The Design and Implementation Of A New Out-of-Core Sparse Cholesky Factorization Method

VLADIMIR ROTKIN and SIVAN TOLEDO
Tel-Aviv University

We describe a new out-of-core sparse Cholesky factorization method. The new method uses the elimination tree to partition the matrix, an advanced subtree-scheduling algorithm, and both right-looking and left-looking updates. The implementation of the new method is efficient and robust. On a 2 GHz personal computer with 768 MB of main memory, the code can easily factor matrices with factors of up to 48 GB, usually at rates above 1 Gflop/s. For example, the code can factor AUDIKW, currenly the largest matrix in any matrix collection (factor size over 10 GB), in a little over an hour, and can factor a matrix whose graph is a 140-by-140-by-140 mesh in about 12 hours (factor size around 27 GB).

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*; *efficiency*; E.5 [**Files**]—*Organization/structure*

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: out-of-core

1. INTRODUCTION

Out-of-core sparse direct linear solvers remain useful in spite of improvments in iterative solvers and in spite of ever-growing main memories. Direct sparse solvers are very reliable: they usually fail only when they run out of memory; they rarely fail due to numerical reasons. Although on large linear systems a direct solver is usually slower than a properly chosen and tuned preconditioned iterative solver, a direct solver may be significantly faster than an iterative solver that does not match well the linear system to be solved. In addition, many

This research was supported in part by an IBM Faculty Partnership Award, by an IBM Equinox Equipment award, and by grants 572/00 and 9060/99 from the Israel Science Foundation (Founded by the Israel Academy of Science and Humanities).
Authors' address: School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel; email: stoledo@tau.ac.il; URL: http://www.tau.ac.il/˜stoledo.

iterative solvers suffer from a number of numerical problems, ranging from breakdowns in the construction of preconditioners and breakdowns during the iterations to slow convergence or even nonconvergence. Therefore, software that needs to solve sparse linear systems often includes a direct solver, sometimes as the only solver and sometimes in addition to iterative solvers. The inclusion of a direct solver assures the developers that the software will be able to solve virtually any linear system given sufficient time and memory.

The large main memories of computers today allow direct solvers to solve many problems in main memory. Most engineering and technical PCs today are configured with several hundred megabytes, and many of them can be configured with as much as 1.5–2 gigabytes of main memory. Server-class machines and supercomputers can utilize tens or hundreds of gigabytes of main memory. However, the scaling behavior of sparse direct solvers is such that they often run out of memory even on computers with large memories, especially when the linear systems arise from discretizations of 3D structures. Even when a server or supercomputer with tens or hundreds gigabytes is available, it is often impractical to use most of the machine's memory for the time required to solve the linear system, since such machines are often expensive shared resources.

Out-of-core sparse direct solvers can solve large linear systems on machines with limited main memory by storing the factors of the coefficient matrix on disks. Disks are cheap (approximately 50–100 times cheaper than main memory per megabyte), so it is practical and cost-effective to equip a machine with tens or hundreds of gigabytes of disk space. As we show in this paper and as others have shown [Bjørstad 1987; George and Rashwan 1985; George et al. 1981; Gilbert and Toledo 1999; Rothberg and Schreiber 1996; Rothberg and Schreiber 1999], access time to data on slow disks does not significantly slow down properly designed sparse factorization algorithms. The ability to store their data structures on disks makes out-of-core sparse direct solvers very reliable, since like in-core direct solvers they usually do not suffer from numerical problems, and unlike in-core solvers, they are less likely to run out of memory, since most of their data structures are stored on disks. Vendors of software that includes sparse linear solvers, such as ANSYS, often include a sparse out-of-core solver in their products due to their reliability [Poole et al. 2001].

The main challenge in designing out-of-core solvers of any kind lies in I/O (input-output) minimization. Disks are slow, and it is essential to perform as little I/O as possible in order to achieve near-in-core performance levels. In addition, in order to achieve high performance it is essential to fully exploit the sparsity of the coefficient matrix and the capabilities of the computer's architecture (i.e., to exploit caches and the processor's functional units).

This paper describes a new out-of-core sparse Cholesky factorization method and its implementation. The method is based on a supernodal left-looking factorization [Ng and Peyton 1993; Rothberg and Gupta 1991], which allows the method to fully exploit the sparsity of the matrix and the capabilities of the hardware. A new technique for minimizing I/O represents the main algorithmic novelty in our new method. This technique, which was first introduced in a pivoting-LU factorization algorithm by Gilbert and Toledo [1999], minimizes I/O by exploiting the elimination tree, a combinatorial structure that compactly represents dependences in the factorization process. In this paper we apply this technique to a supernodal Cholesky factorization (the Gilbert-Toledo LU factorization code is not supernodal, so it is relatively slow). Our code is robust in that, to our knowledge, it copes with smaller main memories better than all previous solvers. The code is freely available as part of the TAUCS subroutine library.[1]

The paper is organized as follows. The next section describes the principles of sparse supernodal Cholesky factorizations, including both left-looking and multifrontal. Section 3 describes out-of-core sparse Cholesky one algorithms, including our new one. Section 4 describes the on-disk data structures that our code uses. Section 5 describes the overall structure of the code, and Section 6 describes the performance of our algorithm using extensive experimental results. Finally in Section 7 we give our conclusions from this research.

2. IN-CORE SPARSE CHOLESKY FACTORIZATION

This section describes the basic structure of the sparse Cholesky factorization; for additional information, see the monographs of Duff, Erisman, and Reid [Duff et al. 1986] and of George and Liu [1981], as well as the papers cited later in this section. The factorization computes the factor L of a sparse symmetric positive-definite matrix $A = LL^T$. All state-of-the-art sparse Cholesky algorithms, including ours, represent explicitly few zero elements in L or even none. Many algorithms do represent explicitly a small number of the zero elements when doing so is likely to reduce indexing overhead. The factor L is typically sparse, but not as much as A. One normally preorders the rows and columns of A symmetrically to reduce fill (elements that are zero in A but nonzero in L).

A combinatorial structure called the *elimination tree of A* plays a key role in virtually all state-of-the-art Cholesky factorization methods, including ours [Schreiber 1982]. The elimination tree (etree) is a rooted forest (tree unless A has a nontrivial block-diagonal decomposition) with n vertices, where n is the dimension of A. The parent $p(j)$ of vertex j in the etree is defined to be $p(j) = \min_{i>j}\{L_{ij} \neq 0\}$. The elimination tree compactly represents dependences in the factorization process and has several uses in sparse factorization methods [Liu 1990]. The elimination tree can be computed from A in time that is essentially linear in the number of nonzeros in A.

Virtually all the state-of-the-art sparse Cholesky algorithms use a *supernodal decomposition* of the factor L, illustrated in Figure 1 [Duff and Reid 1983; Ng and Peyton 1993; Rothberg and Gupta 1991]. The factor is decomposed into dense diagonal blocks and into the corresponding subdiagonal blocks, such that rows in the subdiagonal blocks are either entirely zero or almost completely dense. The set of columns associated with each such block is called a *supernode*. In a *strict* supernodal decomposition, rows in subdiagonal blocks must be either zero or dense; the subdiagonal blocks in a strict decomposition can be packed into completely dense two-dimensional arrays. In an *amalgamated* [Duff and Reid 1983] (sometimes called *relaxed* [Ashcraft and Grimes 1989]) decomposition, a small fraction of zeros is allowed in the nonzero rows of a subdiagonal block. Relaxed decompositions generally have larger blocks than strict ones,

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

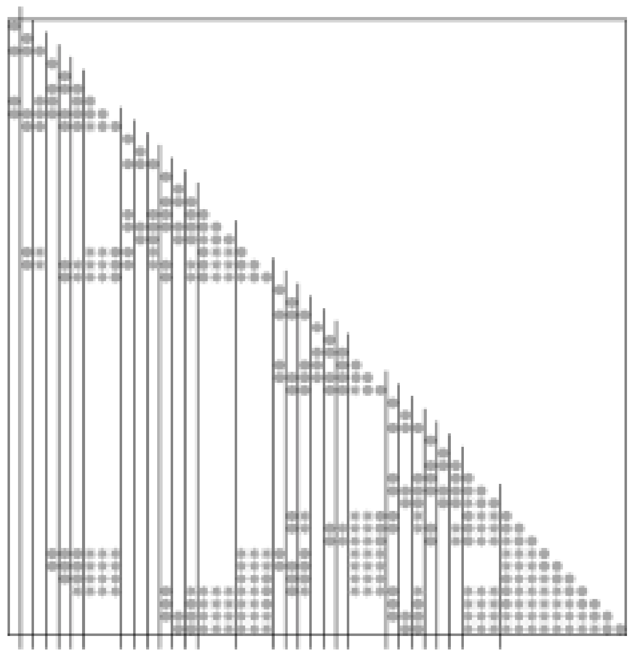

Fig. 1. A fundamental supernodal decomposition of the factor of a matrix corresponding to a 7-by-7 grid problem, ordered with nested dissection. The circles correspond to elements that are nonzero in the coefficient matrix and the stars represent fill elements.

but the blocks contain some explicit zeros. Since larger blocks reduce indexing overheads and provide more opportunities for data reuse in caches, some amount of relaxation typically improves performance. Given A and its etree, a linear time algorithm can compute a useful strict supernodal decomposition of L called the *fundamental* supernodal decomposition [Liu et al. 1993]. This decomposition typically has fairly large supernodes and is widely used in practice. Finding a relaxed supernodal decomposition with larger supernodes is somewhat more expensive, but is usually still inexpensive relative to the numerical factorization itself [Ashcraft and Grimes 1989; Duff and Reid 1983].

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 $\sigma - 1$ using a postorder traversal, where σ is the number of supernodes. We associate with supernode j, the ordered set Ω_j of columns in the supernode, and the unordered set Ξ_j of row indices in the subdiagonal block. 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 1, are $\Omega_{29} = (37, 38, 39)$ and $\Xi_{29} = \{40, 41, 42, 46, 47, 48, 49\}$.

State-of-the-art sparse factorization codes fall into two categories: left looking [Ng and Peyton 1993; Rothberg and Gupta 1991] and multifrontal [Duff and Reid 1983; Liu 1992]. The next paragraph explains how left-looking methods,

```
left-looking-factor($A$, $S$)
  for each root $j \in S$
    call recursive-ll-factor($j$, $A$)
  end for
end

recursive-ll-factor($j$, $A$)
  for each child $k$ of $j$
    call recursive-ll-factor($k$, $A$)
  end for
  set $V_{\Omega_j \cup \Xi_j, \Omega_j} \leftarrow A_{\Omega_j \cup \Xi_j, \Omega_j}$
  for each child $k$ of $j$
    call recursive-ll-update($j$, $V$, $k$, $L$, $S$)
  end for
  factor $V_{\Omega_j, \Omega_j} = L_{\Omega_j, \Omega_j} L^T_{\Omega_j, \Omega_j}$
  solve $L_{\Xi_j, \Omega_j} L^T_{\Omega_j, \Omega_j} = V_{\Xi_j, \Omega_j}$ for $L_{\Xi_j, \Omega_j}$
  append $L_{\Omega_j \cup \Xi_j, \Omega_j}$ to $L$
end

recursive-ll-update($j$, $V$, $k$, $L$, $S$)
  if $\Xi_k \cap \Omega_j = \emptyset$ return
  for each child $k'$ of $k$
    call recursive-ll-update($j$, $V$, $k'$, $L$, $S$)
  end for
  $V_{(\Omega_j \cup \Xi_j) \cap \Xi_k, \Omega_j \cap \Xi_k} \leftarrow$
      $V_{(\Omega_j \cup \Xi_j) \cap \Xi_k, \Omega_j \cap \Xi_k} - L_{(\Omega_j \cup \Xi_j) \cap \Xi_k, \Omega_k} L^T_{\Omega_j \cap \Xi_k, \Omega'_k}$
end
```

Fig. 2. Supernodal left-looking sparse Cholesky. These three subroutines compute the Cholesky factor L given a matrix A and its supernodal etree S.

including our out-of-core method, work. Since some out-of-core factorizations use multifrontal methods, we also explain how multifrontal methods work, in order to compare our left-looking method to multifrontal methods.

Given a matrix A and its supernodal etree S, the code shown in Figure 2 computes the Cholesky factor of L using a left-looking method. The code factors supernodes in postorder. To factor supernode j, the code copies the jth supernode block of A into working storage V. The code then updates V using blocks of L in the subtree rooted at j. Once all the updates have been applied, the code factors V to form the jth supernode block of L. The factorization of V is performed in two steps: we first factor its dense diagonal block and then solve multiple linear systems with the same dense triangular coefficient matrix $L^T_{\Omega_j,\Omega_j}$. Two sparsity issues arise in the subroutine `recursive-ll-update`: testing the condition $\Xi_k \cap \Omega_j = \emptyset$ and updating V. The condition $\Xi_k \cap \Omega_j = \emptyset$ can be tested in time $\Theta(\xi_k)$ by looking up the elements of Ξ_k in a bitmap representation of Ω_j, but the set of supernodes that update supernode j can also be computed directly from S and A [Liu 1990]. The update to V is a sparse-sparse update, which is typically implemented by representing $V_{(\Omega_j \cup \Xi_j) \cap \Xi_k, \Omega_j \cap \Xi_k}$ in an $(\omega_j + \xi_j)$-by-$\zeta_{j,k}$ array, where $\zeta_{j,k} \geq |\Omega_j \cap \Xi_k|$. An auxiliary integer array of size n maps matrix indices into rows in the array that stores V. V can be copied into the array inside the jk update, in which case $\zeta_{j,k} = |\Omega_j \cap \Xi_k|$, or once

```
multifrontal-factor($A$, $S$)
  for each root $j \in S$
    call recursive-mf-factor($j$, $A$)
  end for
end

recursive-mf-factor($j$, $A$)
  set $F^{(j)}_{\Omega_j \cup \Xi_j, \Omega_j \cup \Xi_j} \leftarrow 0$
  add $F^{(j)}_{\Omega_j \cup \Xi_j, \Omega_j} \leftarrow F^{(j)}_{\Omega_j \cup \Xi_j, \Omega_j} + A_{\Omega_j \cup \Xi_j, \Omega_j}$
  for each child $k$ of $j$
    $U^{(k)}_{\Xi_k, \Xi_k} \leftarrow$recursive-ll-factor($k$, $A$)
    extend-add $F^{(j)}_{\Omega_j \cup \Xi_j, \Omega_j \cup \Xi_j} \leftarrow F^{(j)}_{\Omega_j \cup \Xi_j, \Omega_j \cup \Xi_j} - U^{(k)}_{\Xi_k, \Xi_k}$
    discard $U^{(k)}_{\Xi_k, \Xi_k}$
  end for
  factor $F^{(j)}_{\Omega_j, \Omega_j} = L_{\Omega_j, \Omega_j} L^T_{\Omega_j, \Omega_j}$
  solve $L_{\Xi_j, \Omega_j} L^T_{\Omega_j, \Omega_j} = F^{(j)}_{\Xi_j, \Omega_j}$ for $L_{\Xi_j, \Omega_j}$
  append $L_{\Omega_j \cup \Xi_j, \Omega_j}$ to $L$
  update $U^{(j)}_{\Xi_j, \Xi_j} \leftarrow F^{(j)}_{\Xi_j, \Xi_j} - L_{\Xi_j, \Omega_j} L^T_{\Xi_j, \Omega_j}$
  return $U^{(j)}_{\Xi_j, \Xi_j}$
end
```

Fig. 3. Supernodal multifrontal sparse Cholesky. These subroutines compute the Cholesky factor L given a matrix A and its supernodal etree S.

before we begin updating supernode j, in which case we use $\zeta_{j,k} = \omega_j$. The two options lead to different storage-computation tradeoffs that we discuss in the next section. To allow finding the intersection $(\Omega_j \cup \Xi_j) \cap \Xi_k$ quickly, the sets Ξ_k are kept sorted in etree postorder. This allows us to exploit the identity $(\Omega_j \cup \Xi_j) \cap \Xi_k = \Xi_k \cap \{k' : k' \text{ is a descendant of } k\}$.

Figure 3 describes how multifrontal methods work. These methods factor supernodes in etree postorder. To factor supernode j, the algorithm constructs a symmetric $(\omega_j + \xi_j)$-by-$(\omega_j + \xi_j)$ full matrix $F^{(j)}$ called a *frontal matrix*. This matrix represents the submatrix of L with rows and columns in $\Omega_j \cup \Xi_j$. We first add the jth supernode of A to the first ω_j columns of $F^{(j)}$. We then factor all the children of j. The factorization of child k returns a ξ_j-by-ξ_j full matrix $U^{(k)}$ called an *update matrix*. The update matrix contains all the updates to the remaining equations from the elimination of columns in supernode j and its descendants. These updates are then subtracted from $F^{(j)}$ in a process called *extend-add*. The extend-add operation subtracts one compressed sparse matrix from another containing a superset of the indices; the condition $\Xi_k \subseteq \Omega_j \cup \Xi_j$ always holds. Once all the children have been factored and their updates incorporated into the frontal matrix, the first ω_j columns of F_j are factored to form columns Ω_j of L. A rank-ω_j update to the last ξ_j columns of $F^{(j)}$ forms $U^{(j)}$, which the subroutine returns. If j is a root, then $U^{(j)}$ is empty ($\xi_j = 0$). The frontal matrices in a sequential multifrontal factorization are allocated on the calling stack, or on the heap using pointers on the calling stack. The frontal matrices that are allocated at any given time reside on one path from a root of the etree to some of its descendants. By delaying the allocation of the frontal matrix until after we factor the first child of j and by cleverly restructuring the etree, one can reduce the maximum size of the frontal-matrices stack [Liu 1986].

3. OUT-OF-CORE SPARSE FACTORIZATION METHODS

This section describes our new out-of-core sparse Cholesky factorization algorithm. Since our new algorithm builds on previous ones, and in particular on the algorithms proposed by Rothberg and Schreiber [1999], we also describe previously-proposed algorithms. We describe both our new algorithm and most of the algorithms in Rothberg and Schreiber [1999] using a unified framework, which allows us to describe multiple algorithms succinctly and to easily explain the differences between them.

The algorithms described in this section are robust in the sense that they can work with very little main memory, but most need at least $O(n)$ words of main memory, where n is the order of A. Arrays of size n allow the algorithms to perform sparse-sparse update operations efficiently. We therefore assume that the size M of main memory satisfies $M > cn$ for some integer $c \geq 1$. Our algorithm, for example, requires at least $20n$ words of memory.

The literature also contains proposals for non-robust methods that may fail when L is too large compared to main memory, even for relatively small n. For example, Liu [1987] proposes a method that works as long as for all $j = 1, \ldots, n$, all the nonzeros in $L_{j:n,1:j}$ fit simultaneously in main memory. Such methods are highly unlikely to successfully factor matrices whose graphs are large 3D meshes, since their factors usually have a relatively large and almost dense block at the lower-right corner. Robust methods can factor such matrices with little difficulty, as shown in Section 6, so we consider methods such as Liu's obsolete and do not discuss them further.

3.1 A Naive Out-of-Core Left-Looking Factorization

A simple way to use the left-looking approach in an out-of-core factorization is to keep at most two supernodes in memory at any given time, $V_{\Omega_j \cup \Xi_j, \Omega_j}$ and $L_{(\Omega_j \cup \Xi_j) \cap \Xi_k, \Omega_k}$. It is easy to ensure that each supernode occupies at most half the main memory by breaking large supernodes into chains of smaller ones. In this approach, whenever we need a block of L in `recursive-ll-update`, we read it from disk. When we complete the update and factorization of another block, we write it to disk.

While this method is robust in that it does not break down even with very small main memories, it does suffer from a serious defect. Each supernode k in L is read from disk every time we factor a supernode j that k must update. As we show later, it is possible to update many supernodes when we read a factored supernode from disk. Another way to view this defect is to observe that if many supernodes are small, then during most of the running time, this method only utilizes a small fraction of main memory.

We are not aware of any algorithm that uses such a naive approach.

3.2 A Robust Out-of-Core Multifrontal Factorization

The multifrontal approach is more difficult to adapt to out-of-core factorizations. The problem is that frontal matrices may be too large to fit in main memory.[2] We cannot reduce their size arbitrarily, since $F^{(j)}$ is $(\omega_j + \xi_j)$-by-$(\omega_j + \xi_j)$; we can control the ω_j's by splitting large supernodes, as we have done in the out-of-core left-looking method, but we cannot control the ξ_j's at all. As long as all the frontal matrices fit in main memory, we can use an out-of-core multifrontal method that simply keeps the stack of frontal matrices on disk. We can improve upon this simple method by keeping the most recently used frontal matrices in main memory, hence reducing I/O. But when frontal matrices grow too large to fit in main memory, these simple adaptations fail.

Rothberg and Schreiber [1999] proposed a robust out-of-core multifrontal method. Their method splits large frontal matrices into blocks of columns that do fit in main memory. After we factor all the children of a supernode, we form and factor the frontal matrix of the supernode block-by-block. All the updates from the update matrices $U^{(k)}$ of children of supernode j are applied the current block. Then, updates from previous blocks of the frontal matrix are applied. The block is then factored and written to disk. It seems likely that a two-dimensional decomposition of the frontal matrix into blocks would perform less I/O than Rothberg and Schreiber's block-column approach, as is the case for dense matrices [Toledo 1997; Toledo and Gustavson 1996], but we are not aware of such implementations.

The main defect in Rothberg and Schreiber's robust multifrontal method is that on thin long supernodes (small ω_j and large ξ_j), processing a supernode requires reading and writing $\Theta((\omega_j + \xi_j)^2)$ words from disk, but only performs $\Theta(\omega_j(\omega_j + \xi_j)^2)$ floating-point operations. The poor computation-to-I/O ratio creates an I/O bottleneck that left-looking methods do not suffer from. Rothberg and Schreiber observed this phenomenon in experiments involving matrices arising in linear-programming codes. The poor computation-to-I/O ratio in narrow supernodes also affects other levels of the memory hierarchy, such as memory-to-vector-registers traffic and memory-to-cache traffic; it was to alleviate this bottleneck that Duff and Reid [1983] originally proposed supernode amalgamation.

3.3 Etree-Oblivious Out-of-Core Factorizations

Rothberg and Schreiber [1996, 1999] also proposed a method that is oblivious to the etree. The method partitions the set $\{1, 2, \ldots, n\}$ into contiguous subsets of indices, and partitions the rows and columns of the matrix accordingly. The partitioning is done so that the nonzeros in each induced block of L occupy no more than one third of main memory. The factorization is carried out by applying an out-of-core dense factorization algorithm to the sparse blocks.

[2]In the experiments that we present in Section 6, the size of the largest frontal matrices would range from 20 MB (for the matrix ldoor) to 4 GB (for the matrix 500-100-100) if the matrices were stored in LAPACK's symmetric-packed format. If the matrices were stored in square arrays, the size of frontal matrices would roughly double. These numbers are based on actual measurements of front sizes during the factorization. See also the discussion in Rothberg and Schreiber [1999].

Rothberg and Schreiber report that this method performs poorly compared to supernodal-decomposition-based methods.

3.4 Compulsory-Subtree Out-of-Core Factorizations

All the remaining factorization methods that we discuss in this section utilize a fundamental I/O-reduction concept that we call *compulsory subtrees*. This concept was previously used by Gilbert and Toledo [1999], and in a limited form by Rothberg and Schreiber [1999], but it was never formally defined or named.

Suppose that we partition the set of supernodes of L into disjoint subsets that are factored one after the other. Furthermore, suppose that we keep no parts of L and no update matrices in main memory when we start the factorization of a subset. That is, when we complete the factorization of a subset we write to disk the columns of L that are were computed but not yet written to disk, as well as all the update matrices that are still in memory. We then clear main memory (except perhaps for the elimination tree and other structural data) and start the factorization of the next subset. Although clearing up memory in this way is not strictly necessary and may increase the amount of I/O, it greatly simplifies both memory management in the factorization code and the analysis of different factorization schedules. Most codes operate in a way that is consistent with this assumption, which we call the *cold-subtree assumption*. We use the term “cold” since the factorization of each subset starts with an empty, or cold, main memory. The next lemma explains why we use the term “subtree”.

LEMMA 3.1. *Under the cold-subtree assumption, and assuming that the factorization is done in some postorder of the elimination tree, the amount of I/O that a factorization code performs does not change if we partition each subset into subsets that form maximal connected subtrees of the elimination tree.*

PROOF. We first claim that if a subset s contains a supernode i and one of its descendants j, then the subset must include all the supernodes on the path between i and j in the etree. Suppose for contradiction that a supernode k along the path is in a different subset s'. If s' is factored before s, then k is factored before its descendant j, which is impossible, since the elimination ordering must be a postordering of the etree. If s' is factored after s, then i is factored before its descendant k, again a contradiction. Hence, the claim is true.

Therefore, if a subset includes more than one connected subtree of the etree, the maximal connected components of the subset do not have supernodes in an ancestor-descendant relationship. Since supernodes that are not in such a relationship do not update each other, the subset can be partitioned without increasing the amount of I/O that must be performed. □

This lemma implies that packing disconnected supernodes into a subset never reduces I/O. Therefore, from now on we assume that the subsets are connected. In particular, we can assume that each subset corresponds to a subtree of the etree.

Consider the I/O that a cold-subtree out-of-core algorithm performs when it factors a subtree s:

(1) Reading the columns of A that are part of the supernodes in s (only if A is not kept in main memory).
(2) Writing out columns of L to disk.
(3) Writing out the update matrix for the root of the subtree (in a multifrontal factorization).
(4) Reading columns of L and/or update matrices from descendants that are not in s of the supernodes in the subtree s.
(5) Writing update matrices for supernodes other than the root of the subtree (in a multifrontal factorization).
(6) Reading columns of L and/or update matrices from supernodes within the subtree.

An algorithm must perform I/O in the last two categories only when there is not enough memory to store these update matrices and the columns of L in main memory during the factorization of the subtree. Also, an algorithm must read columns from descendants of the subtree more than once only when there is not enough memory to store these descendant columns in main memory during the factorization of the subtree. In other words, on a computer with an infinite main memory, a cold-subtree out-of-core factorization algorithm only writes out each column of L once; it writes out only update matrices associated with root subtrees, reads only columns of L and/or update matrices from proper descendants of the subtree, and reads each one of these at most once. Hence, we classify these I/O operations as *compulsory* given the partitioning into subtrees. We classify other I/O operations, including writing out update matrices of non-root supernodes and reading them back, reading columns of L from within the subtree, and reading other columns of L more than once, as *capacity* operations, since they are performed due to the small capacity of main memory relative to the subtree, not due to the cold-subtree algorithm.

In general, partitioning the etree into large subtrees reduces compulsory I/O relative to a partitioning with small subtrees, but may result in capacity I/O. More specifically, merging parent-child subtrees always reduces compulsory I/O and may increase capacity I/O. Therefore, state-of-the-art out-of-core factorizations usually try to partition the etree into the largest possible subtrees that may be factored without any capacity I/O. Such partitions correspond to block decompositions of dense matrices in out-of-core algorithms, where choosing the largest block size that fits in main memory tends to minimize I/O.

We refer to cold-subtree out-of-core factorizations that only perform compulsory I/O as *compulsory-subtree* factorizations. The next two subsections describe Rothberg and Schreiber's compulsory-subtree algorithms, and our new new compulsory subtree algorithm, respectively.

3.5 Compulsory-Leaf-Subtree Factorizations

Rothberg and Schreiber [1999] describe a family of factorization methods that we call *compulsory-leaf-subtree* methods. Most of their methods partition the etree into large compulsory subtrees at the bottom of the etree and into compulsory single-supernode subtrees above them. That is, their algorithm finds subtrees all of whose leaves are leaves of the etree and chooses them to be as large as possible for factorization without any capacity I/O. Above these leaf subtrees, which they call domains, each subtree consists of a single supernode, which must fit in main memory (the symbolic elimination phase splits large supernodes so that each supernode fits in main memory).

The leaf subtrees are always factored using an in-core multifrontal method. The size of the leaf subtrees is chosen so that the stack of update matrices fits in main memory. During the factorization of a leaf subtree, columns of L can be written out to disk when they are no longer needed, which reduces memory usage and allows the algorithm to use larger leaf subtrees.

Rothberg and Schreiber propose several different compulsory-leaf-subtree methods that differ in how they update and factor the supernodes above the leaf subtrees. One method, which they call MF, continues with a multifrontal factorization. The main difficulty with this method is that it needs to break frontal matrices that do not fit in main memory into blocks of columns and to process each block separately. Another method, which they call PPLL, continues with a left-looking factorization, fetching supernodes of L as needed from disk to update the current supernode. This method fetches supernodes of L from disk whether they belong to a leaf subtree or to a nonleaf subtree. A third method, which they call PPLL_U, uses a hybrid approach to update supernodes in nonleaf subtrees. In this method, the update matrix of the root of each leaf subtree is written out to disk. When the method factors a supernode in a nonleaf subtree, it uses these update matrices to apply updates from supernodes in leaf subtrees, but uses columns of L that were previously written to disk to apply updates from other subtrees. This third method may save I/O relative to the PPLL method when several columns in a leaf subtree update the same element of the trailing matrix; in such cases, the update matrix represents all of these updates in a single number, the sum of the individual updates. However, when the update matrix of the root of a leaf subtree is large, storing and fetching it may increase the total amount of I/O in the algorithm relative to a partitioning that excludes this root from the leaf subtree. Rothberg and Schreiber propose a clever technique to choose the leaf subtrees in the PPLL_U method in a way that minimizes the total amount of I/O.

Rothberg and Schreiber also discuss a limited form of multi-supernode subtrees above leaf subtrees, which they call over-decomposition. They suggest that these subtrees should consist of a fixed number of supernodes, which means that supernodes must be limited in size more than if single-supernode subtrees are allowed. They do not explicitly state that the supernodes in a nonleaf subtree should form a connected subtree, but we assume that this is the case in their code. They report that over-decomposition consistently improved performance, but not by much, largely because the performance of their code was already satisfactory even without over-decomposition (i.e., I/O does not represent a significant fraction of the running time, even without over-decomposition).

3.6 General Left-Looking Compulsory-Subtree Factorizations

We now describe our new algorithm, which uses a more general subtree-partitioning scheme than Rothberg and Schreiber's algorithms, and which uses a hybrid right/left-looking method with a sophisticated schedule to factor compulsory subtrees.

We partition the etree into subtrees using a bottom up traversal. Before the partitioning begins, we compute the exact amount of memory required to store about 20 size-σ integer arrays, where σ is the number of supernodes. These arrays store the etree and references to supernodes of L stored on disk. We denote the remaining amount of memory by M'. The supernodes and subtrees are constructed so that the following properties always hold:

(1) Each supernode fits into at most one third of the remaining memory. That is, $\omega_j(\omega_j + \xi_j) \le M'/3$ for all j. Since M' depends on the number of supernodes, we actually enforce a slightly stronger constraint, that each supernode fits into one third of the remaining memory even if $\sigma = n$. Formally, we require that for all j,
$$\omega_j(\omega_j + \xi_j) \le \frac{1}{3}(M - 20n)\ .$$
(2) Within a subtree, for each subtree leaf, all the supernodes along the path from the leaf to the root of the subtree must fit into one third of the remaining memory. That is, denoting the leaf l and the root of the supernode r, we always have
$$\sum_{\substack{j_1=\ell \\ j_{i+1}=p(j_i)}}^{j_i=r} \omega_{j_i}(\omega_{j_i} + \xi_{j_i}) \le \frac{1}{3}M'.$$

Given this partitioning of the etree into supernodes and subtrees, we now describe the subtree-factorization method. When the factorization of a subtree s begins, all its descendant subtrees have already been factored and stored on disk. The algorithm factors the columns associated with s using a depth-first traversal of s. When a supernode j is first visited during the traversal, its nonzero structure is loaded from disk (it was stored on disk during the symbolic elimination phase), and memory is allocated for its nonzero elements. When the depth-first traversal is about to leave a supernode and return to its parent, the following operations are performed:

(1) It applies updates from descendants of every child i of j that is not in s to all the supernodes in s. That is, the algorithm enumerates the children i such that $p(i) = j$ and $i \notin s$. For each of these children, the algorithm traverses the subtree rooted at i of the elimination tree depth-first and applies updates from supernodes in this subtree to supernodes in s. We explain later how individual supernode-supernode updates are performed.

(2) Once these updates have been performed, we can factor j using LAPACK and the level-3 BLAS.

(3) The algorithm now performs a partial right-looking update, in which supernode j updates its ancestors in s. To determine which supernodes must be

updated, we simply compute the intersection of Ξ_j and the ancestors of j in s.

(4) The algorithm writes supernode j to disk and releases the memory it occupied.

Clearly, the supernodes that are stored in main memory at any point during this depth-first traversal lie on a single root-to-leaf path in s. Therefore, they occupy together no more than $M'/3$ memory. The following lemma shows that the algorithm is correct.

LEMMA 3.2. *The general left-looking compulsory-subtree algorithm correctly computes the Cholesky factorization of A.*

PROOF. To prove correctness, we need to show that every required supernode-supernode update is applied exactly once, and that all the required updates to a supernode are applied before it is factored.

Suppose that supernode k must update supernode j. If j and k belong to the same compulsory subtree s, then k is factored before j, because supernodes in s are factored in postorder. Since j must be an ancestor of k, supernode j updates supernode k immediately following the factorization of k.

We now analyze the case in which j and k belong to different compulsory subtrees. Let s be the compulsory subtree containing j, let l be the last supernode in s on the path from j to k, and let i be the first supernode not in s on the path from j to k. Supernode l is the parent of supernode j. When the algorithm processes l, it will apply the updates to l and its ancestors in s from the subtree rooted at i. This includes the k-j update. This shows that each required update is applied.

It is easy to see that no update is applied more than once. If j and k are in the same subtree, the update is applied exactly once, after k is factored. If j and k belong to different subtrees, the update is applied exactly once, just before i is factored. When proper ancestors of i are factored, updates from k are not applied, since i is in s, so the subtree rooted at l is not traversed again. $\square$

The supernode-supernode updates are performed using level-3 BLAS combined with scatter-gather operations. During an update, at least $M'/3$ words of memory are available, beyond the memory used for storing supernodes in s and the updating supernode k. This chunk of memory is used to allocate an in-core dense array X. Recall that the required update is a subtraction of

$$L_{(\Omega_j \cup \Xi_j) \cap \Xi_k, \Omega_k} L^T_{\Omega_j \cap \Xi_k, \Omega_k}$$

from supernode j. The two row sets $(\Omega_j \cup \Xi_j) \cap \Xi_k$ and $\Omega_j \cap \Xi_k$ are contiguous rows in supernode k, so a level-3 BLAS operation can multiply them directly. The product is stored into the array X. From X the update is scatter-subtracted into the compressed array storing supernode j. (The update may touch noncontiguous elements in the compressed representation of j, so a scatter operation is required.)

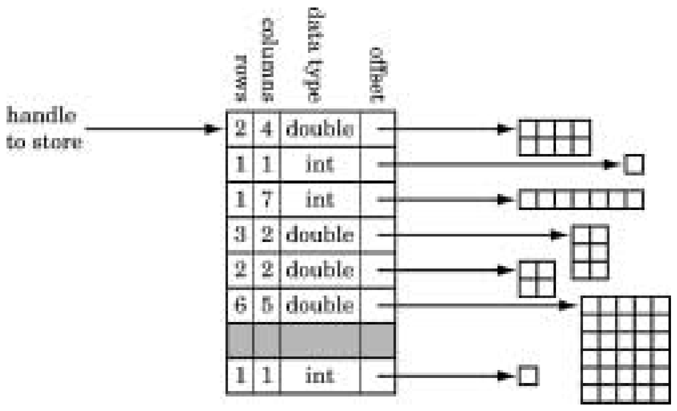

Fig. 4. A store. The store contains 7 matrices, with indices 0–5 and 7. No matrix was yet stored in position 6. Each entry in the store contains a matrix with a given number of rows and columns, a given data type, and a given offset in the file or files containing the store.

4. ON-DISK DATA STRUCTURES

Our solver uses a sophisticated persistent-storage abstraction called a *store*, which we have developed specifically for use in out-of-core sparse factorization codes. We use stores in this code, as well as in a new implementation of the Gilbert-Toledo sparse-pivoting LU factorization code, which is also part of the TAUCS library. A layer of special input-output subroutines accesses stores on behalf of the out-of-core factorization algorithm and on behalf of the corresponding triangular-solve subroutine. The first part of this section describes stores, and the second part describes how the code uses stores to represent the sparse Cholesky factor.

4.1 Stores

The basic storage abstraction that the I/O layer supports is called a *store*. A store is a one-dimensional array of rectangular matrices, as illustrated in Figure 4. A store can hold matrices of different dimensions and different primitive data types. That is, a matrix in a store can have any dimensions and can consist of integers, single- or double-precision floating-point real numbers, and single- or double-precision complex numbers. The number of matrices in a store is bounded only by the number of matrices that can be indexed using an integer. Matrices can be appended to a store in any order, but once appended, a matrix cannot be deleted from a store or resized. Deletion and resizing is not required in our out-of-core solver, and forbidding these operations simplifies the design of the I/O subroutines.

The store abstraction is designed to achieve three main objectives. First, it allows the code to store large factors on machines in which single files are limited to 2 or 4 GB. A single store can span many operating system files. Our I/O subroutines limit each file to 1 GB, but treat a collection of files as one contiguous store. Limiting files to 1 GB also circumvents portability issues that would have arisen from using 64-bit file pointers, and allows the constituent files to be transfered reliably between machines and operating systems. Second, treating the store as a one-dimensional array of matrices relieves our sparse solver from performing offset calculations to locate a particular piece of data. As long as the solver refers to data by a matrix index, the I/O subroutines can convert the matrix index to a store offset, which consists of the identity of a file and an offset within the file. Third, the abstraction allows the solver to perform all the I/O operations in terms of its own data structures, which consists of compressed supernodal blocks of L, of arrays of rows and column indices that represent the supernodes, and of arrays of integers that represent the elimination tree. By meeting these three objectives, the abstraction bridges the semantic gap between the I/O operations that an out-of-core sparse direct solver needs to perform and the I/O operations that the operating system can perform.

Store operations are performed in the following way. As long as a store is open, the code keeps in main memory the *metadata array* that describe matrix sizes, data types, and on-disk locations. Reading or writing an existing matrix is performed by looking up its on-disk location and calling the operating system to transfer the data to or from disk. If a single matrix spans more than one operating-system file, several calls to the operating system may be required. When a matrix with a given index is first written to a store, the code may need to extend the size of the metadata array. When a store is closed, the code writes the metadata array after the last matrix in the files, and then writes size and location of the metadata array in the first two words of the store, which are reserved for this purpose. When the code needs to open the store, it reads the first two words to locate the metadata array and to determine its size, and then reads the metadata array from disk. Saving the metadata array at the end of the store allows the array to grow without moving any of the matrices.

One significant restriction on stores is the fact that all the constituent files of a store must reside in the same directory. Capabilities such as logical-volume management and software disk arrays, which are now present in most operating systems, allow users to map multiple physical disks into a single directory that can store factors that do not fit on a single physical disk.

Our I/O subroutines also do not support nonblocking I/O. Our experience with previous out-of-core linear algebra software [Gilbert and Toledo 1999; Rabani and Toledo 2001; Toledo and Gustavson 1996; Toledo and Rabani 2001] have shown that performing nonblocking I/O adds significantly to the complexity of the software and to the difficulty of configuring it. These issues, together with the fact that earlier experiences with out-of-core sparse direct solvers [Gilbert and Toledo 1999; Rothberg and Schreiber 1999] have shown that out-of-core sparse factorizations do not spend significant amount of time waiting for I/O, led us to decide not to use nonblocking I/O.

4.2 Representing the Cholesky Factor in a Store

The store that represents the Cholesky factor includes not only the matrix element and their indices, but also the decomposition of the factor to supernodes, the elimination tree, and even the dimension of the factor. The inclusion of all of this data in the disk-resident store, together with the fact that the on-disk representation of a closed store includes its own metadata, ensures that the disk representation of the factor is self-contained. This allows us to create the factor in one program, which is then terminated. Another program can later solve linear systems using this factor, perhaps on a different machine. We can even move the factor to another file system, another machine, or even to backup media between the factor and the subsequent solves. (A large main memory reduces I/O significantly in the factor phase, but not in the solve phase; therefore, it is sometimes best to perform the factorization on an expensive machine with a large main memory, but the subsequent solves on a cheaper machine with a smaller memory.)

5. THE OVERALL STRUCTURE OF THE OUT-OF-CORE SOLVER

As stated above, the solver is part of TAUCS, a library of sparse linear solvers. In TAUCS, sparse matrices are represented in a compressed-column format. This is the input format to the out-of-core solver. The compressed-column format stores the values of the nonzeros in the lower triangle of A, as well as their row indices. The nonzeros and indices are stored one column after the other, and an array specifies the start position of each column. The compressed-column format also specifies the data type of the matrix's elements (single or double precision, real or complex) and the fact that it is a symmetric matrix represented using its lower triangle.

The solution starts with an ordering phase. First, a fill-reducing ordering is found. The user can choose between several symmetric ordering codes, such as METIS [Karypis and Kumar 1998], Liu's GENMMD, and AMD [Amestoy et al. 1996], which are all interfaced to TAUCS. For large problems, METIS is usually the most effective among these. Next, the rows and columns of the matrix are permuted, resulting in a new reordered compress-column matrix. In principle, the original input matrix is no longer needed, but it is required for computing the residual and its norm, and perhaps for iterative refinement of the solution. Although the ordering and permuting steps are performed using an in-core algorithm, the input matrices are typically small enough to be reordered in main memory. When the matrix is particularly large or main memory particularly small, the ordering phase may cause intensive paging in the virtual-memory system, but even in these cases the ordering phase is typically much cheaper than the subsequent factorization phase. We are not aware of any out-of-core matrix reordering code (and as explained, one is not really needed).

The next phase involves creating an empty store. This is performed by a routine that receives a single string argument. The argument specifies the path and base name for the operating system files that will constitute the store. For example, the string `"/tmp/chol"` causes the files of the store to be named `/tmp/chol.0`, `/tmp/chol.1`, and so on.

Next, the matrix is factored and the factor is stored in the empty store. This operation is performed by a single TAUCS routine with only three arguments: the input matrix, a handle to the empty store, and the amount of main memory to be used in the factorization. TAUCS contains an auxiliary system-dependent routine that can usually determine the amount of physical memory of the machine, so the user need not specify this amount by hand. Instructing the out-of-core factorization routine to use less than or more than the true amount of physical memory reduces performance. Using too little memory causes the algorithm to use small compulsory subtrees and to perform more I/O. Using too much memory forces the use of virtual memory to store the compulsory subtrees and other data structures, which causes paging. In this case, I/O is performed both explicitly by the out-of-core algorithm and implicitly by the virtual-memory system, which tends to increase the total amount of I/O and reduce performance. In extreme cases, or on machines configured with small virtual memories, instructing the algorithm to use a large amount of memory can cause it to fail completely due to memory-allocation failure. To summarize, we advise users to instruct the algorithm to use an amount of memory equal to or slightly smaller than the amount of physical memory, and we provide a routine to determine this size.

The factorization routine itself goes through several steps to factor the matrix. First, the elimination tree is computed using Liu's algorithm [Liu 1990]. Next, the algorithm performs a symbolic elimination phase, which determines the nonzero structure of the Cholesky factor and identifies supernodes. The cost of determining the elimination tree is essentially linear in the number of nonzeros in A, and the cost of the symbolic analysis is essentially linear in the number of nonzeros in the factor L. TAUCS also includes an implementation of the Liu-Ng-Peyton algorithm [Liu et al. 1993] for determining the structure of fundamental supernodes in time almost linear in the size of A. Using this algorithm would significantly reduce the cost of the symbolic analysis phase, but since it is low in any case, we do not use this optimization. TAUCS also includes a routine that amalgamates small supernodes, but we do not currently use it in the out-of-core solver. The symbolic analysis phase constrains the size of supernodes to one third of the available memory, as explained in Section 3.6. The symbolic analysis phase is performed in main memory. The symbolic analysis is followed by a decomposition into compulsory subtrees. The matrix is then factored and the supernodes written into the store.

To solve a linear system using the out-of-core factor, the right-hand side is first permuted, to conform to the row and column ordering of the matrix. Next, the user's code calls the out-of-core solve routine. The solve routine performs the forward and backward triangular solves, each time reading the factor from disk. The triangular solutions operate recursively, traversing the elimination tree from top to bottom or from bottom to top. To enable this approach, the elimination tree is written to the store during the factorization phase, along with the factor itself.

Finally, the user's code permutes the solution vector to conform to the input ordering of the unknowns, computes the residual and perhaps refines the solution iteratively, and then deletes the out-of-core factor if it is not needed. If multiple solves are needed in a sequence, the out-of-core factor can be kept.

6. RESULTS

This section presents experimental results demonstrating the effectiveness and high performance of our general compulsory-subtree out-of-core factorization. The experiments were designed to address the following questions:

(1) How reliable is the algorithm?
(2) What is the performance of the out-of-core algorithm relative to that of an in-core algorithm? In other words, what is the impact of I/O on the performance of this algorithm?
(3) How does the amount of main memory affect the performance of the algorithm?
(4) Is the running time of the out-of-core algorithm predictable? This is an important issue for an algorithm that runs for hours, since if the running time is unpredictable, the user cannot know if he/she can expect the answer in an hour or in a week.
(5) What is the effect of keeping only one path of supernodes in memory, rather than an entire compulsory subtree?

In the following experiments, we refer to the general compulsory-subtree algorithm presented in Section 3.6 as the *paged-subtree* algorithm (since it pages supernodes in and out of memory during the factorization of a subtree). We refer to a variant in which an entire compulsory subtree is kept in memory as the *subtree-in-memory* algorithm. We note that even the less sophisticated subtree-in-memory method is more advanced than Rothberg and Schreiber's PPLL method, which only allows a single or a fixed number of supernodes per subtree. Rothberg and Schreiber's code is not publicly available, so we could not directly test our code against theirs.

Regarding the second question, it is worth noting that our experiments compare the total cost of our out-of-core algorithms with the cost of in-core algorithms, but the experiments do not isolate the cost of I/O. Our out-of-core algorithm is significantly different than in-core methods, in that it combines left-looking and right-looking updates, whereas in-core methods are typically either left-looking or multifrontal. Therefore, it is possible that some of the performance degradation in going out of core are due to the details of the algorithm and not to I/O. Our code is instrumented to measure time spent in I/O operations, but these measurements are not reliable, since the operating system performs much of the I/O, especially writes, in the background. Therefore, we do not attempt to quantify I/O and other overheads.

6.1 Test Matrices

We used three classes of matrices for testing our algorithm; these are described in Table I. The first class consists of test matrices from the PARASOL test-matrix collection.[3] These are the only test matrices that we could find that cannot be factored in main memory on a machine with 768 MB of main memory. All the

[3]`http://www.parallab.uib.no/parasol/data.html`.

Table I. The Matrices that We Used to Evaluate the Algorithm

Matrix	dim(A)	nnz(A)	nnz(L)	bytes(L)	flops
inline1	5.05e+05	1.87e+07	1.76e+08	1.59e+09	1.52e+11
ldoor	9.52e+05	2.37e+07	1.43e+08	1.29e+09	7.39e+10
audikw1	9.44e+05	3.93e+07	1.26e+09	1.11e+10	5.95e+12
40-40-40	6.40e+04	2.51e+05	1.42e+07	1.36e+08	1.55e+10
50-50-50	1.25e+05	4.93e+05	3.83e+07	3.55e+08	6.69e+10
60-60-60	2.16e+05	8.53e+05	8.66e+07	7.95e+08	2.19e+11
70-70-70	3.43e+05	1.36e+06	1.70e+08	1.54e+09	5.84e+11
80-80-80	5.12e+05	2.03e+06	2.95e+08	2.59e+09	1.30e+12
90-90-90	7.29e+05	2.89e+06	4.89e+08	4.23e+09	2.72e+12
100-100-100	1.00e+06	3.97e+06	7.70e+08	6.57e+09	5.31e+12
110-110-110	1.33e+06	5.29e+06	1.16e+09	9.80e+09	9.54e+12
120-120-120	1.73e+06	6.87e+06	1.72e+09	1.44e+10	1.71e+13
130-130-130	2.20e+06	8.74e+06	2.40e+09	2.00e+10	2.76e+13
140-140-140	2.74e+06	1.09e+07	3.28e+09	2.72e+10	4.37e+13
500-30-30	4.50e+05	1.77e+06	1.18e+08	1.07e+09	1.62e+11
500-40-40	8.00e+05	3.16e+06	3.05e+08	2.73e+09	7.23e+11
500-50-50	1.25e+06	4.95e+06	6.43e+08	5.71e+09	2.37e+12
500-60-60	1.80e+06	7.14e+06	1.14e+09	9.91e+09	5.87e+12
500-70-70	2.45e+06	9.73e+06	1.87e+09	1.59e+10	1.25e+13
500-80-80	3.20e+06	1.27e+07	2.84e+09	2.38e+10	2.39e+13
500-90-90	4.05e+06	1.61e+07	4.13e+09	3.44e+10	4.31e+13
500-100-100	5.00e+06	2.00e+07	5.82e+09	4.82e+10	7.50e+13

The first three are from the PARASOL test-matrix collection, and the others are the Laplacians of regular 3D meshes of the dimensions shown. For each matrix, the table shows its dimension, the number of nonzeros in its lower triangle, the number of nonzeros in the lower triangle of its Cholesky factor (using METIS reordering prior to the factorization), the size of its factor file in bytes, and the number of floating-point operations performed during the factorization.

other matrices that we have obtained from test-matrix collections are too small and do not exercise the out-of-core capability of our solver.

To supplement these matrices, we also generated matrices from regular 7-point discretizations of the Poisson equation in 3D. The graphs of these matrices are regular 3D meshes. We generated both N-by-N-by-N meshes, and 500-by-N-by-N meshes. The N-by-N-by-N meshes generate relatively dense factors (their top-level separator has N^2 vertices, so the topmost supernode is usually an N^2-by-N^2 dense matrix). For $N \ll 500$, which is the range that we used, the 500-by-N-by-N meshes smaller separators and generates sparser factors.

The amount of disk space required to factor these matrices is given in the fifth column of Table I (the code only stores the factor itself on disk).

6.2 Test Environment

We performed the experiments on two IBM Intel-based workstations, which are identical except for the amount of main memory they have. Both machines have 2 GHz Pentium 4 processors with 512 KB level-2 caches. The main memory of both uses RDRAM chips (Pentium 4 computers can utilize several kinds of memory chips, such as SDRAM, DDR, and RDRAM; the RDRAM chips provide the highest memory bandwidth). One machine has 768 MB of main memory, and the other has 192 MB (the second machine has more than 192 MB of memory installed, but we configured the operating system to only use 192 MB; this is completely equivalent to a machine with 192 MB of memory). We used machines with two different amounts of memory in order to measure the effect of memory-size on the performance of the algorithm.

The 768 MB configuration is more typical of the machines we expect users to use. Therefore, the performance of the algorithm on that machine is more representative than the performance on the 192 MB machine, which is used mostly to show how the algorithm behaves with very little main memory.

Each machine had two hard-disks installed. A 40 GB IDE hard disk was used to store the input matrices and experiment logs, but nothing else. A 75 GB IBM DeskStar IDE disk was used for swap space and to store the Cholesky factors, one at a time. Each machine used two swap partitions of 2 GB each at the fast end of the 75 GB disk. The rest of the disk was formated as a file system for the factors. Nothing else was stored on the disks (the machines use a remote root file system). The DeskStar disk that was used for paging and for the Cholesky factor files has an I/O bandwidth of about 36 MB/s at the fast end, which degrades to about 18 MB/s at the slow end. This I/O bandwidth is fairly typical for disks today (early 2003); the 40 GB disk, which is slightly newer, has bandwidths ranging from 46 to 25 MB/s. We measured the bandwidth using a program called `zcav`[4] by Russel Coker.

The machines run Linux with a 2.4.19 kernel. We compiled our code with the GCC C compiler, version 2.95.3, and with the `-O3` compiler option. We used ATLAS[5] Version 3.4.1 [Whaley and Dongarra 1998] by Clint Whaley and others for the BLAS. This version exploits vector instructions on Pentium 4 processors (these instructions are called SSE2 instructions). Using these BLAS routines and these machines, our in-core multifrontal sparse factorization code factors matrices whose graphs are 3D meshes at a rate of approximately 1.6×10^9 flops.

We used METIS[6] [Karypis and Kumar 1998] version 4.0 to symmetrically re-order the rows and columns of the matrices prior to factoring them.

The graphs and tables use the following abbreviations: IC (in-core), OOC (out-of-core), SIM (subtree-in-memory), and PS (paged subtree).

6.3 Baseline Tests

To establish a performance baseline for our experiments, we compare the performance of our in-core code to that of MUMPS version 4.3 [Amestoy et al. 2000, 2001, 2003]. MUMPS is a parallel and sequential in-core multifrontal factorization code for symmetric and unsymmetric matrices. We tested the sequential version, with options that tell MUMPS that the input matrix is symmetric positive definite and that instruct it to use METIS to preorder the matrix. We used the default values for all the other run-time options. This setup results in a multifrontal factorization that is quite similar to TAUCS's in-core multifrontal factorization.

4 `http://www.coker.com.au/bonnie++/`.
5 `http://math-atlas.sourceforge.net`.
6 `http://www-users.cs.umn.edu/~karypis/metis/`.

Table II. The Results of the In-Core Baseline Tests

	MUMPS			TAUCS		
Matrix	analyze	factor	flops	analyze	factor	flops
40-40-40	2.0	14.7	1.77e+10	3.3	11.7	1.60e+10
50-50-50	4.2	55.2	7.83e+10	4.3	41.0	6.84e+10
bmwcra1	6.2	59.3	6.26e+10	6.5	51.8	6.25e+10
crankseg1	2.5	30.8	3.26e+10	2.4	25.3	3.38e+10
crankseg2	3.2	41.6	4.62e+10	2.9	34.9	4.82e+10
thread	1.4	33.4	3.63e+10	1.4	23.4	3.74e+10

The table shows the running times, in seconds, of the analysis phase and the numerical factorization phase of MUMPS and TAUCS on a set of 6 relatively small matrices that can be factored in main memory. For both codes, the factorization is multifrontal, assumes that the input matrix is symmetric-positive definite, and uses METIS to reorder the matrices. The table also shows the number of floating-point operations that each code performed during the numerical factorization phase.

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 is used for all the other experiments.

We compared the two codes on the 768 MB machine, on two regular meshes and four matrices from the PARASOL test-matrix collection. All of these matrices and their factors fit in main memory without causing any paging activity. The matrices were selected arbitrarily.

The results of the baseline tests, which are presented in Table II, show that the in-core performance of TAUCS is quite similar to the in-core performance of MUMPS. There are small differences in the number of floating-point operations that each code performs, due to different supernode amalgamation strategies. TAUCS' numerical factorization is faster on these matrices. This test is not a comprehensive comparison of these codes, and we do not claim that TAUCS is faster in general. For example, some of the difference might be due to the different compilers that were used. However, the results do indicate that the in-core performance of TAUCS, which we use to assess the performance of our out-of-core algorithm, is representative of high-quality modern sparse Cholesky factorization codes.

6.4 Performance

The complete results of our experiments are shown in Tables III and IV, but the results are also summarized using easier-to-understand graphs. One thing that is easier to see in the tables, however, is the set of matrices on which the out-of-core factorization failed. As the tables show, this happens only on the 192 MB machine. When the factorization failed, this was always due to running out of memory. In some cases the algorithm determined that $M \leq 20n$ and stopped, and in a few other cases the algorithm failed to allocate sufficient memory during the factorization. Clearly, given a sufficiently large matrix the algorithm can also fail on machines with large main memories, but our experiments show

Table III. Complete Results on the 192 MB Machine

Matrix	$T_{\text{IC-MF}}$	$T_{\text{IC-LL}}$	$T_{\text{OOC-SIM}}$	$IO_{\text{OOC-SIM}}$	$T_{\text{OOC-PS}}$	$IO_{\text{OOC-PS}}$
inline1	1161	567	276	3.00e+09	270	2.74e+09
ldoor	1095	843	264	2.10e+09	250	1.81e+09
audikw1			7642	1.18e+11	7629	1.18e+11
40-40-40	36	17	21	2.14e+08	23	2.14e+08
50-50-50	258	116	103	7.71e+08	102	7.65e+08
60-60-60	994	542	324	2.90e+09	310	2.53e+09
70-70-70	2234	3417	834	8.58e+09	819	8.08e+09
80-80-80			1841	2.04e+10	1852	2.05e+10
90-90-90			4237	5.66e+10	4254	5.60e+10
100-100-100			9468	1.51e+11	9456	1.51e+11
110-110-110			20803	3.76e+11	10863	3.77e+11
120-120-120			55773	1.17e+12	55754	1.17e+12
130-130-130						
140-140-140						
500-30-30			257	2.02e+09	266	2.28e+09
500-40-40			1020	9.75e+09	1017	9.20e+09
500-50-50			3447	3.96e+10	3461	3.98e+09
500-60-60			10126	1.45e+11	10152	1.44e+11
500-70-70						
500-80-80						
500-90-90						
500-100-100						

The table shows factorization times in seconds for the multifrontal and left-looking in-core methods, as well as for the out-of-core subtree-in-memory and paged-subtree methods. The table also shows the total I/O volume in bytes for the two out-of-core methods. Missing numbers imply that the factorization failed due to lack of memory. The factorization times include the construction of the elimination tree and the symbolic analysis phases.

that even on a fairly standard machine (the one with 768 MB) the algorithm can successfully factor huge matrices.

Figure 5 provides an overview of the performance of the solver. The data in the figure shows that on a machine with 768 MB of main memory, our algorithm can factor matrices whose factors have 3–4 billion nonzeros overnight (10–12 hours), and matrices whose factors have about 6 billion nonzeros in around 24 hours. The data also shows that the algorithm can factor matrices whose factors have around one billion nonzeros in an hour or two.

The data in Figure 5 also shows that the amount of storage required to store the factor is slightly over 8 bytes per nonzero, the amount required to store the double-precision floating-point numbers. Hence, the overhead of storing indices and other information is negligible.

Figure 6 shows the performance of the algorithm in floating-point operations per second. The data shows that on the machine with 768 MB of main memory, the algorithm usually achieves a performance of 0.8–1 Gflop/s. This level of performance is about 50% of the performance of the in-core multifrontal algorithm on small matrices. The most important conclusion from this data is that the performance of the algorithm is acceptable despite the large amount of I/O that it performs.

Table IV. Complete Results on the 768 MB Machine

Matrix	$T_{\text{OOC-SIM}}$	$IO_{\text{OOC-SIM}}$	$T_{\text{OOC-PS}}$	$IO_{\text{OOC-PS}}$
inline1	200	2.14e+09	201	2.14e+09
ldoor	139	1.67e+09	143	1.67e+09
audikw1	4555	4.12e+10	4570	4.13e+10
40-40-40	20	1.41e+08	21	1.41e+08
50-50-50	73	5.88e+08	73	5.88e+08
60-60-60	209	1.26e+09	208	1.26e+09
70-70-70	566	3.65e+09	569	3.31e+09
80-80-80	1262	8.72e+09	1290	8.99e+09
90-90-90	2465	1.69e+10	2474	1.63e+10
100-100-100	4574	3.75e+10	4508	3.39e+10
110-110-110	8032	6.89e+10	8077	6.94e+10
120-120-120	16304	1.52e+11	16295	1.52e+11
130-130-130	25235	2.87e+11	25234	2.84e+11
140-140-140	42825	5.30e+11	42840	5.32e+11
500-30-30	181	1.61e+09	181	1.61e+09
500-40-40	716	5.44e+09	704	4.42e+09
500-50-50	2141	1.47e+10	2164	1.33e+10
500-60-60	5083	3.60e+10	5070	3.48e+10
500-70-70	10603	8.77e+10	10422	7.92e+10
500-80-80	20319	1.89e+11	20348	1.86e+11
500-90-90	40028	4.60e+11	39951	4.54e+11
500-100-100	78905	1.08e+12	79084	1.08e+12

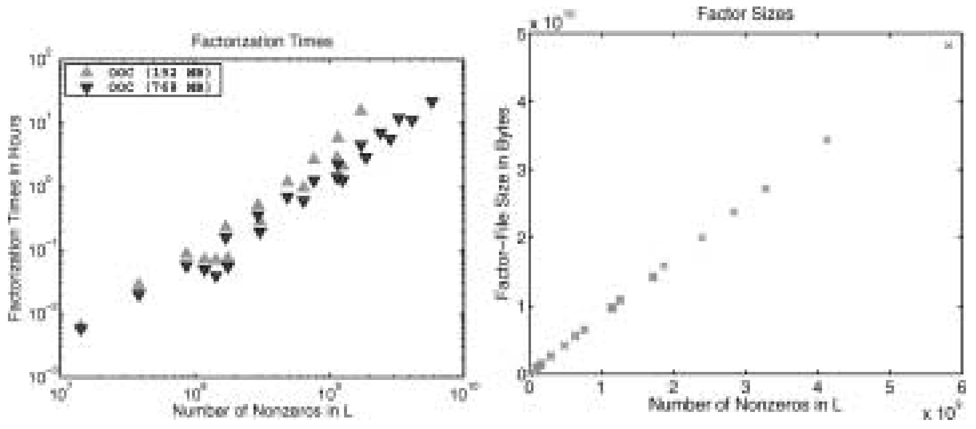

Fig. 5. Out-of-core factorization times and the sizes of the resulting Cholesky factors, as a function of the number of nonzeros in the factor. The data in the plot on the left shows the performance on two machines with different amounts of memory.

The data in Figure 6 also shows that the amount of main memory has a significant impact on performance. The algorithm achieves significantly higher performance on the machine with 768 MB of main memory than on the machine with only 192 MB of memory. This is an expected consequence of the fact that more memory allows the algorithm to use larger subtrees and to perform less I/O. Another memory-size-related phenomenon that is evident in Figure 6 is that as matrices grow, performance declines, although not dramatically. This

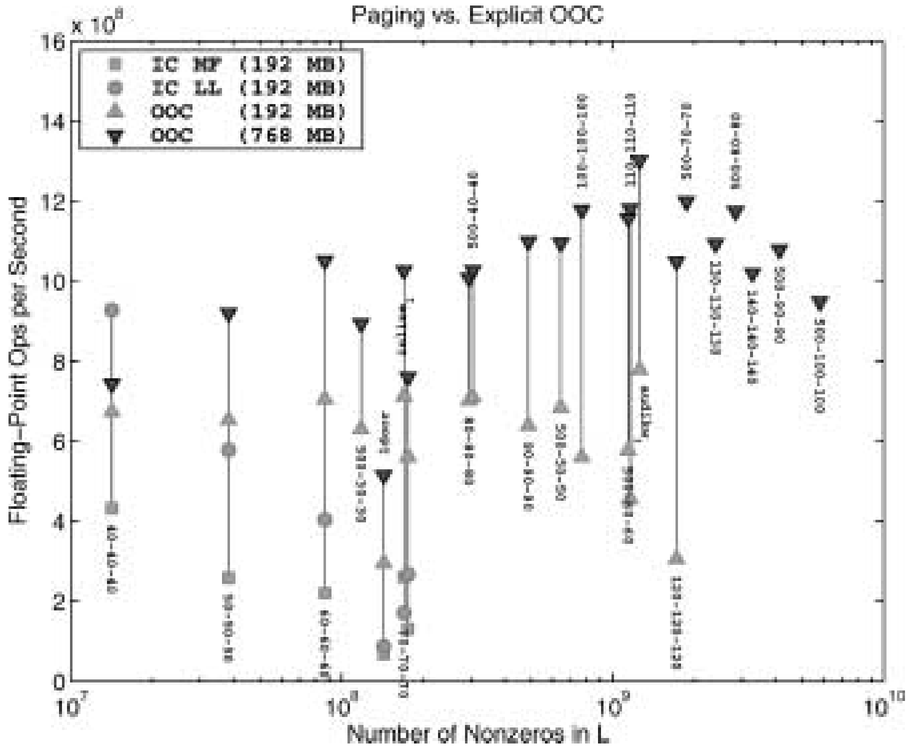

Fig. 6. Performance of the out-of-core factorizations. The figure shows the performance of the out-of-core factorization on two machines with different amounts of memory, as well as the performance of two in-core algorithms that use the operating system's demand-paging mechanism.

is evident only on the 192 MB machine, on which large matrices leave little memory for subtrees of the factor. This causes the amount of memory left for subtrees to shrink and the I/O activity to increase. On the machine with 768 MB, none of these matrices fill a substantial amount of the main memory, so this behavior does not occur, although it would occur on even larger matrices.

Another important conclusion that we can draw from Figure 6 is that an explicit out-of-core algorithm is far more efficient than an in-core algorithm that uses the operating system's demand-paging mechanism. On large matrices, the in-core algorithms are slower by factors of up to 3. The difference is probably due to the better locality in the explicit out-of-core algorithm, which builds a schedule that is specifically designed for the machine's memory size. Note that the swap area (page file) was stored on the fastest area of the same disk that was also used to store the out-of-core factors. Obviously, we can only factor in-core matrices whose factors are smaller than 2 GB on these 32 bit machines, so the out-of-core algorithms have justification beyond performance.

Following the ordering and symbolic analysis phases of the direct solver, which are relatively fast, the remaining factorization time can be estimated fairly accurately. Figure 6 shows that on the 768 MB machine, the computational rate is between 0.7 and 1.3 Gflop/s. Therefore, after the symbolic analysis

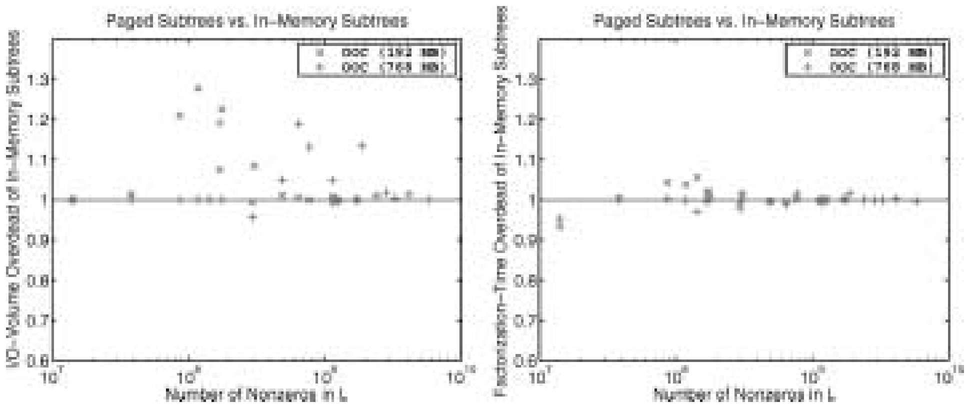

Fig. 7. I/O-volume and factorization-time overheads due to subtree-in-memory scheduling. The baseline of each graph is the performance of the paged-subtree algorithm. The markers show the performance of the subtree-in-memory algorithm relative to the paged-subtree algorithm.

phase computes the number of flops required to factor the matrix, the running time of the numerical factorization phase can be determined to within a factor of 2. This is important, since it means that the user can make an informed decision regarding whether to wait for the factorization to end or to abort the solver, if he or she cannot afford to wait for the factorization to complete. Similarly, the exact nonzero count in the factor that the symbolic analysis phase provides allows the algorithm to determine whether it can successfully factor a matrix within a given amount of disk space. If space is insufficient, the factorization can be aborted before a single floating-point operation is performed. The computational rate of the factorization on the 192 MB machine varies more since much of the memory of this machine is often consumed by the matrix itself.

Figure 7 compares the performance of our best algorithm, the paged-subtree algorithm, to that of the subtree-in-memory algorithm. The data clearly shows that the paged-subtree algorithm usually performs significantly less I/O (up to 25% less), and never significantly more I/O. The difference is particularly large on mid-sized matrices, where size is relative to the amount of memory. This happens since on small matrices, the entire factor fits into a single subtree or very few subtrees in any case, so the subtree-decomposition method does not matter much. On larger matrices whose factors are much larger than memory, the improved locality of the paged-subtree algorithm saves I/O. The point at which this happens depends, of course, on the size of main memory. But on the largest matrices, most of the I/O is performed in the context of single-supernode subtrees near the top of the elimination tree. When only one supernode fills a subtree there is no difference between the subtree-decomposition methods, so on these large matrices paged-subtree scheduling does not matter much. In one case the paged-subtree algorithm performs more I/O than the subtree-in-memory algorithm. We think that this is caused by the greedy nature of our scheduling algorithm, which always prunes the largest possible subtree from the bottom of the elimination tree. It seems that in this particular mid-size case, using a larger subtree at the top would have been better.

In spite of the difference in I/O volume between the two subtree-decomposition methods, the factorization times are fairly similar: always within 10% of each other, with a quite a few cases in which subtree-in-memory is faster. The discrepancy is caused by the relatively high performance of the disks that we used: I/O is fast enough that a 25% reduction in I/O does not translate into a significant saving in time. We anticipate that on machines with faster processors (or more processors) and/or slower disks, the factorization times would be better correlated with I/O volume.

7. DISCUSSION AND CONCLUSIONS

Our out-of-core algorithm can solve enormous systems within a reasonable amount of time on typical workstations. The out-of-core algorithm avoids the need for still-rare and still-expensive 64-bit workstations with tens of gigabytes of main memory, or the need for a cluster, one of which is necessary for solving such systems in main memory.

For many classes of matrices, the scaling behavior of sparse direct triangular factorization is superlinear in the number of nonzeros in A, as known from theory[7] and as demonstrated in Figure 5. But while the asymptotic behavior is superlinear, the constants involved are small, which allows us to solve huge linear systems within reasonable amounts of time. In particular, our algorithm can solve a matrix such as AUDIKW, the largest matrix in any test-matrix collection, whose dimension is almost a million, in about an hour and 15 minutes. The algorithm can factor a matrix whose graph is a 140-by-140-by-140 mesh with over 2.7 million vertices in about 12 hours. We believe that the size of linear systems that our algorithm can solve on typical workstations overnight or during a lunch break makes the algorithm useful.

The algorithm is highly reliable, in that the amount of main memory it needs is proportional to n, the dimension of the system, and not to the number of nonzeros in A or in L. This implies that our algorithm can handle almost any matrix that any solver, whether iterative or direct, can solve.

The running time and storage requirements of the algorithm are predictable. The symbolic elimination phase determines the size of the factor and the number of floating-point operations to compute it. Given that with sufficient main memory the algorithm operates within a fairly narrow floating-point-operations-per-second range, the running time can be predicted to within a factor of about 2. This feature allows the user to decide whether he/she wants to spend the computation time and disk space to solve a given problem, before significant resources have been used. Since the column counts in L can be computed even before the symbolic elimination, this information can be made available even earlier than we now provide it.

[7]For example, Hoffman, Martin and Rose [Hoffman et al. 1973] show that $\Theta(n^{3/2})$ multiplications are required to factor any symmetric permutation of a matrix whose graph is a $\sqrt{n}$-by-$\sqrt{n}$ regular mesh. Lipton, Rose, and Tarjan [Lipton et al. 1979] generalized this result by showing that the number of multiplications is always bounded from below by a function proportional to the cube of the smallest approximately-balanced vertex separator in the graph of the matrix.

ACKNOWLEDGMENTS

Thanks to Ed Rothberg for discussions regarding the details of his sparse Cholesky factorization algorithms. Thanks to Didi Bar-David for configuring the operating system on the test machines and for telling us about `zcav`. Thanks to John Reid, the associate editor, and to the two anonymous referees for helpful comments, and suggestions.

REFERENCES

Amestoy, P. R., Davis, T. A., and Duff, I. S. 1996. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Applic. 17*, 886–905.

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

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. Applic. 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`.

Ashcraft, C. and Grimes, R. 1989. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Soft. 15*, 4, 291–309.

Bjørstad, P. E. 1987. A large scale, sparse, secondary storage, direct linear equation solver for structural analysis and its implementation on vector and parallel architectures. *Parallel Computing 5*, 1, 3–12.

Duff, I. S., Erisman, A. M., and Reid, J. K. 1986. *Direct Methods for Sparse Matrices*. Oxford University Press.

Duff, I. and Reid, J. 1983. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Soft. 9*, 302–325.

George, J. A., Heath, M. T., and Plemmons, R. J. 1981. Solution of large-scale sparse least squares problems using auxiliary storage. *SIAM J. Sci. Statis. Comput. 2*, 4, 416–429.

George, A. and Liu, J. W. H. 1981. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall.

George, A. and Rashwan, H. 1985. Auxiliary storage methods for solving finite element systems. *SIAM J. Sci. Statis. Comput. 6*, 4, 882–910.

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, Texas. 10 pages on CDROM.

Hoffman, A. J., Martin, M. S., and Rose, D. J. 1973. Complexity bounds for regular finite difference and finite element grids. *SIAM J. Numer. Anal. 10*, 364–369.

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

Lipton, R. J., Rose, D. J., and Tarjan, R. E. 1979. Generalized nested dissection. *SIAM J. Numer. Anal. 16*, 346–348.

Liu, J. W. H. 1986. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Soft. 12*, 3, 249–264.

Liu, J. W. H. 1987. An adaptive general sparse out-of-core Cholesky factorization scheme. *SIAM J. Sci. Statis. Comput. 8*, 4, 585–599.

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

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

Liu, J. W. H., Ng, E. G., and Peyton, B. W. 1993. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Applic. 14*, 242–252.

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

Poole, G., Liu, Y.-C., and Mandel, J. 2001. Advancing analysis capabilities in ansys through solver technology. In *Proceedings of the 10th Copper Mountain Coference on Multigrid Methods*. Copper Mountain, Colorado.

Rabani, E. and Toledo, S. 2001. Out-of-core SVD and QR decompositions. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*. Norfolk, Virginia. 10 pages on CDROM.

Rothberg, E. and Gupta, A. 1991. Efficient sparse matrix factorization on high-performance workstations—exploiting the memory hierarchy. *ACM Trans. Math. Soft. 17*, 3, 313–334.

Rothberg, E. and Schreiber, R. 1996. An alternative approach to sparse out-of-core factorization. presented at the 2nd SIAM Conference on Sparse Matrix, Coeur d'Alene, Idaho.

Rothberg, E. and Schreiber, R. 1999. Efficient methods for out-of-core sparse cholesky factorization. *SIAM J. Sci. Comput. 21*, 129–144.

Schreiber, R. 1982. A new implementation of sparse gaussian elimination. *ACM Trans. Math. Soft. 8*, 256–276.

Toledo, S. 1997. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Anal. Applic. 18*, 4, 1065–1081.

Toledo, S. and Gustavson, F. G. 1996. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of the 4th Annual Workshop on I/O in Parallel and Distributed Systems*. Philadelphia, 28–40.

Toledo, S. and Rabani, E. 2001. Very large electronic structure calculations using an out-of-core filter-diagonalization method. *J. Computational Physics*.

Whaley, R. C. and Dongarra, J. J. 1998. Automatically tuned linear algebra software. Tech. rep., Computer Science Department, University Of Tennessee. Available online at www.netlib.org/atlas.

Received February 2003; revised September 2003; accepted September 2003