TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

An Out-of-Core Sparse Symmetric Indefinite Factorization Method

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree of Tel-Aviv University by

Omer Meshar

The research work for this thesis has been carried out at Tel-Aviv University under the direction of Prof. Sivan Toledo

Abstract. This paper introduces a new out-of-core sparse symmetric indefinite factorization method.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 input-output 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 with a high-performance out-of-core sparse Cholesky code. More specifically, the new code provides a new capability that none of these existing codes has: it can factor symmetric indefinite matrices whose factors are larger than main memory; it is somewhat slower, but not by much. For example, it factors, on a conventional 32-bit workstation, an indefinite finite-element matrix whose factor size is about 10 GB in less than an hour.

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 1–2 hours.

This method allows us to solve linear systems $Ax{=}b$ with a single right-hand-side and linear systems $AX = B$ with multiple right-hand side efficiently and accurately. Linear systems with a symmetric indefinite coefficient matrix arise in optimization, in finite-element analysis, and in shift-invert eigensolvers (even when the matrix whose eigendecomposition is sought is definite).

Linear solvers that factor the coefficient matrix into a product of permutation, 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 either when the user has little expertise in iterative methods, or 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 the rows and columns of the matrix. Therefore, matrices are normally permuted into a form whose factors are relatively sparse prior to the factorization itself. Although the problem of finding a minimal-fill ordering is NP-complete, there exists effective heuristics that work well in practice (as well as provable approximations that have not been shown to work well in practice). Even when the matrix has been prepermuted using a fill-reducing permutation, the factor is often much larger (denser) than the matrix, and it may not fit in memory even when the matrix fits comfortably. When the factor does not fit within main memory, the user has three choices: either to resort to a so-called *out-of-core* algorithm, which stores the factors on disk, to switch to an iterative algorithm, or to switch to 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 the 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 the matrices are dense, relatively simple 1- and 2-dimensional partitions into blocks of consecutive rows and columns work well; when the 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 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 the 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 proposed. 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 [17] and by Rotkin and Toledo [18]. Gilbert and Toledo proposed a method for the more general problem of factoring a general sparse unsymmetric matrix [9]. This algorithm is more widely applicable than the algorithm that we present here, but it is also significantly slower. For earlier sparse out-of-core methods, see the references in the articles cited above.

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 [4], but to the best of our knowledge a left-looking formulation has never been described in the literature. We partition the matrix into compulsory subtrees [18] to achieve I/O efficiency, but the matrix is partitioned dynamically during the numeric factorization, to account for pivoting. (The method of [18] partitions the matrix statically before the numeric factorization begins.) To achieve a high computational rate, we have implemented a partitioned *dense* LDL^T factorization, which we use 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, 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 between 1 and 2 billion floating-point operations per second, *including the disk input-output time.*

The paper is organized as follows. The next section provides background on sparse symmetric indefinite factorizations. The section that follows presents our left-looking formulation of the factorization; we use this formulation in both in-core and out-of-core codes. Section 4 presents our new out-of-core algorithm and its implementation. Section 5 presents our experimental results, and Section 6 presents our conclusions.

2. Background

This section provides background material required for the rest of the thesis. The first part of this section describes the fundamentals of sparse symmetric indefinite matrices and factorizations. The second part presents the basics of Out-of-Core algorithms and their use. This part follows quite closely the observations of Sivan Toledo's survey on out-of-core algorithms [20].

2.1. **Sparse Symmetric Indefinite Matrices.** In many applications, there lies a need to solve linear systems, $Ax = b$, where A is the matrix coefficeint, b is the vector of right-hand-side values and x is the vector of variables we are looking for (in general, we might have $AX = B$, where B is multiple right-hand-sides) . When there are significantly less nonzeros then entries in A, we call A *sparse,* and we can exploit the sparseness of A by using special data structures, which consume less memory than regular dense matrices. In many linear systems, A is often *symmetric*, $a_{ij} = a_{ji}$, a fact that again can be exploited for better performance and less memory needs. (In this work we have used the compressed-column-storage scheme to represent sparse symmetric matrices .)

Symmetric matrices are classified into two groups: *positive-definite* matrices and *indefinite* matrices. A positive-definite matrix is a symmetric matrix A for which the quadratic $x^T Ax$ is positive for all nonzero vectors x, or , equivalently, for which all the eigenvalues are positive. In the positive definite case, there exists a lower-triangular matrix C such that $A = CC^T$. Factoring A into $CC^T$is called the *Cholesky factorization* of A. A symmetric matrix that has both positive and negative eigenvalues, is called indefinite. This is of course, by its definition, the more general case of the symmetric matrices. In this case, there exist a permutation matrix P, a unit lower-triangular matrix L (has 1's on the diagonal) and a block diagonal matrix D, with 1-by-1 and 2-by-2 nonzero blocks such that $A = PLDL^T P^T$. For example: (here $P = I$)

$$A = \begin{bmatrix} 0 & \epsilon & 0 \\ \epsilon & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & & \\ 0 & 1 & \\ \frac{1}{\epsilon} & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & \epsilon & \\ \epsilon & 0 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \frac{1}{\epsilon} \\ & 1 & 0 \\ & & 1 \end{bmatrix} = LDL^T$$

Finding such P,L and D is the goal of the symmetric indefinite factorization. In order to solve linear systems efficeintly, we first factorize the coefficeint A into a produce of permutation, triangular and diagonal (block-diagonal) matrices. This allowes us to utilize the simple and quick algorithms of solving triangular and diagonal linear systems. For example, let us assume that A is symmetric indefinite, and that we already factorized A into $A = PLDL^T P^T$. In order to solve $Ax = b$, we solve the following triangular and block-diagonal systems (after applying the permutations in P).

1) $Lz = b$ (for z)
2) $Dy = z$ (for y)
3) $L^T x = y$ (for x)

Each step in a triangular and block-diagonal system is simple - either a single or a couple of equations are to be solved. This factorization can be also used to compute the inertia of A [6].

2.2. **Sparse Symmetric Indefinite Factorizations.** This section examines the sparse symmetric indefinite factorization of A to $PLDL^T P^T$. The permutation P is computed during the factorization to ensure numerical stability. 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 growth in L is controlled, is usually insignificant when a partial pivoting strategy is used, like

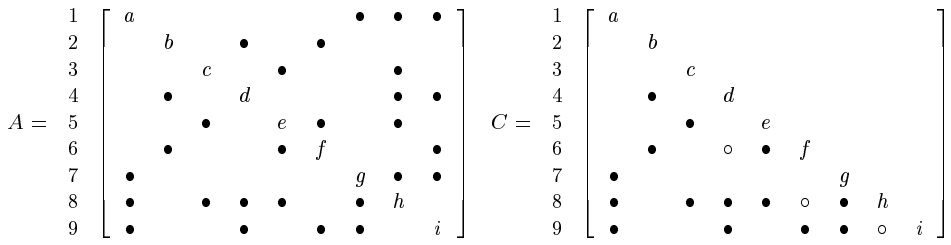

FIGURE 2.1. A sparse matrix example and its Cholesky factor. Each • represents an original nonzero in A, and ∘ , a fill in the cholesky factor C (a new nonzero).

the one proposed by Bunch and Kaufman [6]. When complete pivoting [7] or rook pivoting is used [5], 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 where one choice of P would lead to a factor with $\Theta(n^2)$ nonzeros and to a $\Theta(n^3)$ factorization cost, whereas another choice, equally good from a numerical point of view, would lead to $\Theta(n)$ work and nonzeros, where n is the order of 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 $Q^T AQ = PLDL^T P^T$. If the choice $P = I$ is numerically sound for the factorization of $Q^T AQ$, then the amount of fill in L is roughly the same as the amount of fill 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, and so-called *oxxo* blocks with zero diagonals 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 so that the resulting permutation QP is not too different from Q alone. We explain how the pivot search is constrained below.

A combinatorial structure called the *elimination tree of* A [19] (etree) plays a key role in virtually all symmetric factorization methods, both definite and indefinite [13]. When A is definite, the etree is used to predict the structure of the factor, to represent data dependences, and to schedule the factorization. In symmetric indefinite factorizations, the etree is used to constrain P so that L does not fill too much. The etree is also used in indefinite factorization in which P is thus constrained to represent data dependences, to schedule the factorization, `and to estimate the structure of the factor (but not to predict it exactly).`

The elimination tree is a rooted forest (tree unless A has a nontrivial block-diagonal decomposition) 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 illustration of this can be seen in Figures 2.1 and 2.2. 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 the state-of-the-art sparse indefinite factorization algorithms use a *supernodal decomposition* of the factor L, illustrated in Figure 2.3 [8, 15, 16]. The factor is decomposed into dense diagonal blocks and into the corresponding subdiagonal blocks, such that rows in the subdiagonal rows are either entirely zero

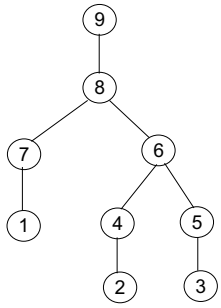

FIGURE 2.2. The elimination tree for the matrix example in figure 2.1

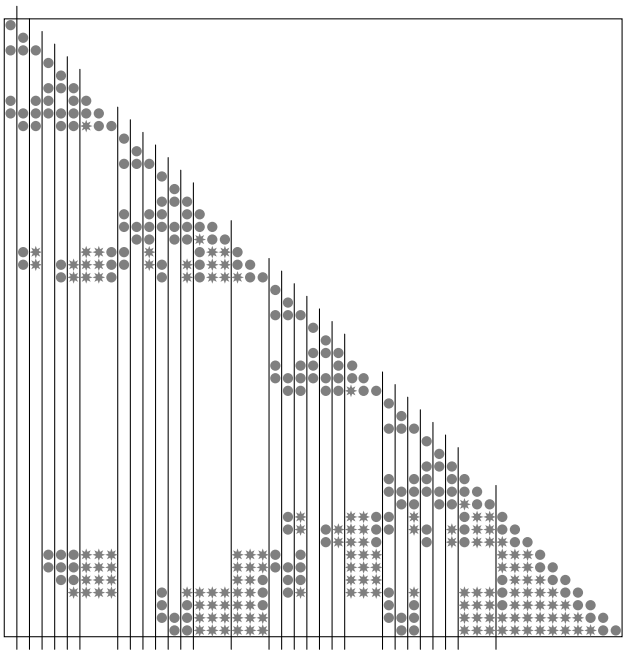

FIGURE 2.3. A fundamental supernodal decomposition of the factor of a matrix corresponding to a 7-by-7 grid problem, ordered with nested dissection. The circles correspond to elements that are nonzero in the coefficient matrix and the stars represent fill elements.

or almost completely dense. In an indefinite factorization, the algorithm computes a supernodal decomposition *for the Cholesky factor C before the numeric factorization begins*. The decomposition is refined during the numeric factorization, such that it is a correct decomposition of the actual factor L.

The supernodal decomposition is represented by a *supernodal elimination tree* or an *assembly tree*. In the supernodal etree, a tree vertex represents each supernode. The vertices are labeled 0 to $s-1$ using a postorder traversal, where s is the number of supernodes. 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 ignore oxxo pivots in this discussion.) We associate with supernode j the ordered set Ω_j of columns in the supernode in C, and the unordered set Ξ_j of nonzero row indices in the subdiagonal block of C. We denote by $\tilde{\Omega}_j$ and $\tilde{\Xi}_j$ the same sets in L. The ordering of indices in Ω_j is some ordering consistent with

a postorder traversal of the non-supernodal etree of A. We define $\omega_j = |\Omega_j|$ and $\xi_j = |\Xi_j|$. For example, the sets of supernode 29, the next-to-rightmost supernode in Figure 2.3, are $\Omega_{29} = (37, 38, 39)$ and $\Xi_{29} = \{40, 41, 42, 46, 47, 48, 49\}$.

If A is positive definite, the factorization algorithm factors all the columns in Ω_j during the processing of vertex j of the etree, and these columns update the columns in Ξ_j. When A is indefinite, however, the algorithm may be unable to factor all the columns in Ω_j during the processing of vertex j. The columns that are not factored are *delayed* to j's parent $\pi(j)$. The parent tries to factor the delayed columns; if this fails too, the failed columns are delayed again to $\pi(\pi(j))$; at the root, all the remaining columns are factored. In essence, a column is delayed when all the admissible pivot rows, the rows with index in Ω_j, are numerically unstable; delaying provides new admissible pivot rows. We denote the set of columns that were delayed from j to its parent by $\Delta_j = \tilde{\Xi}_j \setminus \Xi_j$.

State-of-the-art sparse factorization codes fall into two categories: left looking [15, 16] and multifrontal [8, 14]. In the indefinite case, the multifrontal approach is much more common and is well documented in literature [5, 8]. The left-looking approach is used in one code that we are aware of, SPOOLES [4], but the formulation of the algorithm is not documented in the literature (Ashcraft and Grime's paper [4] documents the software but not the algorithm). In the next section we address the multifrontal in-core method, and in the next chapter we formulate our left-looking approach.

2.3. **The Multifrontal Indefinite In-Core Factorization.** The multifrontal algorithm traverses the etree and factors the matrix in postorder. To process supernode j, the algorithm creates a so-called *frontal matrix* $F^{(j)}$. This matrix is dense, and its rows and columns corresponds to the columns in Ω_j, to the columns $\Psi_j = \bigcup_{k, j=\pi(k)} \Delta_k$ that were delayed from the subtrees rooted at j's children, and to the columns that columns $\Omega_j \cup \Psi_j$ update. This matrix is initialized to zero, then updated by the nonzeros of A in columns Ω_j. Next, the updates from the subtrees are added to the frontal matrix. All the updates from a child k are packed in an *update matrix* U_{Ξ_k, Ξ_k}. These matrices are added to $F^{(j)}$ in a scatter-add operation that is called *extend-add*. The algorithm now tries to factor the columns in $\Omega_j \cup \Psi_j$. If the diagonal element of a column is large enough compared to the rest of the column, the column is factored and a 1-by-1 block is added to D. If the diagonal element is too small, but a 2-by-2 block consisting of two columns in $\Omega_j \cup \Psi_j$ is large enough compared to the rest of the two columns, they are factored using a 2-by-2 diagonal block. Different codes use different strategies for choosing pivot columns. Once one or two columns are factored, the remaining uneliminated columns in $\Omega_j \cup \Psi_j$ are updated, to allow future pivoting decisions to be made. The factorization of supernode j may fail to factor some of the columns in $\Omega_j \cup \Psi_j$ (essentially those with relatively large elements in rows outside $\Omega_j \cup \Psi_j$). These columns are put into Δ_j and are delayed to $\pi(j)$. After supernode j is factored, the algorithm computes its own update matrix by adding the update from columns in $\tilde{\Omega}_j$ to columns in Ξ_j, which may already contain updates from j's descendants. Note that columns in Δ_j are not updated in this step, because they have already received all the updates columns in $\tilde{\Omega}_j$.

Our codes use a pivot search and a pivot-admissibility test proposed by Ashcraft, Grimes and Lewis [5]. The literature also contains a number of other strategies (including additional strategies in the paper by Ashcraft et al., which is the most recent algorithmic paper on this subject). We used this particular strategy since it is efficient and since it is backed up by extensive research. For further details on this and other strategies, see HIGHAMXXX[10, 11] and the references therein. Figures 2.4 and 2.5 describe the strategy that we use.

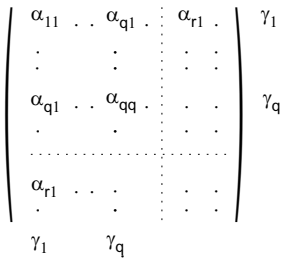

FIGURE 2.4. Notation for pivot entries. The symbols γ_1 and γ_q denote the absolute value of the largest subdiagonal element in columns 1 and q. The elements a_{q1} and a_{r1} are the largest elements in column 1 within and outside $\Omega_j \cup \Psi_j$, respectively.

```
if γ1 =0 then the first column is already factored
else if |a11|≥ α̂γ1 then use a11 as a 1-by-1 pivot
else if |aqq|≥ α̂γq then use aqq as a 1-by-1 pivot
else if max {|aqq|γ1 + |aq1|γq, |aq1|γq + |aq1|γ1} ≤
|a11aqq − a²q1| / α̂ then
     use [ a11   aq1 ]
         [ aq1   aqq ] as a 2-by-2 pivot
else
     no pivot found; repeat search using next column
end if
```

FIGURE 2.5. Our pivot search and admissibility test. This strategy is from [5, Figure 3.3]. The scalar $\hat{\alpha}$ is a parameter that controls the 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 $\hat{\alpha} = 0.001$.

2.4. **Out-of-Core Algorithms.** Many algorithms use data structures while processing and solving a problem. These data structures hold information gathered on the problem, partial solutions and the state of the algorithm, to name a few of their tasks. The most common algorithms, called *In-Core* algorithms, save these data structures in main memory, with little or no use of disk I/O. However, when the data structures are too large to fit in main memory, they must be stored on disks. The in-core algorithm does this inattentionaly, by using the virtual memory, when possible. Accessing data that is stored on disks, or in virtual memory, is very slow compared to access to main memory. Use of virtual memory in such algorithms, may cause their performance to be unacceptable. The in-core algorithms might even fail if the virtual memory does not suffice.

To achieve better performance, an algorithm must handle storage of the data on disks, in a much clever way; the algorithm should access data in large contiguous blocks and be able to reuse data on main memory as much as possible. Such algorithms, that are designed to achieve high performance when using data stored on disks, are called *Out-of-Core* algorithms. Such out-of-core algorithms enable users to efficeintly solve large problems on relatively cheap computers, due to the fact that disk space is signifficantly cheaper than main memory - RAM. [1]Performance of out-of-core algorithms is usually similar to their corresponding in-core algorithms,

[1]In the begginning of 2004, RAM costs about 300$ per GByte, whereas disks cost about 2$ per GByte.

and although they might be poorer in performance, they are more reliable and have a much wider space of solvable problems.

The main difference between an out-of-core algorithm and its corresponding in-core algorithm, is in scheduling the operations. When an algorithm is to be executed out-of-core, the schedule must both satisfy the data-flow constraints of the in-core algorithm, and minimize the I/O, by reordering independant operations. In addition, the data structures stored on disk should be chosen so that I/O can be performed in large blocks, and that when data is read from disk, it is utilized as much as possible before it is evicted from main memory. Good utilization of in-memory data, is the main challenge in designing an out-of-core algorithm. The better the algorithm uses the data stored temporarily in main memory, the more infrequently it needs to access the on-disk data structures, which leads to better performance of the algorithm (I/O wise and otherwise).

Out-of-core algorithms appear in many fields of numerical linear algebra (see [20] for a few examples), including dense and sparse matrix factorizations. In the dense case, the matrix can be partitioned as needed, considering only the I/O needs. Scheduling the factorization of a sparse matrix is a somewhat harder task than in the dense case. The sparsity pattern of the matrix limits our ability to partition it to contiguous blocks, therefor, the design of the out-of-core sparse factorization, is not trivial. For the positive definite case, a few out-of-core Cholesky methods have been proposed. Rothberg and Schreiber [17] describe a family of algorithms which factor sparse positive definite matrices, using multifrontal and left-looking methods. Rotkin and Toledo [18] propose a left-looking approach for the out-of-core Cholesky factorization, with right-looking updates when appropriate. As far as we know, there is no out-of-core method currently proposed for the sparse symmetric indefinite case (excluding our method we are proposing here).

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 [17, 18]. The difference between the multifrontal and the left-looking approaches is in the way that updates to the trailing submatrix are represented. In the multifrontal algorithm, updates are computed when a supernode is factored, and they are accumulated in frontal matrices and proparagated up the tree. In the left-looking approach, updates are not represented explicitly until they are applied to a supernode. The disadvantage of the multifrontal approach is that it often simulteneously represents multiple updates to the same nonzero in L. The representation of these updates, which are not part of the data structure that represents the factor L, uses up memory and causes additional I/O activity.

Unfortunately, left-looking sparse indefinite factorizations are not described in the literature. There is actually one in-core code that uses such a method, Spooles [4], but the algorithm that it uses is not described explicitely anywhere. Therefore, we developed a left-looking sparse indefinite factorization algorithm, which we describe in this section. (The source of Spooles is available, but we have not studied its algorithm from the sourc code.) 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 also traverses the etree and factors the matrix in postorder, but updates to columns are performed in a different way. To process supernode j, the algorithm creates a dense matrix $L^{(j)}$ that will contain all the nonzeros in the supernode (in the columns belonging to the supernode). This matrix is essentially the columns with indices in $\Omega_j \cup \Psi_j$ from the frontal matrix $F^{(j)}$. The columns in Ω_j of this matrix are initialized to zero, and then updated by the nonzeros of A in columns Ω_j. The columns in Ψ_j are simply copied from the

corresponding matrices of the children. Now the algorithm traverses (again) the subtree rooted at j to compute and apply updates to columns Ω_j. At a desendant k, the algorithm determines whether $\tilde{\Xi}_k \cap \Omega_j \neq \emptyset$. If the intersection is not empty, the algorithm computes the update from supernode k to the columns with indices in Ω_j and applies these updates to $L^{(j)}$. These updates are computed into a dense matrix, whose elements are then scatter-added to $L^{(j)}$. This allows us to compute the updates using a dense matrix-matrix multiplication routine. The algorithm then continues recursively to k's children. If, on the other hand, $\tilde{\Xi}_k \cap \Omega_j = \emptyset$, the algorithm returns to k's parent without searching the subtree rooted at k for updates to j; there are none [13].

Note that the algorithm does not test for updates from supernode k to columns in Ψ_j and it does not apply updates to these columns. Since these columns were delayed from one of j's children, say ℓ, all the updates from the subtree rooted at ℓ were already applied to these columns, and the columns in subtrees rooted at other children of j can never update these columns. Therefore, these columns are fully updated.

Now that all the updates from already-factored columns have been applied to $L^{(j)}$, the algorithm tries to factor the columns in $\Omega_j \cup \Psi_j$. This factorization is performed using exactly the same strategy as in the multifrontal algorithm. The set of columns $\tilde{\Omega}_j$ that were successfully factored is added to the factor matrices L and D, and the remaining columns, Δ_j, are delayed to j's parent. By the time these columns are delayed, all the updates from the subtree rooted at j, including from j itself, have been applied to them.

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 themselves. The parent $\pi(j)$ would then read these columns from A and would apply updates to 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, where as the previous strategy treated columns in $\Omega_{\pi(j)}$ differently than those in $\Psi_{\pi(j)}$. However, this strategy performs the same numerical update operations to a delayed column more than once, which increases its computational cost. Due to this increased cost we decided to use the previous strategy.

4. 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 algoithms try to update as many supernodes as possible whenever a factored supernode is read into main memory [9, 18]. Such algorithms maintain in main memory a set of partially-updated but yet-unfactored supernodes, called a *panel*. The panel forms a connected subtree of the elimination tree. These algorithms read from disk the supernodes that must update one of the leaves of the panel, 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 panel that k updates, and is then evicted from memory. Once j is fully updated, it is factored, it updates all the other supernodes in memory, and is evicted. Supernode j is now pruned from the panel, and the factorization continues with another leaf. Such algorithms are not pure left-looking algorithms: they are hybrids of left-looking updates and right-looking updates. They are classified as left-looking because right-looking updates are only applied to 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 adapted this strategy to the factorization of symmetric indefinite matrices. Our algorithm differs from the symmetric-positive definite algorithms of [17, 18] 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.

4.1. **The Left-Looking Out-Of-Core Symmetric Indefinite Algorithm.** Our out-of-core algorithm applies the left-looking panel-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 finding a *panel*, a connected leaf subtree of the residual etree (the etree of the 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 than allocates in-core supernode matrices for the supernodes in the panel and reads the columns of A into them. Then, the algorithm uses the general strategy outlined in the previous paragraph to factor the supernodes of the panel one at a time. Whenever a supernode is factored, it updates its ancestors in the panel 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 more difficult lies in selecting the next panel to be factored. Delaying a column often causes additional fill in 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 panelization procedure cannot ensure that the panel that it selects will fit in main memory.

Our new algorithm addresses this issue in two ways. First, when a column is delayed, we update the symbolic structure of the factorization by moving the delayed row and column to the structure of the parent supernode. This ensures that at the beginning of the next phase, the panelization algorithm uses the most up-to-date information regarding the size of supernodes. They might still expand more after the panel is selected, but at least all the expansion that has already occured is accounted for. Second, the panelization procedure only adds supernodes to the panel as long as the combined predicted size of the panel is at most 75% of the available amount of main memory (after setting aside explicitly memory for other data structures of the algorithm). This helps minimize the risk that supernode expansion will overflow main memory. Normally, if the panel overflows, this causes paging activity and some slowdown in the factorization, but it could also lead to memory-allocation failure. As in [18], we also limit the size of each supernode, to help ensure that an admissible panel can always be found.

The other difficulty lies in delaying columns across panel boundaries. Suppose that columns are delayed from the root supernode j of a panel. The next panel might not include $\pi(j)$, so there is no point in keeping these columns in memory, where they will use up space but will not be used soon. 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 I/O abstraction layer, the so-called *store* [18], we actually write j again to disk without the delayed columns once the delayed columns have been added to $\pi(j)$.)

4.2. **Comparison with Algorithms for Symmetric Positive-Definite Matrices.** Out-of-core factorization algorithms for sparse symmetric positive-definite algorithms can panelize the entire factor prior to the 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 panelization algorithm to decompose the etree into panels before the factorization begins. This has been done by Gilbert and Toledo's [9], by Rotkin and Toledo's [18], and in a more limited way by Rothberg and Schreiber's [17]. As we have explained, this is not possible in the indefinite case, so we adopted a dynamic panelization strategy.

We also note that [9] and [18] actually used a more sophisticated panelization techinque than the one that we described above. A supernode only updates its ancestors in the etree. Therefore, there is no benefit in simulteneously storing in memory supernodes that are not in a descendant-ancestor relationship. Therefore, [9] and [18] allow panels to be larger than the amount of available main memory, and they page supernodes in and out of panels without incurring extra I/O. This reduces the total amount of I/O. Since experiments in [18] have shown that the reduction is not highly significant, however, we have not adopted this strategy in the new indefinite code.

4.3. **Implementation.** Our implementation of the out-of-core indefinite algorithm is an adaption of the sparse Cholesky code of Rotkin and Toledo [18], and in particular, the new code is now part of TAUCS, a suite of publicly-available sparse linear solvers[2]. We use the same hardware-abstraction layer, which is based on a disk-resident data structure called a *store*. The algorithm is implemented in C, with calls to the level-2 and level-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 that we explained above. The code is right-looking and blocked, to exploit the level-3 BLAS and acheive high performance. The blocking strategy is based on the LAPACK code DSYTRF, a blocked Bunch-Kaufman symmetric indefinite factorization code. We could not use the LAPACK code, mainly because our code actually factors the diagonal block of a rectangular matrix, not a square matrix, and the elements in the subdiagonal block affect the admissibity of pivots (in an LU factorization with partial pivoting, such elements can be used as pivots, but here doing so ruin the symmetry). In addition, our pivoting strategy allows pivots with smaller norms than DSYTRF, to reduce the number of delayed columns and the additional fill that follows. In addition to more growth in L, the smaller pivots could also lead to inaccuracies in computing columns of L that correspond to 2-by-2 pivots, since DSYTRF uses the inverse of the diagonal blocks; we use an LU factorization with partial pivoting to reduce that risk.

Our implementation also includes a multiple-right-hand side solve routine. Once the factor has been computed and is stored on disk, the time it takes to solve linear system is determined mostly 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 with the same coefficient matrix during one read-solve process, we can amortize the cost of reading the factor over many linear systems. Even for fairly large numbers of right-hand-side, the solution time is determined mostly by the disk-read time, so the marginal cost of simulteneously solving additional linear system is close to zero.

Many applications can exploit the code's ability to efficiently solve a large number of linear systems with 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 XXX.

Finally, we mention that we have added not a single factorization code to TAUCS, the out-of-core sparse factorization code, but also two in-core sparse symmetric indefinite codes, one multifrontal and the other left-looking.

[2] `http://www.tau.ac.il/~stoledo/taucs/`

Name	Source	SPD?	$\dim(A)$	$\text{nnz}(A)$
s0tau	Bustany	no		
sme1	Ekroth	no		
ldoor	PARASOL	yes		
inline-1	PARASOL	yes		
audikw-1	PARASOL	yes		
femlab1	Ekroth	no		

TABLE 1. Test matrices from real-world applications. Some of the matrices are from the PARASOL test-matrix collection (www.parallab.uib.no/parasol/data.html), some were donated by Ismail Bustany from *Barcelona Design*, and some were donated by Anders Ekroth from *Comsol*. 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.

5. Tests and Results

We now describe 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 set presents the performance of our in-core implementation of the algorithm and of 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, so that we can later use them to assess the performance of the out-of-core algorithm. In this set of experiments we compare the performance of our in-core code to the performance of another recent, well-known, and high-performance code. We also compare the performance of our symmetric-indefinite and Cholesky in-core codes, the performance of left-looking and multifrontal variants, and the performance 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 to the performance of our best in-core code, 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 sparse Cholesky algorithm of Rotkin and Toledo [18], 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 performance results that show the performance benefit of simulteneously solving multiple linear systems with the same coefficient matrix.

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

5.1. **Test Matrices.** We performed our experiments on three sets of symmetric matrices; each set contains both indefinite and positive-definite matrices. The first set, listed in Table 1 consists of matrices that arise in real-world applications. This set is quite small, since there are not many matrices in test-matrix collections that are large enough for evaluating the performance of out-of-core codes. To maximize the utility of these matrices, some of which are positive definite, we generated indefinite matrices from the definite ones by shifting the diagonal.

The second set of matrices consists of synthetic matrices whose graphs are regular three-dimensional meshes. The set consists of positive-definite matrices, which

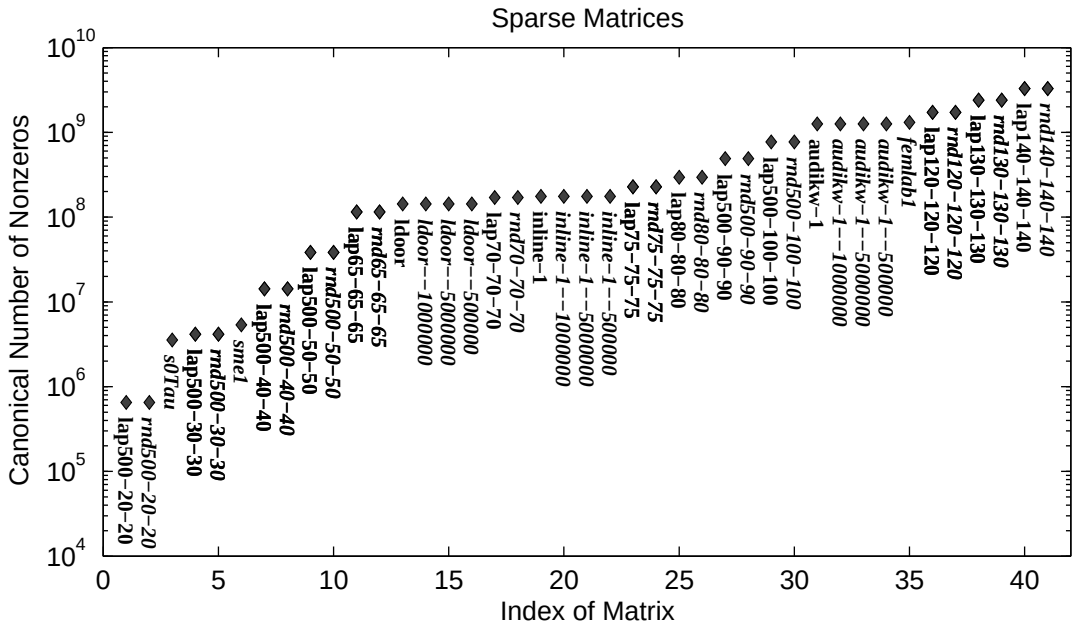

FIGURE 5.1. Sparse test matrices. Names printed in upright (roman) font signify positive-definite matrices, whereas names printed in italics signify indefinite matrices. Positive-definite matrices whose graph is an x-by-y-by-z mesh are named "lapx-y-z" and indefinite meshes are named "*rndx-y-z*". The numbers that follow two dashes are shift values, for matrices whose diagonal was shifted to make them indefinite. The x-axis is simply an index; these matrices are shown using the same x-axis in all subsequent graphs. The y-axis shows the number of nonzeros in the symmetric indefinite lower-triangular factor of each matrix, following a reordering using METIS.

are discretizations of the Laplacian on a 3D mesh using a 7-point stencil, and of indefinite matrices. The indefinite matrices are generated by using the same underlying graph, but assigning symmetric random values (uniform in $[0, 1]$) to the elements of the 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 140-by-140-by-140, and some are longer in one dimension than in the others, such as 500-by-100-by-100. Generally speaking, perfect cubes lead to more fill in the factorization than meshes with wide aspect ratios.

The matrices in the first two sets are listed in Figure 5.1. The matrices are ordered in this figure by the number of nonzeros in their symmetric indefinite factor. The same ordering is used in all the other plots. In other words, we identify matrices by their index in Figure 5.1.

The third set, which we use in only one limited experiment, consists of dense matrices. They are shown in Figure 5.2.

5.2. **Test Environment.** We performed the experiments on an Intel-based workstation. This machine has a 2.4 GHz Pentium 4 processors 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. We compiled our code with the GCC C

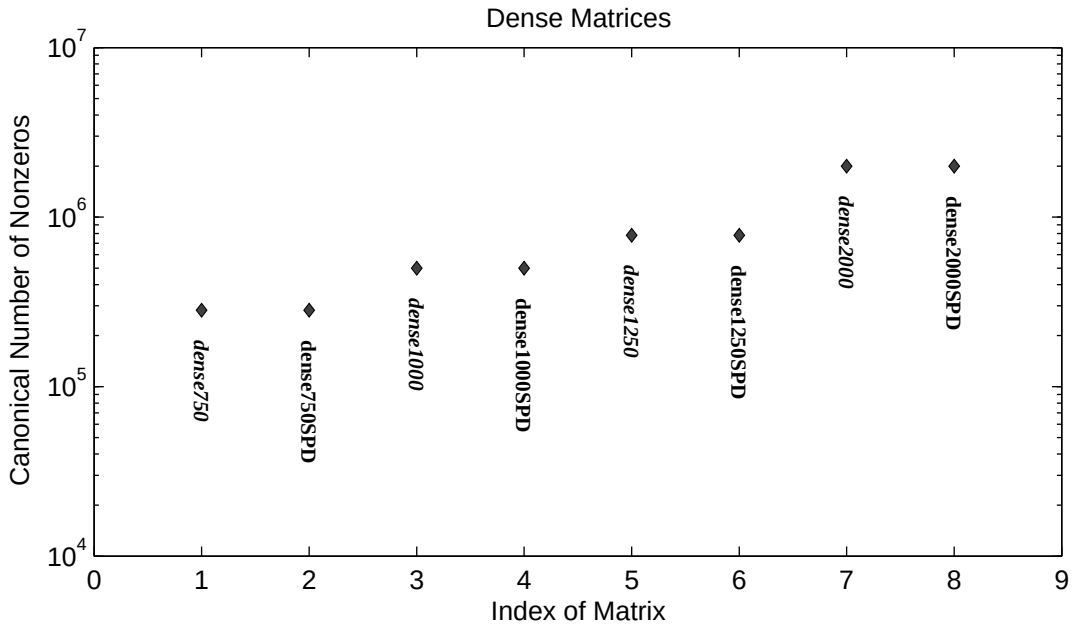

FIGURE 5.2. Dense test matrices. The axes are the same as those of Figure 5.1.

compiler, version 3.3.2, and with the `-O3` compiler option. We used the implementation of the BLAS (Basic Linear Algebra Subroutines) written by Kazushige Goto, version 0.9[3]. This version exploits vector instructions on Pentium 4 processors (these instructions are called SSE2 instructions). This setup allows our code to compute the 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), LL (left-looking), and MF (multifrontal). ????

5.3. **Baseline Tests.** To establish a performance baseline for our experiments, we compare the performance of our code, called TAUCS, to two in-core codes. One is the in-core sparse factorizations in TAUCS, both Cholesky and symmetric indefinite. Our in-core codes can use either a left-looking or a multifrontal algorithm, and we test both. The other code that we use for the baseline tests is MUMPS version 4.3 [2, 1, 3]. MUMPS is a parallel and sequential in-core multifrontal factorization code for symmetric and unsymmetric matrices. We used METIS[4] [12] version 4.0 to symmetrically reorder the rows and columns of the all the matrices prior to factoring them. We tested the sequential version of MUMPS, with options that instruct it to use METIS to preorder the matrix and tell MUMPS that the input matrix is symmetric, and positive definite when appropriate. We used the default values for all the other run-time options.

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

[3] http://www.cs.utexas.edu/users/flame/goto/

[4] `http://www-users.cs.umn.edu/~karypis/metis/`

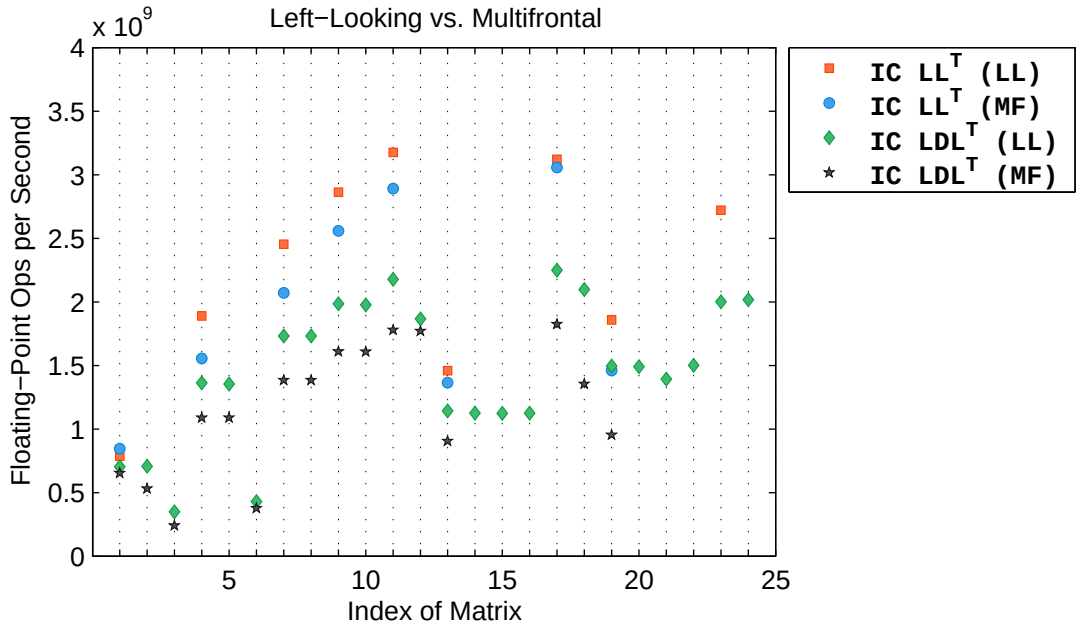

FIGURE 5.3. The performance of the in-core factorization codes in TAUCS. The figure only shows the performance on the subset of matrices that could be factored in core. The computational rates, in floating-point operations per second, are *canonical rates*, as explained in the text. In particular, a higher rate indicates faster completion time, not a higher operation count.

The results of the baseline tests are shown in Figures 5.3, 5.4, and 5.5. The results, both here and later, display performance in terms of *canonical floating-point-operations per second*. This metric is the ratio of the number of floating-point operations in the out-of-core symmetric-indefinite factorization in TAUCS to numeric factorization time in seconds. Thus, if one code achieves a canonical rate of 2×10^9 and another code achieves a rate of 4×10^9, then the second code ran in exactly half the time, independently of how dense a factor each code produced.

Figure 5.3 compares the performance of the left-looking and multifrontal factorizations in TAUCS. The plot only shows a subset of the matrices, those small enough to be factored in core. The results show that the 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.

Figure 5.4 shows the performance of TAUCS relative to that of MUMPS. MUMPS ran out of memory on many of the matrices that TAUCS was able to factor in core. MUMPS also reported that s0tau is singular and halted; this is not a defect, but simply reflect different built-in thresholds in TAUCS and MUMPS. The data shows that on this setup, TAUCS is consistently faster. This result does not imply that one code is superior to the other in general, since our comparison is quite limited. This result merely indicates, for the purpose of our experimental evaluation, that the performance of the in-core routines of TAUCS are comparable to the performance of MUMPS.

Figure 5.5 shows that the routine that 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

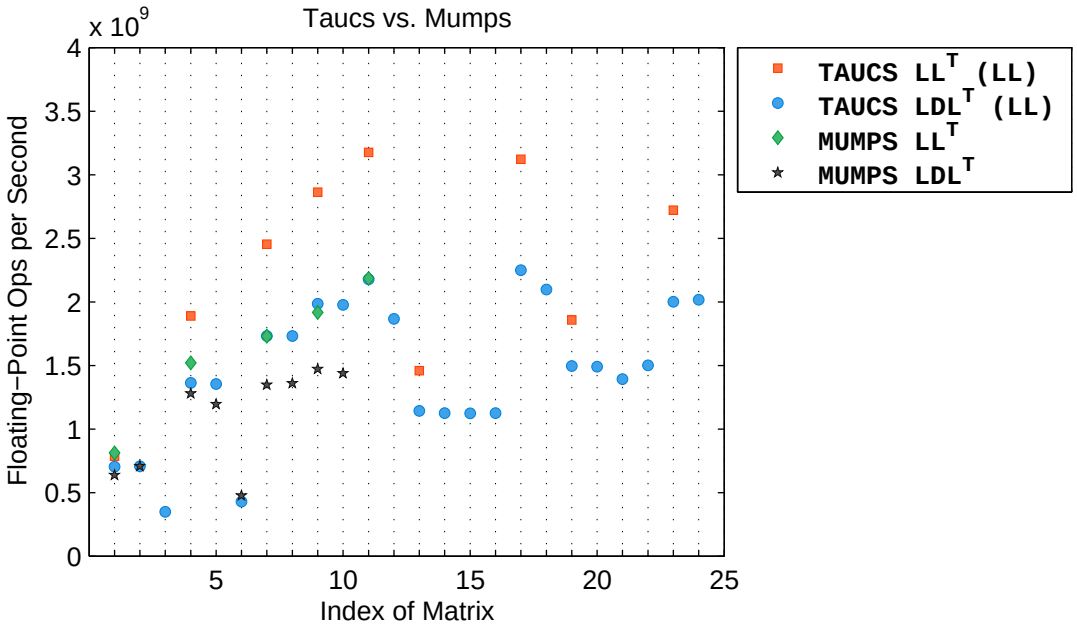

FIGURE 5.4. The performance of TAUCS vs that of MUMPS, again in canonical rates.

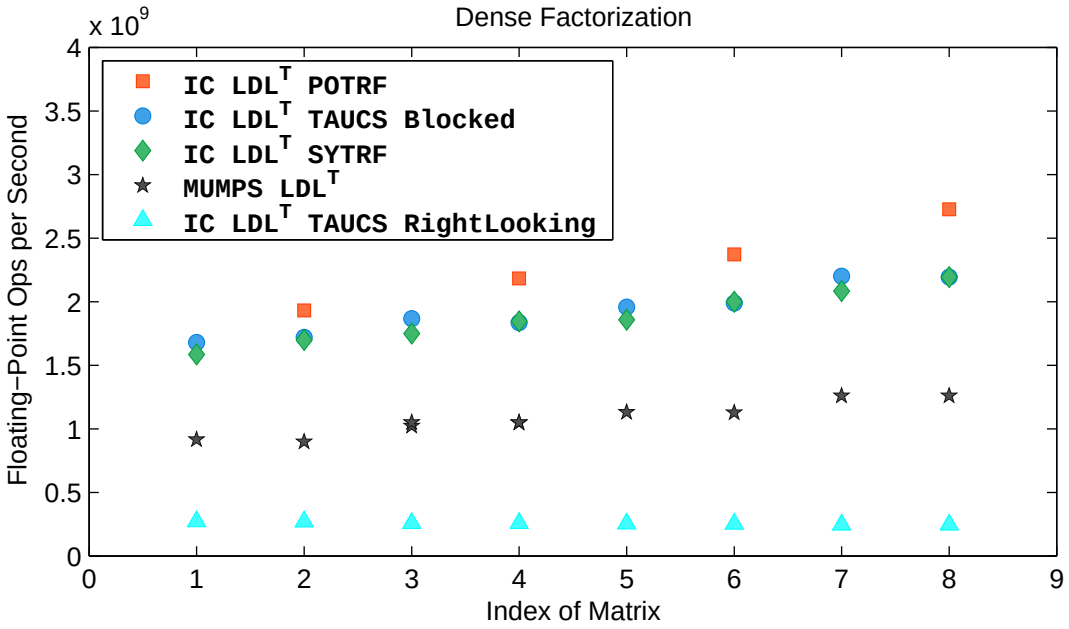

FIGURE 5.5. A comparison of dense symmetric-indefinite and Cholesky factorization kernels. The figure shows the performance of TAUCS and MUMPS on dense matrices, and with 4 different kernels in TAUCS.

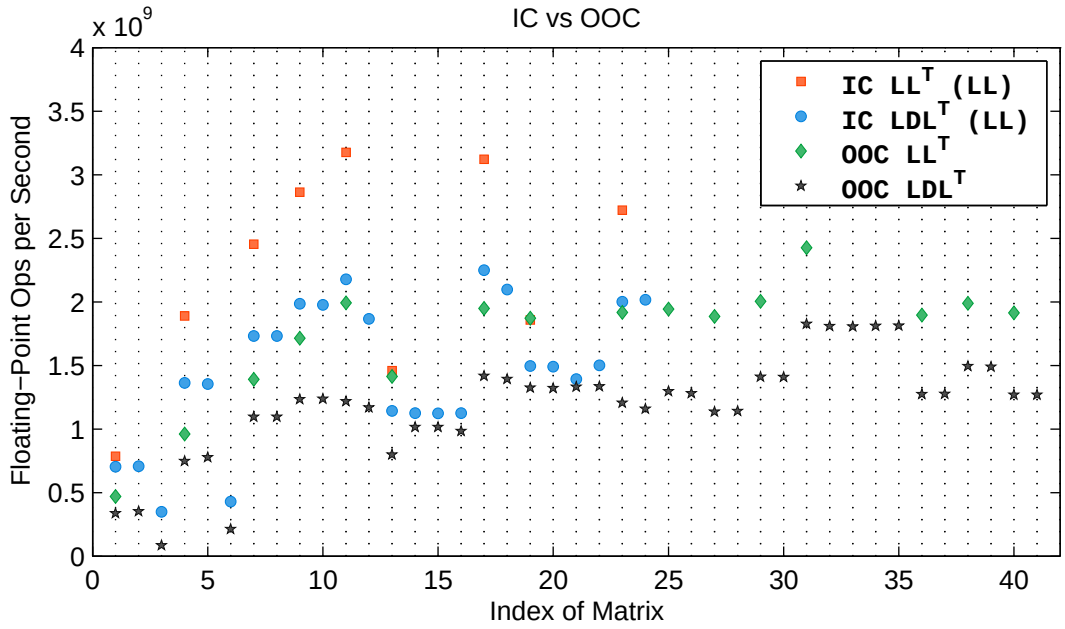

FIGURE 5.6. The performance of the new out-of-core symmetric indefinite factorization code. For comparison, the graph also shows the performance of three other TAUCS codes: the in-core symmetric indefinite factorization and the in- and out-of-core Cholesky factorizations.

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 that it is faster than MUMPS'.

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

5.4. **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.

Figure 5.6 presents the performance of the new out-of-core symmetric-indefinite factorization algorithm. As expected, the performance of the code is always lower than the performance of the in-core symmetric indefinite code and than the performance of the Cholesky codes, when the other codes do not break down. However, the performance difference between the in-core and the out-of-core symmetric indefinite codes is usually small, which suggests that the performance penalty paid for the extra robustness is small. The performance difference between the symmetric-indefinite and the 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 between 1×10^9 and 2×10^9 floating-point operations per second.

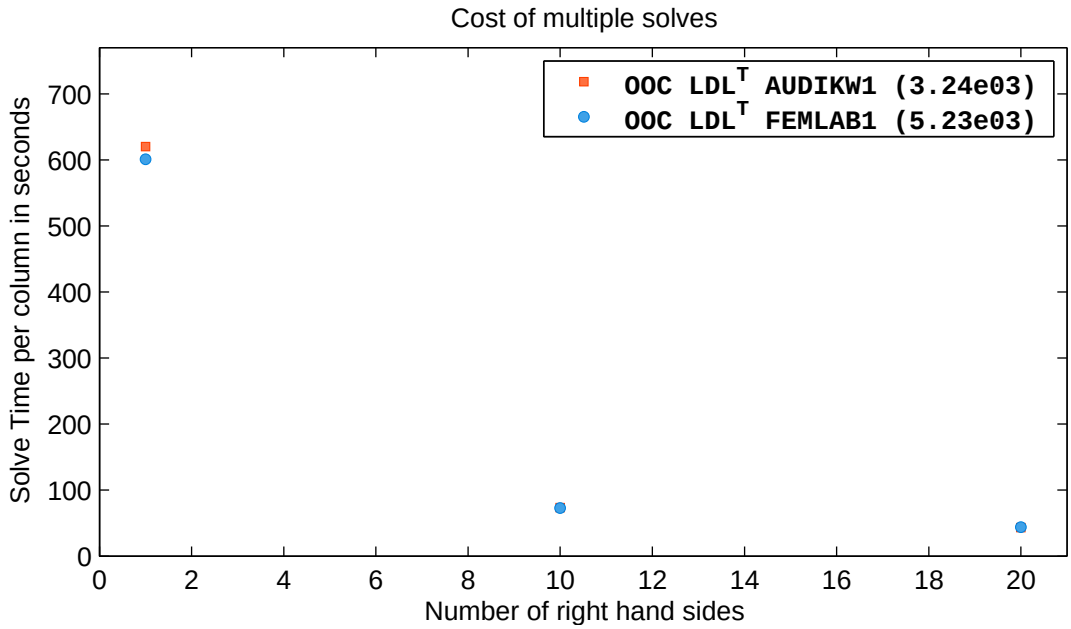

FIGURE 5.7. The performance of the out-of-core solve phase as a function of the number of right-hand-sides. The numerical factorization time is stated in parentethes.

Figure 5.7 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 to read the factor from disk. However, the disk-read time can be amortized over multiple right-hand sides. When multiple linear systems are solved simultaneously, the solve-time per system drops dramatically.

Figure 5.8 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 and when its inertia changes.

Figure 5.9 shows that the out-of-core code slows down when it must run with limited main memory. To conduct this experiment, we configured the test machine so that the operating system is only aware of 512 MB of main memory. In the runs that we conducted with 512 MB, we instructed the factorization code to use only 384 MB of memory, 75% of the available memory, the same fraction as in experiments with 2 GB of memory. On small matrices the slowdown is not significant, but on large matrices it can reach a factor of 2.6. Furthermore, the largest matrices, lap140-140-140 and rnd140-140-140, could not be factored at all with only 512 MB of memory. Still, this experiment shows that even on a machine with a relatively small amount of memory, 512 MB, our code can factor very large matrices. But a larger memory helps, both in terms of the ability to factor very large matrices, and in terms of running times.

6. Discussion and Conclusion

Thanks toTo the best of our knowledge, this paper presents the first out-of-core sparse symmetric indefinite factorization algorithm. Our implementation of the algorithm is reliable and performs well. 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 Choleksy codes.

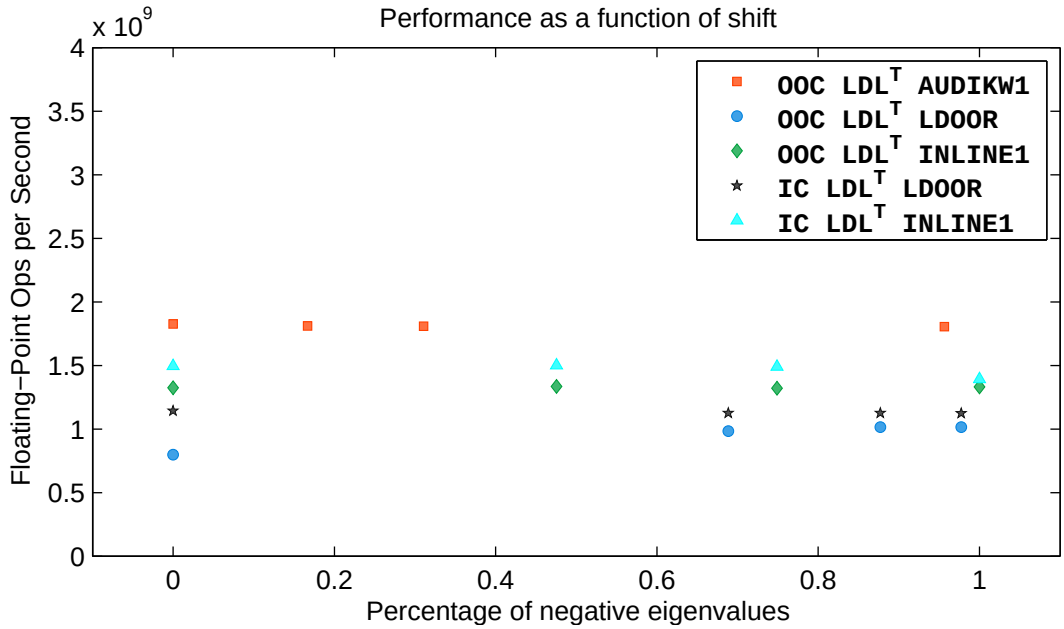

FIGURE 5.8. The performance of the symmetric indefinite codes as a function of the percentage of negative eigenvalues in the matrix. The figure shows the performance of the code on shifted versions of three large positive-definite matrices.

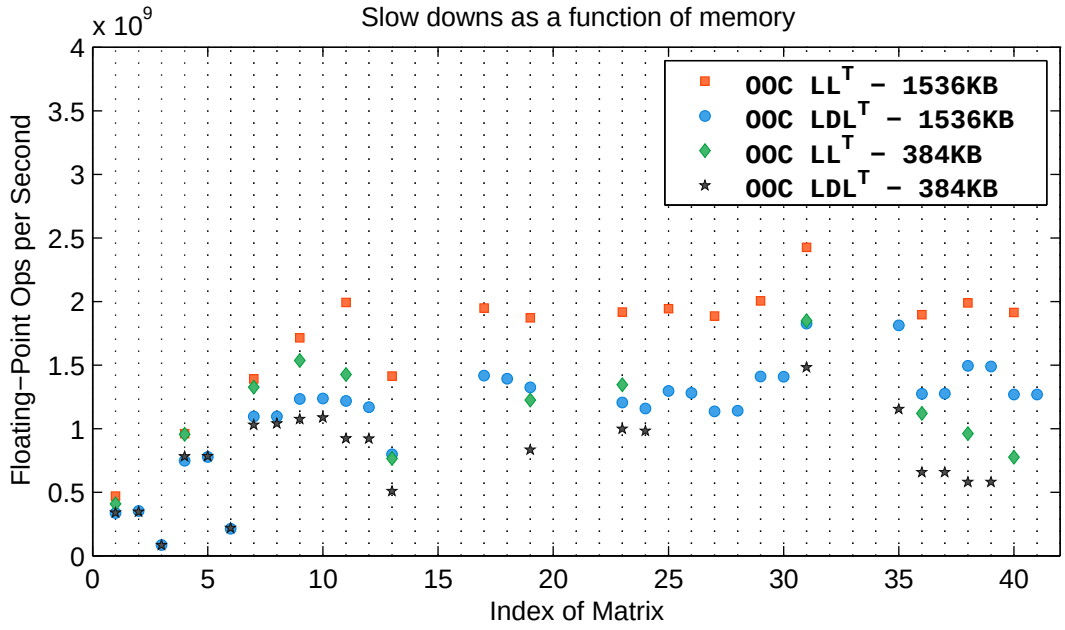

FIGURE 5.9. The performance of the out-of-core as a function of main memory size. The memory sizes shown in the legend are the target memory usage given to the code; the operating system itself had access to 2048 MB in one set of experiments and 512 MB in the other set.

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 simulteneously solve for multiple right-hand sides reduces even further the per-system cost of the solve phase.

Acknowledgement. Thanks to Dror Irony for assisting in the implementation of the in-core code, and for writing the inertia-computation subroutine. Thanks to Anders Ekroth and Ismail Bustany for sending us test matrices. Thanks to Didi Bar-David for configuring the disks of the test machine. This research was supported in part by an IBM Faculty Partnership Award, by grant 572/00 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and by grant 2002261 from the United-States-Israel Binational Science Foundation.

References

[1] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23:15–41, 2001.

[2] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering*, 184, 2000.

[3] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. MUltifrontal Massively Parallel Solver (MUMPS version 4.3), user's guide. Available online from `http://www.enseeiht.fr/lima/apo/MUMPS/doc.html`, July 2003.

[4] Cleve Ashcraft and Roger Grimes. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San-Antonio, Texas, 1999. 10 pages on CD-ROM.

[5] Cleve Ashcraft, Roger G. Grimes, and John G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM Journal on Matrix Analysis and Applications*, 20:513–561, 1998.

[6] J. R. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric indefinite linear systems. *Math. Comput.*, 31:163–179.

[7] James R. Bunch, Linda Kaufman, and Beresford N. Parlett. Decomposition of a symmetric matrix. *Numer. Math.*, 27:95–109, 1976.

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

[9] John R. Gilbert and Sivan Toledo. High-performance out-of-core sparse LU factorization. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San-Antonio, Texas, 1999. 10 pages on CDROM.

[10] Nicholas J. Higham. Stability of the diagonal pivoting method with partial pivoting. Technical Report 1, Manchester, England, 1997.

[11] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[12] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998.

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

[14] Joseph W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, 1992.

[15] Esmond G. Ng and Barry W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, 14(5):1034–1056, 1993.

[16] Edward Rothberg and Anoop Gupta. Efficient sparse matrix factorization on high-performance workstations—exploiting the memory hierarchy. *ACM Transactions on Mathematical Software*, 17(3):313–334, 1991.

[17] Edward Rothberg and Robert Schreiber. Efficient methods for out-of-core sparse cholesky factorization. *SIAM Journal on Scientific Computing*, 21:129–144, 1999.

[18] Vladimir Rotkin and Sivan Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. To appear in *ACM Transactions on Mathematical Software*, February 2003.

[19] Robert Schreiber. A new implementation of sparse gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982.

[20] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In James M. Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 161–179. American Mathematical Society, 1999.

School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel.
stoledo@tau.ac.il, omerm@tau.ac.il
URL: `http://www.tau.ac.il/~stoledo`