because the corresponding supernodes have not yet been factored. If a column has not been assembled, its admissibility for elimination, either alone or with another column, cannot be ascertained. This is why columns might be delayed.

Our codes use the pivot-admissibility criteria from MA27 [Duff and Reid 1982]. Our presentation of these criteria is based on Ashcraft et al. [1998]. The literature also contains a number of other strategies (including those of Ashcraft et al. [1998], which is the most recent algorithmic article on this subject). We used this particular strategy since it is both simple and effective and backed by extensive research. For further details on this and other strategies, see Higham [2002], Chapter 11, and references therein.

The pivoting strategy we use works as follows. Let q be an uneliminated column in $\Omega_j \cup \Psi_j$. We denote by Φ_j the set of columns in $\Omega_j \cup \Psi_j$ eliminated thus far. We denote by $\bar{A}$ the current reduced matrix,

$$\bar{A} = A - L_{*,\Phi_j} D_{\Phi_j,\Phi_j} L_{\Phi_j,*}^T - \sum_{k \leq j} L_{*,\bar{\Omega}_k} D_{\bar{\Omega}_k,\bar{\Omega}_k} L_{\bar{\Omega}_k,*}^T \ .$$

We also denote

$$p = \arg \max_{i \in \Omega_j \cup \Psi_j \backslash \Phi_j} \left|\bar{A}_{i,q}\right|$$
$$\gamma_q = \max_{\substack{i \neq q \\ i \in \Omega_j \cup \Psi_j \backslash \Phi_j}} \left|\bar{A}_{i,q}\right| = \left|\bar{A}_{p,q}\right|$$
$$\gamma_p = \max_{\substack{i \neq p \\ i \in \Omega_j \cup \Psi_j \backslash \Phi_j}} \left|\bar{A}_{i,p}\right| \ .$$

Figure 3 specifies our pivot-admissibility test, using this notation.

The aforementioned pivot-admissibility criteria determines, given a column q, whether q and/or $p = p(q)$ can be eliminated. But how do we select q?

Our pivot-search strategy is based on that in Figure 3.4 of Ashcraft et al. [1998], which in turn was based on that of MA27 [Duff and Reid 1982]. We try to eliminate the columns of a supernode in order. After all the columns in a supernode have been examined, we try to eliminate them again, in the same order. Any column that cannot be eliminated in the second pass is delayed. This strategy is less exhaustive than that of Figure 3.4 in Ashcraft et al. [1998] because the latter reexamines all uneliminated columns whenever a column is successfully eliminated, and reorders the columns to delay the reexamination of a failed column. We acknowledge that a more aggressive strategy might be able to successfully factor more matrices or reduce delays, but our focus in this research is on out-of-core and not pivoting issues.

Our code uses a stability threshold $\hat{\alpha} = 0.001$. This value is smaller than the value used in MA27 [Ashcraft et al. 1998] and some subsequent codes. We selected this value based on limited experimentation. A later experiment on additional matrices (the results of which are shown in Figure 9) validated this choice, in that the code produced an unstable factorization on only one matrix from this set. A larger threshold would have caused more delays, which slows the factorization and might cause it to run out of memory.

5. THE OUT-OF-CORE FACTORIZATION ALGORITHM

When the factor L does not fit in main memory, out-of-core algorithms store factored supernodes on disks. In a left-looking algorithm, a factored supernode k is read into memory when it needs to update another supernode j. In a naive algorithm, supernode k is read from disk many times, once for each supernode that it updates. More sophisticated algorithms [Gilbert and Toledo 1999; Rotkin and Toledo 2004] maintain in main memory a set of partially updated but yet-unfactored supernodes, called a *compulsory subtree* (sometimes called a panel). This is a connected subtree of the elimination tree. These algorithms read from disk the supernodes that must update one of the leaves of the current subtree, say j. A supernode k that is read updates the leaf j for which it was brought to memory, then all the other supernodes in the subtree that k updates, and is then evicted from memory. Once j is fully updated, it is factored, updates all other supernodes in memory, and is evicted. Supernode j is now pruned from the subtree, and the factorization continues with another leaf. This strategy allows these algorithms to update many supernodes whenever a factored supernode is read into main memory. Such algorithms are not pure left-looking, but rather hybrids of left- and right-looking updates. They are classified as left-looking because right-looking updates are only applied to those supernodes that continue to reside in main memory until they are factored; partially updated supernodes are never written to disk.

The next subsection explains how we adapt this strategy to the factorization of symmetric-indefinite matrices. Our algorithm differs from the symmetric positive-definite algorithms of Rothberg and Schreiber [1999] and Rotkin and Toledo [2004] not only in that it can factor indefinite matrices, but also in some

aspects of the automatic planning of the factorization schedule. The second subsection highlights these differences.

5.1 The Left-Looking Out-Of-Core Symmetric-Indefinite Algorithm

Our out-of-core algorithm applies a left-looking subtree-oriented strategy to the out-of-core factorization of symmetric-indefinite matrices. The algorithm works in phases. At the beginning of each phase, main memory contains no supernodes at all. The supernodes that have already been factored are stored on-disk. The algorithm begins a phase by selecting a forest of connected leaf-subtrees of the residual etree (the etree of yet-unfactored supernodes). By a leaf-subtree, we mean a subtree whose leaves are all leaves of the residual etree; the leaves of the leaf-subtree are either leaves in the full etree, or all their children have already been factored. The algorithm then allocates in-core supernode matrices for the supernodes in the subtree and reads the columns of A into them. Then, the algorithm uses the general strategy outlined in the previous paragraphs to factor the supernodes of this forest one-at-a-time. Whenever a supernode is factored, it updates its ancestors in the subtree and is evicted from main memory. Hence, when the phase ends, no supernodes reside in memory, and a new phase can begin.

The application of this strategy to symmetric-indefinite factorizations faces two challenges. The first and most difficult lies in selecting the next forest to be factored. Delaying a column often causes additional fill within L, so the amount of memory required to store supernodes, even if they are packed and contain no zeros, grows. Therefore, it is impossible to determine in advance the exact final size of each supernode. As a consequence, the forest-selection procedure cannot ensure that the chosen forest will fit into main memory.

Our new algorithm addresses this issue in two ways. First, when a column is delayed, we update our data structure to indicate that the column has become a member of the parent supernode. This ensures that at the beginning of the next phase, the algorithm that partitions the matrix into subtrees uses the most up-to-date information regarding the size of the supernodes. They might continue to expand after the forest is selected, but at least all the expansion that has already occurred is accounted for. Second, the partitioning procedure only adds supernodes to the subtree as long as the combined predicted size of the subtree is, at most, 75% of the available amount of main memory (after explicitly setting aside memory for other data structures of the algorithm). This helps minimize the risk that supernode expansion will overflow main memory. Normally, if the subtree overflows, this will cause paging activity and some slowdown in the factorization, but it could also lead to memory-allocation failure. As in Rotkin and Toledo [2004], we limit the size of each supernode to help ensure that an admissible forest can always be found.

The other difficulty lies in delaying columns across subtree boundaries. Suppose that columns are delayed from the root supernode j of a subtree. The next forest need not include $\pi(j)$, so there is no point in keeping these columns in memory, where they use up space, but are not quickly used. Instead, we write them to disk and read them again, together with the factored columns of j,

when j updates $\pi(j)$. They will not be needed again (due to a limitation in our high-level I/O library [Rotkin and Toledo 2004], we actually write j again to disk, without the delayed columns, once the delayed columns, have been added to $\pi(j)$).

5.2 Comparison with Algorithms for Symmetric Positive-Definite Matrices

Out-of-core factorization algorithms for sparse symmetric positive-definite algorithms can partition the entire factor into compulsory subtrees prior to numeric factorization. When the matrix is positive definite, there is no need to delay columns, so the size of each supernode is known in advance. This allows the partitioning algorithm to decompose the etree into subtrees before factorization begins. This has been done by Gilbert and Toledo [1999], Rotkin and Toledo [2004], and in a more limited way, by Rothberg and Schreiber [1994]. As we have explained, this is not possible in the indefinite case, so we adopt a dynamic partitioning strategy.

We also note that Gilbert and Toledo [1999] and Rotkin and Toledo [2004] actually used a more sophisticated partitioning technique than the precceding one we described. A supernode only updates its ancestors in the etree. Therefore, there is no benefit in simultaneously storing in memory supernodes that are not in a descendant-ancestor relationship. Hence, Gilbert and Toledo [1999] and Rotkin and Toledo [2004] allow compulsory subtrees to be larger than the amount of available main memory, and they page supernodes both in and out of memory, without incurring extra I/O. This reduces the total amount of I/O. Since experiments in Rotkin and Toledo [2004] have shown that the reduction is not highly significant, however, we have not adopted this strategy in the new indefinite code.

5.3 Implementation

Our implementation of the out-of-core indefinite algorithm is an adaptation of the sparse Cholesky code in Rotkin and Toledo [2004], and in particular, the new code now part of TAUCS, a suite of publicly available sparse linear solvers.[1] We use the same high-level I/O library, which is based on a disk-resident data structure called a *store*. The algorithm is implemented in C, with calls to level-2 and-3 basic linear algebra subroutines (BLAS).

To factor individual supernodes, which are stored as rectangular dense matrices, we have developed a specialized blocked dense code. The code implements the pivoting strategy explained in Section 4. It is right-looking and blocked so as to exploit level-3 BLAS and achieve high performance. The blocking strategy is based on the LAPACK code DSYTRF, a blocked Bunch-Kaufman symmetric-indefinite factorization code. We could not use LAPACK code mainly because our code actually factors the diagonal block of a rectangular matrix, rather than a square matrix, and elements in the subdiagonal block affect the admissibility of pivots (in an LU factorization with partial pivoting, such elements can be used as pivots, but doing so here would ruin the symmetry). In addition,

---

[1]http://www.tau.ac.il/∼stoledo/taucs/

our pivoting strategy allows pivots with smaller norms than those of DSYTRF in order to reduce both the number of delayed columns and the additional fill that follows. We use an LU factorization with partial pivoting of 2-by-2 blocks for stability. In DSYTRF the use of an explicit inverse is numerically sound, since the 2-by-2 diagonal block satisfies certain constraints (see Theorem 11.3 of Higham [2002]); an LU factorization with partial pivoting is applicable to a wider choice of pivots.

Our implementation also includes a multiple righthand-sides solve routine. Once the factor has been computed and stored on-disk, the time it will take to solve a linear system is determined primarily by the time it takes to read the factor from disk. The factor must be read twice: once for the forward solve and once for the backward solve. By solving multiple linear systems of the same coefficient matrix during one read-solve process, we can amortize the cost of reading the factor over multiple solves. A solve with multiple righthand-sides can also exploit fast level-3 BLAS, whereas one for a single righthand-side can use, at most, the level-2 BLAS. Even for fairly large numbers of righthand-sides (e.g., 25 in the experiments that follow), the solution time is dominated by the disk-read time, thus the marginal cost of simultaneously solving additional linear systems is low.

Many applications can exploit the code's ability to efficiently solve a large number of linear systems of the same coefficient matrix. For example, there are several shift-invert eigensolvers that solve multiple indefinite linear systems in every iteration, such as block Lanczos algorithms and subspace iteration (see Stewart [2001] and references therein).

Finally, we mention that the new additions to TAUCS include both one out-of-core and two in-core sparse symmetric-indefinite factorization codes (the latter including one multifrontal and one left-looking).

6. TESTS AND RESULTS

We now describe the experimental results. The goal of these experiments is to demonstrate that our implementation of the new algorithm performs well, and to provide a deeper understanding of the behavior of the algorithm.

The experiments are divided into two sets. The first presents the performance of our in-core implementation of the algorithm and the in-core components of the out-of-core algorithm. The objective of this set of experiments is to establish a known baseline for the in-core algorithms for later use in assessing the performance of the out-of-core algorithm. We compare the performance of our in-core code to those of two other recent, well-known high-performance codes, MUMPS [Amestoy et al. 2001; 2000; 2003] and PARDISO [Schenk and Gärtner 2004; Röllin and Schenk 2006]. We also compare the performance of our symmetric-indefinite and Cholesky in-core codes, of left-looking and multifrontal variants, and of kernels for the in-core factorization of dense diagonal blocks.

In the second set of experiments we compare the performance of our out-of-core code with that of our best in-core code so as to measure the performance penalty imposed by disk I/O. Other experiments in this set explore other aspects of the algorithm. One experiment compares the algorithm to the out-of-core
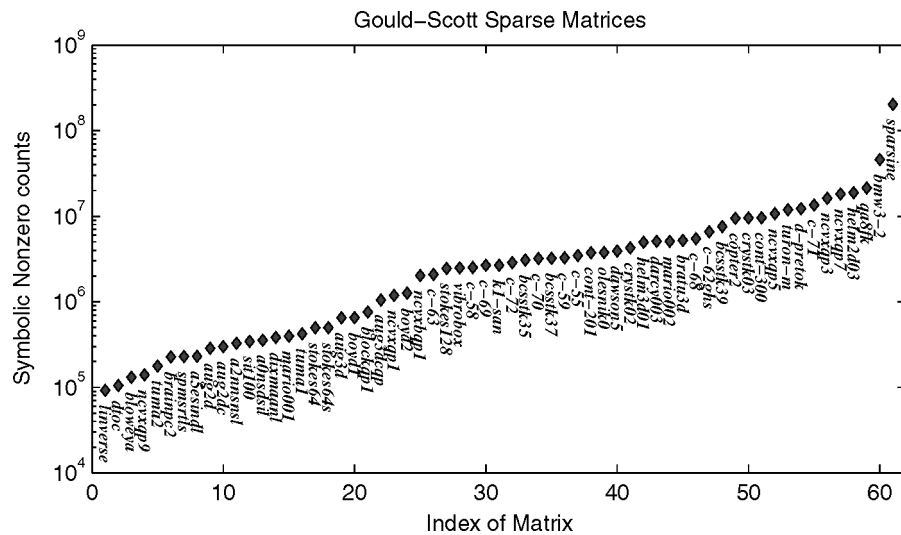
Fig. 4.   Sparse test matrices from the Gould-Scott [2004] study. The x axis is simply an index; these matrices are shown using the same x axis in all subsequent graphs. The y axis shows, for each matrix A, the number of nonzeros in the Cholesky factor of $A + \sigma I$ for a large enough σ, where A is reordered using METIS [KARYPIS AND KUMAR 1998].

sparse Cholesky algorithm of Rotkin and Toledo [2004] in order to measure the effect of indefiniteness on the performance of out-of-core sparse factorization codes. Other experiments explore the effects of the inertia (number of negative eigenvalues) and main-memory size on the performance of the algorithm. We also present results that show the performance benefit of simultaneously solving multiple linear systems of the same coefficient matrix.

Before we present the results of the experiments, however, we describe the matrices and the computer that we used.

6.1 Test Matrices

We performed our experiments on three sets of symmetric matrices; some sets containing both indefinite and positive-definite matrices.

Figure 4 shows the first set of matrices. These are the 61 indefinite matrices from the study of direct sparse symmetric solvers by Gould and Scott [2004]. The experiments with these matrices are designed to assess the ability of our solver to factor difficult indefinite matrices. Most of these matrices are not large enough to evaluate the out-of-core solver (because they can be factored in-core), but they do reveal weaknesses in the delayed pivoting strategy. The matrices are ordered in Figure 4 by the number of nonzeros in the Cholesky factor of $A + \sigma I$ for a large enough σ. The same ordering is used in all other plots. In other words, we identify these matrices by their index in Figure 4.

The second set of test matrices includes additional large sparse matrices. In the graphs, this set is called *additional* matrices. It consists of several matrices,

Table I. Additional Test Matrices from Real-World
Applications

Name	Source	SPD?	dim(A)	nnz(A)
s0tau	Bustany	no	53794	715858
sme1	Ekroth	no	89337	571914
inline-1	PARASOL	yes	503712	18660027
ldoor	PARASOL	yes	952203	23737339
audikw-1	PARASOL	yes	943695	39297771
femlab1	Ekroth	no	1063680	15490073
af-shell10	Schenk	no	1508065	27090195
cont5-2	Schenk	no	2003000	8001004

Some matrices are from the PARASOL test-matrix collec-
tion (www.parallab.uib.no/parasol/data.html), some were do-
nated by Ismail Bustany from *Barcelona Design*, some were do-
nated by Anders Ekroth from *Comsol*, and some by Olaf Schenk.
The third column specifies whether the matrices are symmetric
positive-definite, the fourth their dimension, and the fifth the
number of nonzeros in their lower triangle.

listed in Table I, which arise in real-world applications, as well as a few synthetic
ones whose graphs are regular three-dimensional meshes. Figure 5 shows all
the matrices in this set. To maximize the utility of real-world matrices, some
of which are positive definite, we generated indefinite from definite matrices
by shifting the diagonal. The synthetic meshes include both positive-definite
(which are discretizations of the Laplacian on a three-dimensional mesh using
a 7-point stencil) and indefinite matrices. The indefinite synthetic matrices
are generated by using the same underlying graph, but assigning symmetric
random values (uniform in [0, 1]) to the elements of each matrix. These matrices
tend to have roughly $n/2$ positive and $n/2$ negative eigenvalues, where n is the
dimension of the matrix. Some of the meshes that we use are perfect cubes,
such as 80-by-80-by-80, and some are longer in one dimension than in others,
such as 500-by-50-by-50. Generally speaking, perfect cubes lead to more fill in
the factorization than meshes with large aspect ratios.

The third set of symmetric matrices, which we use in only one limited ex-
periment, consists of dense matrices. They are shown in Figure 6. These dense
matrices are made up of symmetric positive definite- and symmetric-indefinite
matrices, the latter of which have about half positive and half negative eigen-
values.

6.2 Definitions of Success and Failure

Some of the codes failed on some of the matrices, either due to lack of memory or
numerical instability. When a code failed, the corresponding data point is simply
not shown in the graphs. We define failure in the case of TAUCS (and other direct
methods that do not rely on iteration) as being unable to obtain a relative
residual of 10^{-8} or less after forward-elimination and back-substitution. In
fairness to the design of PARDISO, we define failure for PARDISO as the inability to
achieve a relative residual of 10^{-8} or less after two steps of iterative refinement.
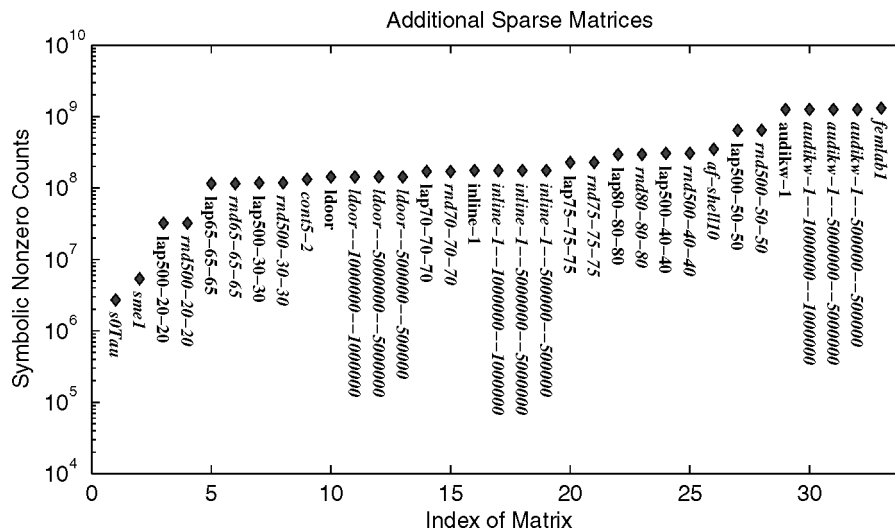
Fig. 5. Additional sparse test matrices, both real-world and derived from three-dimensional meshes. Names printed in upright (roman) font signify positive-definite matrices, whereas those printed in italics signify indefinite matrices. Positive-definite matrices whose graph is an x-by-y-by-z mesh are named "lapx-y-z" and indefinite meshes are named "*rndx-y-z*." The numbers that follow two dashes are shift values for matrices whose diagonal was shifted to make them indefinite.
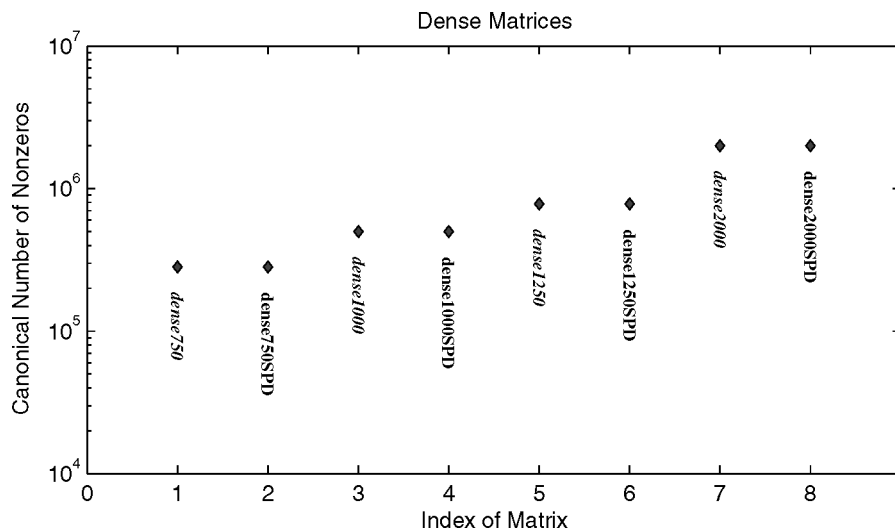

Fig. 6. Dense test matrices. The axes are the same as those of Figure 5. The matrix name consists of its dimension and whether it is symmetric positive-definite (SPD) or symmetric-indefinite.

6.3 Test Environment

We performed the experiments on an Intel-based workstation. This machine has a 2.4 GHz Pentium 4 processor with a 512 KB level-2 cache and 2 GB of main memory (dual-channel with DDR memory chips). The machine runs Linux

with a 2.4.22 kernel, and was configured to run without virtual memory, so no paging by the operating system would be possible. This was done to ensure that the only I/O performed is explicit I/O by our code, rather than implicit I/O performed by the virtual memory mechanism of the operating system.

We compiled our code with the GCC C compiler, version 3.3.2, and the -O3 compiler option. We used the implementation of BLAS (basic linear algebra subroutines) written by Kazushige Goto, version 0.9.[2] This version exploits vector instructions on Pentium 4 processors (these instructions are called SSE2 instructions). This setup allows our code to compute Cholesky factorization of large sparse matrices at rates exceeding 3×10^9 flops (e.g., the Laplacian of a 65-by-65-by-65 mesh).

The graphs and tables use the following abbreviations:

—TAUCS (our sparse code), MUMPS (MUMPS version 4.3), PARDISO (PARDISO version 1.2.1);

—LL (left-looking), MF (multifrontal);

—OOC (out-of-core), IC (in-core);

—SPD (symmetric positive-definite); and

—LL ^{T} and LDL ^{T} for the Cholesky and symmetric-indefinite factorizations, respectively.

6.4 Baseline Tests

To establish a performance baseline for our experiments, we compare the performance of our code, called TAUCS, to three in-core codes. One of the in-core codes is that of sparse factorizations in TAUCS, both Cholesky and symmetric-indefinite. Our in-core codes can use either a left-looking or multifrontal algorithm, and we test both. The other codes we use for the baseline tests are the MUMPS version 4.3 [Amestoy et al. 2000, 2001, 2003] and PARDISO version 1.2.1 [Schenk and Gärtner 2004; Röllin and Schenk 2006]. We use METIS[3] [Karypis and Kumar 1998] version 4.0 to symmetrically reorder the rows and columns of all the matrices prior to factoring them. We tested the sequential versions of both MUMPS and PARDISO with options that instruct these codes to use METIS for preordering the matrix, and to inform the codes that the input matrix is symmetric, and positive definite when appropriate. We used the default values for all other run-time options.

The algorithms used by MUMPS and PARDISO are quite different from the factorization described in this article. MUMPS only uses 1-by-1 pivots, so there are both symmetric-indefinite matrices that it simply cannot factor, and those that it can, but not in a stable way. Both TAUCS and PARDISO use both 1-by-1 and 2-by-2 pivots, but they differ in how they handle the columns for which suitable pivots were not found within the supernode. As discussed in Sections 4 and 5, TAUCS as well as most other direct factorization methods delay such columns so that a stable factorization of a *permutation* of the original matrix is obtained. In contrast, when faced with the same situation, PARDISO modifies the coefficient

---

matrix. It therefore obtains a factorization of a *perturbation* of the coefficient matrix. In the delayed-pivoting approach, delays often cause the number of nonzeros in the factor to grow beyond the number required for a Cholesky factorization of $A + \sigma I$ for a large σ. On the other hand, since the factorization of A is usually stable, PARDISO allows conventional condition-number analysis to predict the accuracy of the solution. In particular, small residuals are typically obtained, except for very poorly scaled matrices. In the perturbation approach, columns are not delayed, so the number of nonzeros in the factors depends less on numerics (there is some dependency due to pivoting within relaxed supernodes and the constraints imposed by static pivoting [Schenk and Gärtner 2004; Röllin and Schenk 2006]). When the matrix is perturbed, the factorization that the code computes is not a backward-stable factorization of A, thus, one or more steps of iterative refinement may be required to obtain satisfactory accuracy, and some problems may not be solvable. In the out-of-core setting, relying on iterative refinement may degrade performance significantly, since each refinement step requires reading the factor twice from disk.

We compiled MUMPS, which is implemented in Fortran 90, using Intel's Fortran Compiler for Linux version 7.1, and the compiler options specified in `makefile` provided by MUMPS for this compiler, namely, `-O`. We linked MUMPS with the same version of BLAS that was used for all other experiments.

We used a precompiled binary version of PARDISO obtained from one of PARDISO's authors, Olaf Schenk. It was linked with the same version of BLAS that was used in all other experiments.

The results of the baseline tests are shown in Figures 7 through 10. The results, both here and later, display performance in terms of *symbolic floating-point operations per second*. This metric merges two separate measures, sparsity and execution speed, into a single number. If one code achieves a symbolic rate of 2×10^9 and another achieves a rate of 4×10^9, then the latter runs in exactly half the time, independent of how dense the factor was that each code produced. We define the symbolic number of floating-point operations for a matrix A to be the number of floating-point operations in the Cholesky factorization of $A + \sigma I$, where σ is a real number large enough to shift the spectrum of A to the positive half of the real line. We compute the symbolic number of floating-point operations using a symbolic factorization of A. The y axis of our graphs measures the number of symbolic floating-point operations against the factorization time in seconds.

Figure 7 compares the performance of left-looking and multifrontal factorizations in TAUCS. The plot only shows a subset of the matrices, namely, those small enough to be factored in-core. The results show that left-looking codes, both Cholesky and symmetric-indefinite, are consistently faster. Therefore, in subsequent graphs we only show the performance of the faster left-looking algorithms.

Figures 8 and 9 show the performance of TAUCS relative to that of MUMPS and PARDISO. MUMPS only uses 1-by-1 pivots, so it produced unstable factorization of many matrices, for example, AUG2D, AUG2DC, AUG3D, BLOWEYA, BMW3-2, and so on. PARDISO perturbs the matrix when a pivot is too small, instead of delaying columns, and uses iterative refinement to solve linear systems using the
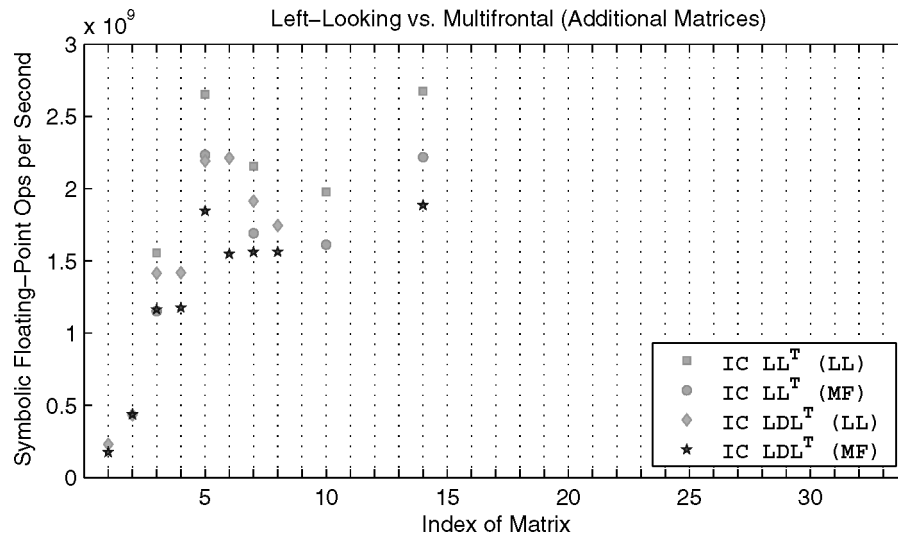
Fig. 7.   The performance of the in-core factorization codes in TAUCS on the additional sparse matrices. The figure only shows the performance on the subset of matrices that could be factored in core. The computational rates, in floating-point operations per second, are *symbolic rates*, as explained in the text. In particular, a higher rate indicates faster completion time, not a higher operation count.
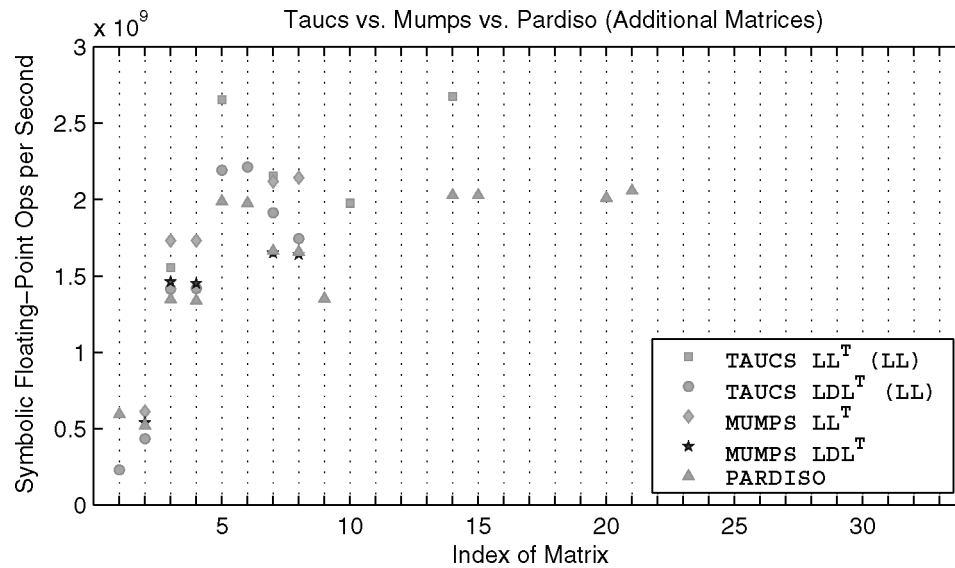

Fig. 8.   The performance of TAUCS versus that of MUMPS and PARDISO on the additional sparse matrices, and again in canonical rates.

factorizations of the perturbed matrix. Avoiding delays allows PARDISO to factor more matrices in-core than TAUCS can. In most cases, the perturbation is sufficiently small to allow iterative refinement to solve the linear system accurately. However, on some matrices, such as CRYSTK02 and CRYSTK03, PARDISO fails
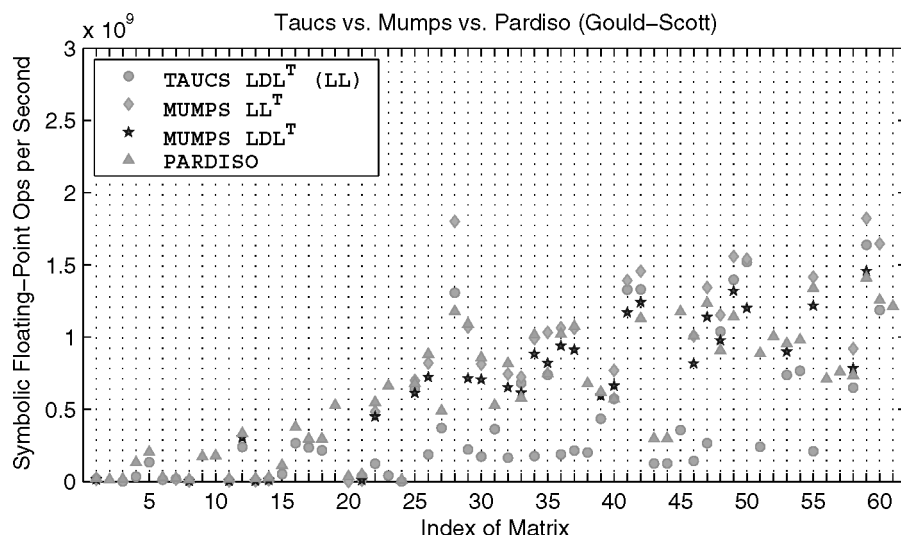
Fig. 9. The performance of TAUCS versus those of MUMPS and PARDISO on the Gould-Scott set of matrices.

to obtain a small residual, even with two steps of iterative refinement. Both TAUCS and MUMPS are able to obtain small residuals on these matrices. TAUCS uses delayed pivoting; the delays cause more fill and, in some cases, cause the code to run out of memory, even on matrices that PARDISO can factor in-core. On one matrix, DTOC, our code TAUCS computed a factorization, but failed to achieve a small residual (the relative norm of the residual was around 100; PARDISO achieved a small residual on this matrix). On three additional matrices, BRATU3D, CONT-201, and CONT-300, TAUCS produced a relative residual that was smaller than 10^{-8}, but not tiny (around 10^{-10}–10^{-8}). This shows evidence of some instability, but not to a catastrophic extent. The data also shows that the performance of the three codes is fairly similar; sometimes TAUCS, sometimes PARDISO, and sometimes MUMPS. This indicates, for the purposes of our experimental evaluation, that the performance of the in-core routines of TAUCS are comparable to those of MUMPS and PARDISO.

To summarize: MUMPS cannot solve many of the systems because it only uses 1-by-1 pivots; PARDISO failed numerically on two matrices, TAUCS failed numerically on one matrix, and is less memory-efficient than PARDISO because TAUCS uses delay pivoting.

Figure 10 shows that the routine we have implemented to factor the diagonal block of supernodes is efficient. The data that the figure presents compares the performance of five dense factorization kernels: LAPACK's POTRF (dense Cholesky), LAPACK's SYTRF (dense LDL^T symmetric-indefinite factorization), our new blocked factorization, an unblocked right-looking version of our new dense kernel, and MUMPS' kernel. The first four were called from within our sparse indefinite factorization code, but on a dense matrix with only one supernode. The data shows that our code slightly outperforms blocked LAPACK's factorization code, and is faster than MUMPS'.
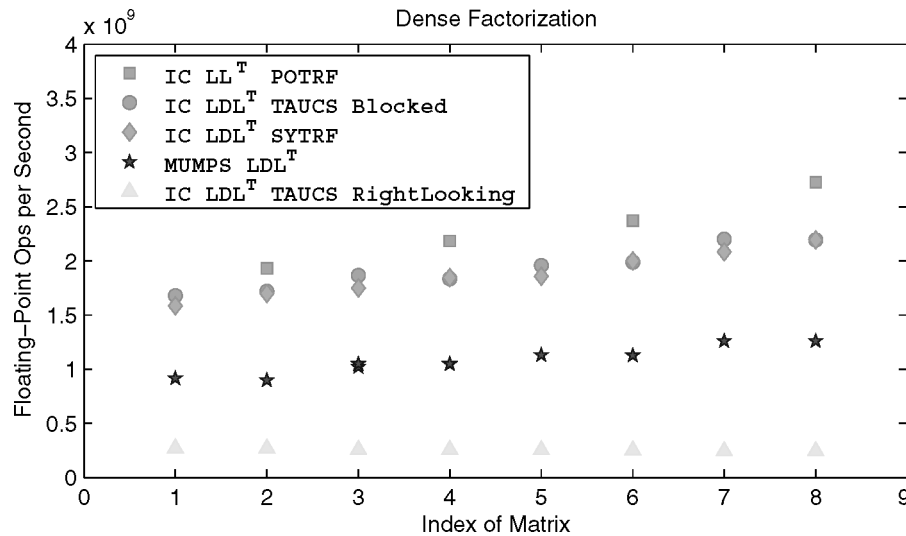
Fig. 10.   A comparison of dense symmetric-indefinite and Cholesky factorization kernels. The performance of TAUCS and MUMPS on dense matrices, and with four different kernels in TAUCS.

The data also indicates that TAUCS factors some sparse matrices faster than it factors dense ones (and hence faster than LAPACK factors dense matrices). The same holds true for MUMPS. These results, which are somewhat surprising, are most likely due to the fact that dense codes factor the matrix by partitioning it into fairly narrow blocks (20 columns by default). In the sparse codes, however, supernodes are sometimes much wider than 20 columns, which allows BLAS to achieve higher performance.

6.5 The Performance of the Out-of-Core Code

Having established the baseline performance of our codes, we now describe the experiments that evaluate the performance of the new out-of-core code.

Figures 11 and 12 present the performance of the new out-of-core symmetric-indefinite factorization algorithm. As expected, the performance of the code is always lower than that of in-core symmetric-indefinite and Cholesky codes, on the condition that the other codes do not break down. However, the performance difference between in-core and out-of-core symmetric-indefinite codes on large matrices is usually less than a factor of two, which suggests that the performance penalty paid for the extra robustness is acceptable. The performance difference between symmetric-indefinite and Cholesky codes is sometimes quite large, but this is not due to out-of-core issues. The out-of-core factorization code often runs at a rate between 1×10^9 and 2×10^9 floating-point operations per second.

As Figure 12 shows, however, on some matrices the performance of TAUCS is poor, sometimes less than 0.1×10^9 symbolic floating-point operations per second. In some cases, this poor performance reflects the catastrophic fill that is caused by the delayed pivoting scheme. For example, symbolic analysis of the
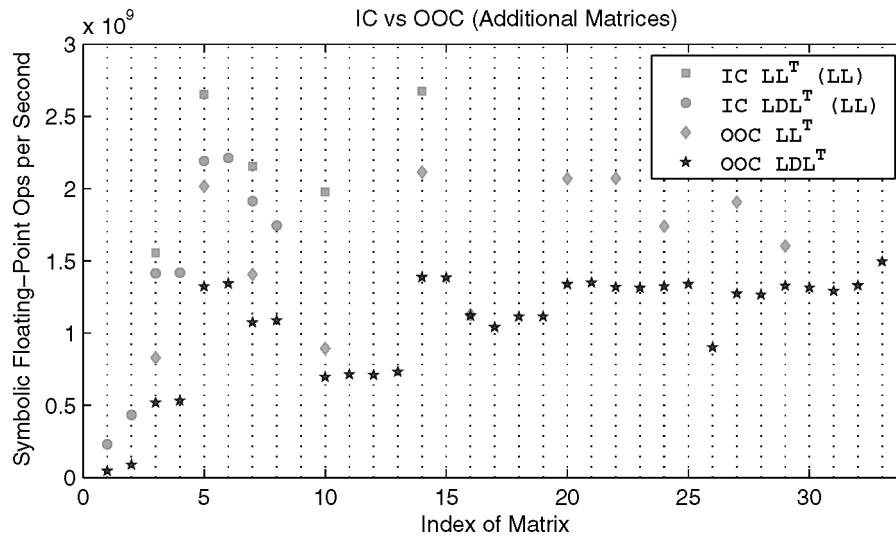
Fig. 11.   The performance of the new out-of-core symmetric-indefinite factorization code on the additional sparse matrices. For comparison, the graph also shows the performance of three other TAUCS codes: in-core symmetric-indefinite factorization and the in- and out-of-core Cholesky factorizations.
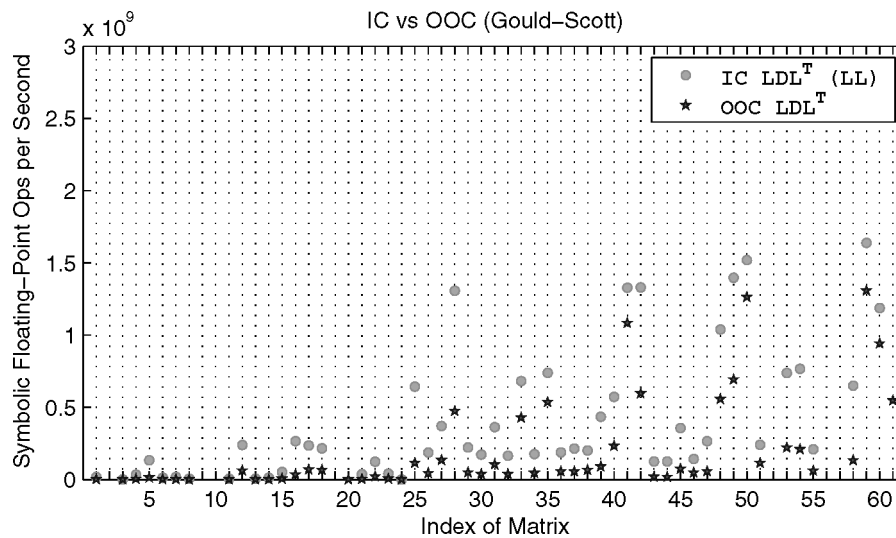

Fig. 12.   The performance of the new out-of-core symmetric-indefinite factorization code on the Gould-Scott matrices.

matrix BOYD2 predicted 1.26×10^6 nonzeros in the Cholesky factor of $A + \sigma I$ and that it would take 4.13×10^6 floating-point operations to compute this Cholesky factor. But the factorization of the indefinite A by TAUCS, both in-core and out-of-core, actually required 2.35×10^{11} floating-point operations and
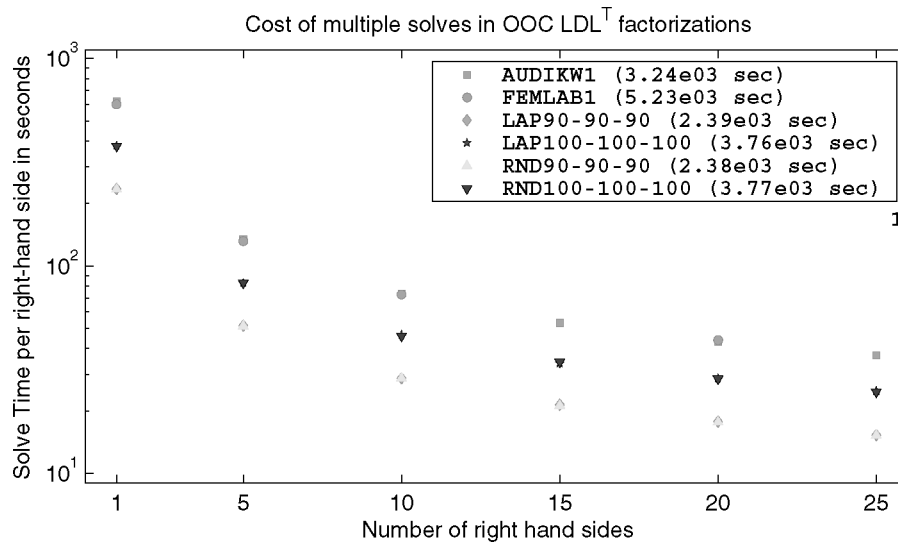
Fig. 13.   The performance of the out-of-core solve phase as a function of the number of righthand-sides. We tested our out-of-core symmetric-indefinite method with six different matrices. The numerical factorization time is stated in parentheses.

produced a factor that was over 50 times more dense than the Cholesky factor of $A + \sigma I$. This catastrophic fill is the result of our specific pivot-search strategy and pivot-admissibility criteria; PARDISO, for example, was able to factor this matrix quickly, in less than four seconds (versus 3189 for our out-of-core code), which probably means that its factorization did not suffer such fill. In other cases, poor performance appears to be caused by very narrow supernodes. For example, on the matrix C-62GHS our codes did delay some columns, but without a significant increase in either fill or operation count. Nonetheless, the running times of our in-core and out-of-core codes were 25 and 115 seconds, respectively, compared to only 5.36 for PARDISO. The lack of wide supernodes also caused large out-of-core running time, since it resulted in almost 250,000 I/O system calls, most of them to write the triangular factor to disk. A third reason for poor out-of-core performance is that large factors are easy to compute. For example, the factor of HELM2D03, with 19×10^6 nonzeros, took less than seven seconds to compute in-core, but due to its size (of 200 MB), took 33 seconds to factor out-of-core.

Figure 13 shows the performance of the solve phase for a few large matrices. When solving a single linear system, the solve time is dominated by the time required to read the factor from disk. However, the disk-read time can be amortized over multiple righthand-sides. When multiple linear systems are solved simultaneously, the solve time per system drops dramatically. The solve routine uses level-3 BLAS routines for matrix-matrix multiplication and for solving triangular linear systems.

Figure 14 shows that our factorization code is relatively insensitive to the inertia of the input matrix. The running times do not vary significantly when a matrix is shifted nor when its inertia changes.
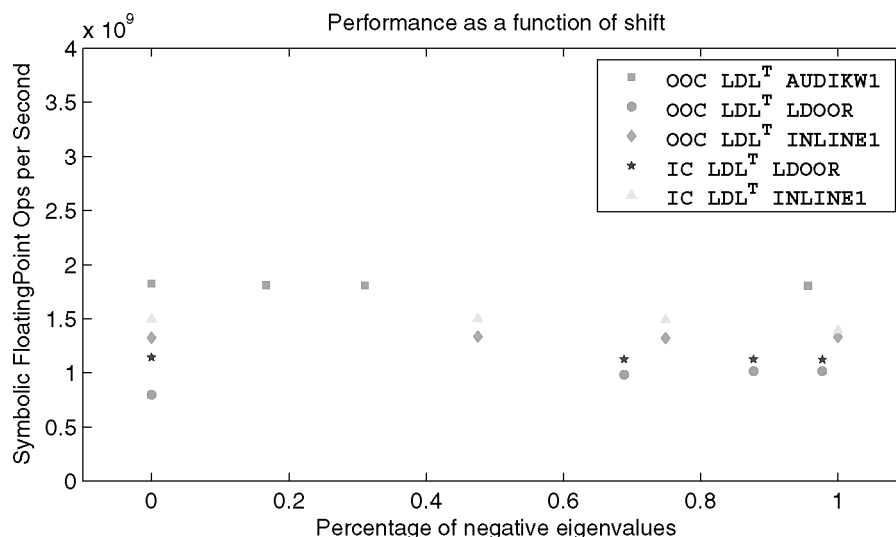
Fig. 14. The performance of the symmetric indefinite codes as a function of the percentage of negative eigenvalues in the matrix. The figure shows the performance of the code on shifted versions of three large positive-definite matrices.
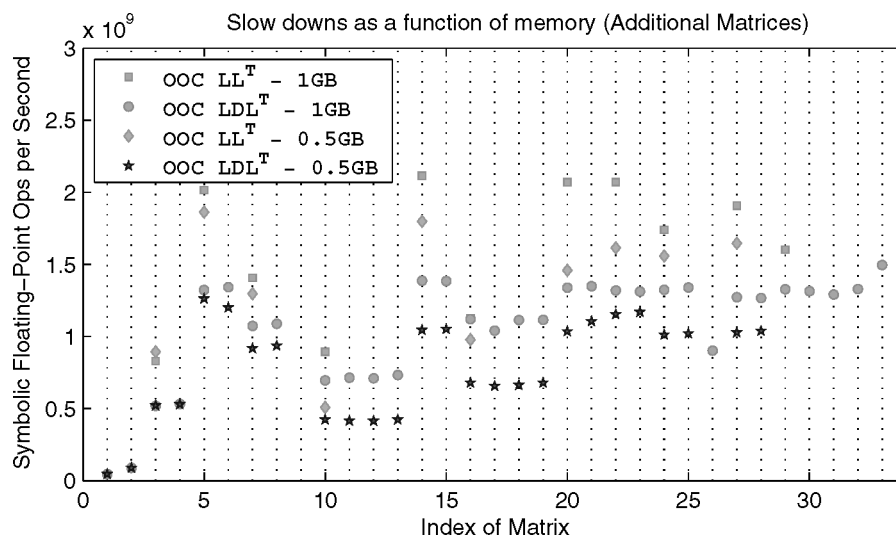

Fig. 15. The performance of the out-of-core code as a function of main memory size. The memory sizes shown in the legend are the target memory usage given to the code; the operating system itself had access to 2048 MB in one set of experiments and 1024 MB in the other set.

Figure 15 shows that out-of-core code slows down when it must run with limited main memory. To conduct this experiment, we configured the test machine such that the operating system was only aware of 1024 MB of the main memory. In the runs that we conducted with 1024 MB, we instructed the factorization code to use only 512 MB of memory (50% of the available memory), which was

the same percentage we used in the experiments with 2 GB of memory. On small matrices the slowdown is not significant, but on some it can reach a factor of 1.7. Furthermore, the largest matrices could not be factored at all with only 1024 MB of memory. Still, this experiment shows that even on a machine with a moderate amount of memory (1024 MB), our code can factor very large matrices. However, a larger memory helps, both in terms of the ability to factor very large matrices, and in terms of running times.

7. DISCUSSION AND CONCLUSION

This article has presented an out-of-core sparse symmetric-indefinite factorization algorithm. To the best of our knowledge, this is the first such algorithm to be presented in the literature, and the first sparse left-looking symmetric-indefinite algorithm to be demonstrated, although both out-of-core and left-looking codes do exist. Our implementation of the algorithm is reliable and performs well. It cannot factor of all the sparse symmetric-indefinite matrices that have been contributed to sparse matrix collections (almost no code can), but it can factor many. Its performance is slower than, but comparable to, that of recent high-performance in-core sparse symmetric-indefinite factorization codes and out-of-core sparse Cholesky codes.

The new code allows its users to directly solve very large sparse symmetric-indefinite linear systems, even on conventional workstations and personal computers. Even when the factor size is 10 GB or more, the factorization time is often less than an hour, and subsequent solves take about 10 minutes. The code's ability to simultaneously solve for multiple righthand-sides reduces even further the per-system cost of the solve phase.

REFERENCES

AMESTOY, P. R., DUFF, I. S., KOSTER, J., AND L'EXCELLENT, J. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl. 23*, 15–41.

AMESTOY, P. R., DUFF, I. S., L'EXCELLENT, J., AND KOSTER, J. 2003. MUltifrontal massively parallel solver (MUMPS version 4.3) user's guide. Available online from http://www.enseeiht.fr/lima/apo/MUMPS/doc.html.

AMESTOY, P. R., DUFF, I. S., AND L'EXCELLENT, J.-Y. 2000. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods Appl. Mechanics Eng. 184*, 501–520.

ASHCRAFT, C. AND GRIMES, R. 1999. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing* San Antonio, Tx. 10 pages on CD-ROM.

ASHCRAFT, C., GRIMES, R. G., AND LEWIS, J. G. 1998. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl. 20*, 513–561.

BUNCH, J. R. AND KAUFMAN, L. 1977. Some stable methods for calculating inertia and solving symmetric indefinite linear systems. *Math. Comput. 31*, 163–179.

BUNCH, J. R., KAUFMAN, L., AND PARLETT, B. N.   1976.   Decomposition of a symmetric matrix. *Numer. Math. 27*, 95–109.

DOBRIAN, F. AND POTHEN, A.   2006.   Oblio: Design and performance. In *Applied Parallel Computing: State-of-the-Art in Scientific Computing; Proceedings of the 7th International Workshop (PARA 2004) Lyngby, Denmark, held in June 2004*. Lecture Notes in Computer Science, vol. 3732. Springer Verlag, 758–767.

DUFF, I. AND REID, J.   1983a.   The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Softw. 9*, 302–325.

DUFF, I. S., ERISMAN, A. M., AND REID, J. K.   1986.   *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, UK.

DUFF, I. S. AND REID, J. K.   1982.   MA27: A set of Fortran subroutines for solving sparse symmetric sets of linear equations. Tech. Rep. AERE R10533, AERE Harwell, Didcot, Oxon, UK.

DUFF, I. S. AND REID, J. K.   1983b.   The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw. 9*, 302–325.

DUFF, I. S. AND SCOTT, J. A.   1999.   A frontal code for the solution of sparse positive-definite symmetric systems arising from finite-element applications. *ACM Trans. Math. Soft. 25*, 4 (Dec.), 404–424.

GILBERT, J. R. AND TOLEDO, S.   1999.   High-Performance out-of-core sparse LU factorization. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing* San Antonio, Tx. 10 pages on CDROM.

GOULD, N. I. M. AND SCOTT, J. A.   2004.   A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Soft. 30*, 3 (Sept.), 300–325.

HIGHAM, N. J.   2002.   *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Society for Industrial and Applied Mathematics, Philadelphia, PA.

IRONS, B. M.   1970.   A frontal solution scheme for finite element analysis. *Int. J. Numeri. Methods Eng. 2*, 5–32.

KARYPIS, G. AND KUMAR, V.   1998.   A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 359–392.

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

LIU, J. W. H.   1992.   The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Rev. 34*, 1, 82–109.

NATANZON, A., SHAMIR, R., AND SHARAN, R.   1998.   A polynomial approximation algorithm for the minimum fill-in problem. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC '98)*. ACM Press, New York, 41–47.

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

RÖLLIN, S. AND SCHENK, O.   2006.   Maximum-weighted matching strategies and the application to symmetric indefinite systems. In *Applied Parallel Computing: State-of-the-Art in Scientific Computing; Proceedings of the 7th International Workshop (PARA 2004), Lyngby, Denmark, held in June 2004*. Lecture Notes in Computer Science, vol. 3732. Springer Verl. 808–817.

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.

ROTHBERG, E. AND SCHREIBER, R.   1999.   Efficient methods for out-of-core sparse Cholesky factorization. *SIAM J. Sci. Comput. 21*, 129–144.

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.

SCHENK, O. AND GÄRTNER, K.   2004.   On fast factorization pivoting methods for sparse symmetric indefinite systems. Tech. Rep. CS-2004-004, Department of Computer Science, University of Basel. Submitted to *Electro. Trans. Numer. Anal.*

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

STEWART, G. W.   2001.   *Matrix Algorithms, vol. 2: Eigensystems*. SIAM, Philadelphia, PA.