# The Design of I/O-Efficient Sparse Direct Solvers

Florin Dobrian and Alex Pothen*

**Abstract**

We consider two problems related to I/O: First, find the minimum primary memory size required to factor a sparse, symmetric matrix when permitted to read and write the data exactly once. Second, find the minimum data traffic between core and external memory when permitted to read and write the data many times. These problems are likely to be intractable in general, but we prove upper and lower bounds on these quantities for several model problems with useful sparsity (i.e., whose computational graphs have small separators). We provide fast algorithms for computing these quantities through simulation for irregular problems. The choice of factorization algorithms (left-looking, right-looking, multifrontal), orderings (nested dissection or minimum degree), and blocking techniques (1- or 2- dimensional blocks) can change the memory size and traffic by orders of magnitude. Explicitly moving the data (files managed by the program) improves performance significantly over implicit data movement (pages managed by the operating system). Thus this work guides us in designing a software library that implements an external memory sparse solver.

## 1 Introduction

We consider two problems that arise in designing external-memory algorithms for solving large, sparse systems of linear equations by direct (factorization-based) methods. Although our main interest is in the disk/core interface (also known as external/internal or secondary/primary), conceptually this study applies to two adjacent layers of any hierarchical storage system.

*Old Dominion University, Norfolk, VA 23529 ({dobrian,pothen}@cs.odu.edu) and ICASE, NASA Langley Research Center, Hampton VA 23681 (pothen@icase.edu). This work was supported by U. S. National Science Foundation grants DMS-9807172; by the U. S. Department of Energy under subcontract B347882 from the Lawrence Livermore Laboratory; and by NASA under Contract NAS1-19480 while the second author was in residence at ICASE.


SC2001 November 2001, Denver (c) 2001 ACM 1-58113-293-X/01/0011 $5.00

In a two-layer storage system, the lower layer (the disk) is assumed to have unlimited capacity, as opposed to the upper layer (the core), which is assumed to have a limited capacity. The first problem corresponds to the situation when the traffic is minimum: when each input data item is read exactly once, each output data item is written exactly once and no temporay data item is moved between primary and external memory. The problem is to minimize the primary memory size needed to factor a sparse matrix under these conditions. We call this scenario read-once/write-once.

The second problem is to minimize the data traffic (between the primary and external memory) needed to factor a sparse matrix, when the input data is read once, but the output and the temporaries may be read and written as many times as needed. We call this the read-many/write-many scenario. The major goal is to design and implement algorithms that reduce the traffic but investigating the primary memory size provides helpful insight, as we shall show.

These two problems are likely to be intractable for an arbitrary sparse matrix, since simpler problems such as pebbling on a directed acyclic graph or minimizing the arithmetic operations in a sparse factorization are intractable. However, as in other sparse matrix computations, we can bound these quantities for problems with useful sparsity: i.e., where the underlying computational graphs have good separators. Hence we characterize these quantities for several common situations corresponding to choices of algorithms, orderings, blocking, and problems. We provide analytical results for model problems and simulation results for more irregular problems.

This research is motivated by the need to solve large-scale linear systems whose sizes are larger than the memory available on sequential or parallel computers. A key factor determining the execution time of the computation is the data traffic across the common storage hierarchy on current computers. For more information about the current interest in external memory algorithms we direct the reader to [1, 12].

Given a sparse linear system $Ax = b$ in which the coefficient matrix $A$ is symmetric positive definite, the Cholesky factorization decomposes $A$ into a product $A = LL^T$, where the Cholesky factor $L$ is lower triangular. A permutation matrix is generally included in this equation as well because the columns and rows of the coefficient matrix must be swapped in order to preserve sparsity in the factor. In this paper we consider that $A$ is already processed by a sparsity preserving algorithm.

We briefly discuss various sparse matrix factorization concepts such as filled graphs, elimination trees, and supernodes; and factorization algorithms such as left-looking, right-looking, and multifrontal algorithms in the next section.

At the beginning of the computation the entries of $A$ (the input) are stored on the disk. At the end of the computation the entries of $L$ (the output) must be stored on the disk as well. However, the computation can be performed only within the core, therefore data must move between the two storage layers.

We are interested in the relationship between the core size and the amount of data traffic between the disk and the core. Clearly, a large enough core would allow minimum traffic: reading the input at the beginning of the computation

and writing the output at the end of the computation. This basically corresponds to an in-core factorization in which the input and the output are stored on the disk. A smaller core determines an out-of-core factorization, in which computation and data movement are interleaved. These scenarios are explored in further detail in Section 3.

Earlier work on out-of-core sparse Cholesky factorization has been reported by Liu, Ashcraft, as well as by Rothberg and Schreiber. Various aspects of the out-of-core factorization with minimum traffic were investigated by Liu [8, 9, 10], who focused on left-looking and multifrontal algorithms. Given an elimination tree, he designed algorithms for computing tree traversals that minimize the core requirements. Ashcraft studied the same problem for the right-looking algorithm [2]. The experimental work by Rothberg and Schreiber [14] implemented an out-of-core multifrontal factorization for which the core is not large enough to allow minimum traffic. An early out-of-core multifrontal implementation is discussed in [3].

In the first two sections after this introduction we overview the three factorization algorithms and identify the major computational scenarios determined by the hierarchical nature of the storage. In the following two sections we present and discuss our results. We conclude in the last section.

# 2 Background

We begin by briefly discussing various sparse matrix factorization concepts such as filled graphs, elimination trees, and supernodes. Then we will consider three factorization algorithms: left-looking, right-looking, and multifrontal, in more detail.

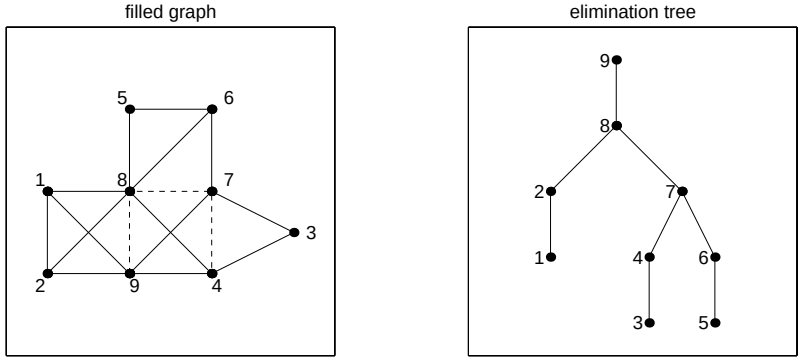


Figure 1: The graph representations of a sparse symmetric matrix $A$, its Cholesky factor $L$, and its elimination tree.

Graph models of sparse matrix factorizations are beneficial in designing efficient sparse algorithms. For the Cholesky factorization, there are two graphs of

interest: the *original graph*, $G(A)$, the adjacency graph of the symmetric matrix $A$; and the *filled graph*, $G(L + L^T)$, the adjacency graph of the factor $L$ and its transpose. Both these graphs are undirected since they represent symmetric matrices. A node in these graphs corresponds to a column (or row) of the matrix, and an edge corresponds to an off-diagonal nonzero in the matrix. Generally the Cholesky factor $L$ contains *fill* nonzeros, i.e., elements that have zero values in $A$ but become nonzero in $L$ during the factorization. Edges corresponding to such nonzeros in the factor are called fill edges.

In the example shown in Figure 1, the solid edges correspond to the edges in $G(A)$; such edges also belong to the filled graph $G(L + L^T)$, under a non-degeneracy assumption on the numerical values of nonzeros in $A$. Additionally the broken edges correspond to fill edges belonging to $G(L + L^T)$. Without loss in generality, for convenience, we assume that the graph $G(A)$ is connected.

A data structure that plays a central role in sparse Cholesky factorization is the *elimination tree* [11]; this tree is also shown in Figure 1. The elimination tree is the transitive reduction of the filled graph (i.e., direct each edge from its lower to its higher numbered endpoint; then remove every edge $(i, j)$ for which there exists a directed path in the filled graph that joins nodes $i$ and $j$). The *parent* of each node in the elimination tree is its lowest numbered higher neighbor in the filled graph.

The elimination tree captures the minimal dependence relationships between the columns in the factorization. An important property of the elimination tree is that if $(i, j)$ is an edge in the filled graph where $i < j$, then node $i$ must be a descendant of node $j$ in the elimination tree. Hence if $k$ and $l$ are nodes such that the subtrees rooted at $k$ and $l$ are node-disjoint, no edge in the filled graph can join any node in one subtree to any node in the other subtree. Hence these subtrees can be computed independent of each other (e.g., in parallel on different processors, or by accessing different files in an external memory algorithm).

Consequently, if node $j$ is an ancestor of a node $i$ in the tree, then the numerical values in column $i$ influence those in column $j$. Furthermore, the only columns that $i$ influences are its ancestors in the elimination tree. Hence all modern implementations of sparse Cholesky factorization compute the elimination tree and then employ it to schedule the numerical computations.

Another important concept is that of a *supernode*. Each supernode consists of columns that correspond to consecutive vertices in the elimination tree that (1) form a path in the tree (they do not have branches except at the first and the last vertex in the path), and (2) have identical subdiagonal nonzero structures. In the filled graph, vertices in a supernode have the same higher numbered neighbors.

In practice, numerical factorization algorithms are guided by the supernodal elimination tree, a more efficient representation. The vertices of a supernodal elimination tree are the supernodes. Further details on sparse matrix concepts might be found in [11, 13] and in the papers listed in [4].

The Cholesky factorization $A = LL^T$ of a symmetric positive definite matrix $A$ may be viewed as a computational procedure that turns each column of $A$ into a column of $L$. We will denote column indices by $j$ and $k$ and factor columns

by $L_j$ and $L_k$.

Two basic types of operations are performed by the algorithms we consider: FACTOR, which scales a column by a scalar, and UPDATE, which modifies a column by the multiple of another column. There are $n$ FACTOR operations, one for each column, where $n$ is the order of $A$ and $L$. These operations must be performed in a topological ordering of the elimination tree. There is an UPDATE$(L_j, L_k)$ operation (where column $j$ of $L$ updates column $k$) for each subdiagonal nonzero $L_{k,j}$ of the factor $L$. The operation UPDATE$(L_j, L_k)$ must be performed after operation FACTOR$(L_j)$ and before operation FACTOR$(L_k)$. We will denote by UPDATE$(*, L_k)$ an update operation to column $k$, and by UPDATE$(L_j, *)$ an update operation from column $j$. Each of these sets of operations could be performed in any order.

The order between the FACTOR operations is therefore fixed under a topological ordering of the elimination tree, while there is additional flexibility in choosing the order between FACTOR and UPDATE operations. There is a different factorization algorithm for each particular order, with left-looking and right-looking algorithms at the extremes of the range. Left-looking factorization corresponds to a lazy update algorithm in which all the UPDATE$(*, L_k)$ operations (to column $k$) are performed immediately before the FACTOR$(L_k)$ operation. Right-looking factorization corresponds to an eager update algorithm in which all the UPDATE$(L_j, *)$ operations (from column $j$) are performed immediately after the FACTOR$(L_j)$ operation.

The left-looking and right-looking algorithms are shown in Figure 2 (with the implicit assumption that $L$ is already initialized as $A$). The two algorithms take advantage of the nonzero structure of $L$, described by the two sets,

$$row[k] = \{j | j \leq k, L_{k,j} \neq 0\}, \quad \text{and} \quad col[j] = \{k | k \geq j, L_{k,j} \neq 0\}.$$

```
for k := 1 to n begin
    for j in row[k] \ {k}
        UPDATE(L_j, L_k);
    FACTOR(L_k);
end
```

```
for j := 1 to n begin
    FACTOR(L_j);
    for k in col[j] \ {j}
        UPDATE(L_j, L_k);
end
```

Figure 2: Left-looking and right-looking factorization.

The multifrontal algorithm is yet different from the left- and right-looking algorithms in that the UPDATE operations are not performed directly between factor columns, when the pair of columns involved is not adjacent in the elimination tree. Instead, updates from a descendant column to an ancestor column in the elimination tree are carried through a chain of temporary columns at each intermediate column on the elimination tree path. In Figure 3 we show a simple multifrontal algorithm, which is guided by the elimination tree. The *children* set is the set of children of a node in the elimination tree.

```
for j := 1 to n begin
    for k in col[j] \ {j}
        CLEAR(T_k^j);
    for i in children[j] begin
        ASSEMBLE(T_j^i, L_j);
        for k in col[i] \ {i, j} begin
            ASSEMBLE(T_k^i, T_k^j);
        end
    end
    FACTOR(L_j);
    for k in col[j] \ {j}
        UPDATE(L_j, T_k^j);
end
```

Figure 3: Multifrontal factorization.

The main characteristic of the multifrontal factorization is the use of temporary data. A temporary column $T_k^j$ is created for operation UPDATE$(L_j, T_k^j)$ if $j$ is not the child of $k$ in the elimination tree, and a CLEAR operation is required to initialize each temporary column with zero entries. If column $j$ is a child of $k$, the UPDATE$(L_j, L_k)$ operation is performed directly, just as in the other two algorithms. The effect of an UPDATE operation is propagated through the ASSEMBLE operations by a chain of temporary columns until it reaches the destination.

It is useful to visualize the data access patterns of the three factorization algorithms. Consider the elimination tree from Figure 4, replicated for each algorithm, and focus on the currently processed node, which is highlighted. If the factorization is left-looking then a subset of the nodes in the subtree rooted at the current node are accessed. If the factorization is right-looking, a subset of the nodes on the path from the current node to the root are accessed. If the factorization is multifrontal, only the current node and its children are accessed.

# 3 Out-of-Core Factorization

Several scenarios are possible in the context of a two-layer storage system, depending on $M$, the size of the core memory. The scenarios are shown in Figure 5, where the horizontal axis corresponds to the core size. The values $M_1$, $M_2$ and $M_3$, which determine the boundaries between the various computational scenarios, depend on the factor $L$ and on the factorization algorithm.

Note that we measure everything in entries, whether an entry corresponds to a single-, double-, or extended-precision number. For simplicity, we assume that the core stores only numerical entries, although it actually needs to store other data as well (such as the integer storage associated with the nonzero structure of the factor, the elimination tree, and other data structures).

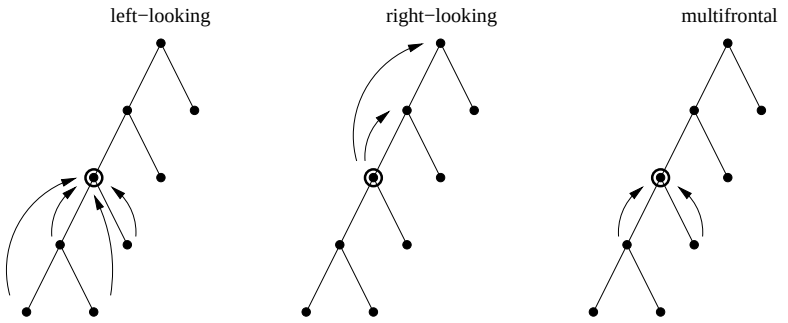


Figure 4: Data access patterns for sparse left-looking, right-looking and multifrontal factorization (the currently processed node is highlighted).

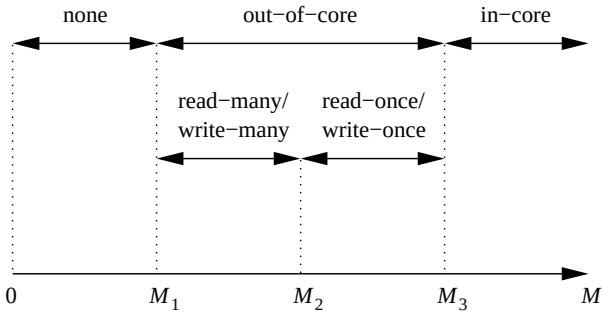


Figure 5: Factorization scenarios

For $0 \leq M < M_1$ the core is too small and the factorization cannot be performed. If the unit of data movement is the column, then $M_1$ is approximatively equal to $2|col[j_m]|$, where $j_m$ is the column with the largest number of nonzero entries. If the unit of data movement is the entry, then $M_1 = 3$, since only three elements need to be stored in memory to compute a single operation in the factorization in this case.

For $M \geq M_3$, the factorization can be performed in-core. The core is large enough to allow minimum traffic (reading the input, writing the output) without interleaving computation and data movement. For left-looking and right-looking factorization $M_3 = |L|$. (We denote by $|L|$ the number of nonzeros in $L$.) For multifrontal factorization $M_3 = |L| + |T|$, where $|T|$ represents the maximum number of temporary entries needed at any step of the factorization. The minimum traffic in any factorization is $|A| + |L|$; if $L$ is initialized with the values in $A$, and only the data structure for $L$ is accessed, then it is $2|L|$.

The interesting case is $M_1 \leq M < M_3$, when computation and data movement need to be interleaved, for an out-of-core factorization. Two major scenarios can be identified within this range. The first scenario is read-once/write-

once, denoted as R1/W1, and corresponds to $M_2 \leq M < M_3$. In this case the core is still large enough to allow minimum traffic if data movement is interleaved with computation. The second scenario is read-many/write-many, denoted as RM/WM, and corresponds to $M_1 \leq M < M_2$. In this case the core too small to allow minimum traffic.

In the remainder of this paper, we characterize the value of $M_2$, the minimum core memory size that permits R1/W1 factorization, theoretically and through simulations for model and representative problems from applications modeled by discretized partial differential equations and linear programs. We will show that $M_2$ is much smaller than $|L|$ for many problems. We also characterize the traffic as a function of the memory available in RM/WM factorization through analysis and simulations.

# 4 Minimum Traffic

Out-of-core factorization with the minimum traffic is possible as long as the core size is within the R1/W1 range. Figure 6 depicts a left-looking R1/W1 algorithm, as an example of this class of algorithms. Note the additional READ and WRITE functions, which perform the data movement, as well as the ALLOCATE function, which performs the core allocation. There is also a REORGANIZE function, called when there is not enough core left. The REORGANIZE function looks for entries that can be dropped from the columns that are already within the core (these correspond to entries in rows above the diagonal element of the column being computed) and then shifts the remaining entries. If there is still not enough core after calling the REORGANIZE function then the factorization aborts by calling the ERROR function.

For each problem there is a minimum core size $M_2$ that allows minimal traffic. The value $M_2$ depends on the factorization algorithm. We call it the left-looking, right-looking, or multifrontal core. We determined the complexity of $M_2$, which we refer to as the core complexity, for each one of the three factorization algorithms and for several model problems. We also implemented algorithms that compute the exact minimum core size for the three factorization algorithms for arbitrary problems.

We show the theoretical results first. We considered three factors in the analysis: branching and balance in the elimination tree, and connectivity in the filled graph.

We begin with model problems that range over extremes of branching and connectivity, discussing only balanced trees at first. For branching, the worst case corresponds to a path, while the best case corresponds to a star. Between these two extremes there are $p$-ary trees, where $p$ is a whole number larger than one. For connectivity, each descendant-ancestor pair in the elimination tree is connected in the filled graph in the worst case, while in the best case the filled graph is the same as the elimination tree.

We studied paths, $p$-ary trees and stars, with worst and best connectivity, then trees that correspond to 2-d and 3-d grids ordered by nested dissection.

```
for k := 1 to n begin
    if not enough core for L_k
        REORGANIZE();
    if not enough core for L_k
        ERROR();
    ALLOCATE(L_k);
    READ(L_k);
    for j in row[k] \ {k}
        UPDATE(L_j, L_k);
    FACTOR(L_k);
    WRITE(L_k);
end
```

Figure 6: R1/W1 left-looking factorization.

These five types of trees are shown in Figure 7. Note that the 2-d and 3-d elimination trees have both path and binary sections, and that the path sections are fully connected in the filled graph.

Table 1 shows the core complexity for the five types of trees. The complexity of the factor is also shown. The results are obtained by solving recurrence equations and computing appropriate sums that are determined by the elimination tree and the connectivity.

| branch. | connect. | factor | left-looking core | right-looking core | multifrontal core |
|---|---|---|---|---|---|
| path | best | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| | worst | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| $p$-ary | best | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ | $\Theta(\log n)$ |
| | worst | $\Theta(n \log n)$ | $\Theta(n)$ | $\Theta((\log n)^2)$ | $\Theta((\log n)^3)$ |
| star | | $\Theta(n)$ | $\Theta(n)$ | $\Theta(1)$ | $\Theta(n)$ |
| 2-d | | $\Theta(n \log n)$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| 3-d | | $\Theta(n^{4/3})$ | $\Theta(n^{4/3})$ | $\Theta(n^{4/3})$ | $\Theta(n^{4/3})$ |

Table 1: The core complexity for several balanced elimination trees.

As an example, consider right-looking factorization on a binary elimination tree (the result immediately generalizes to $p$-ary trees). For best connectivity the core must store at least three entries. For worst connectivity we need to sum all the entries along a leaf-to-root path, with $h = \log(n+1)$ being the tree height:

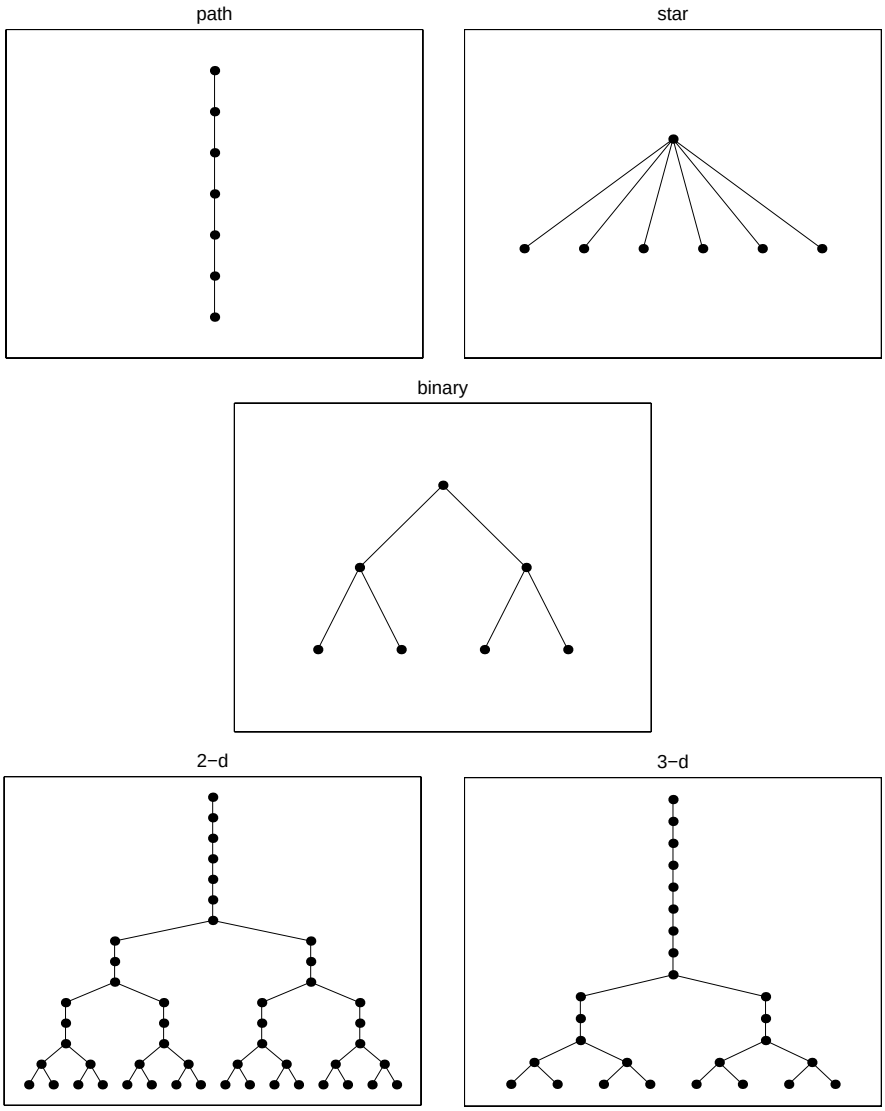


Figure 7: Balanced elimination trees.

$$M_2 = \sum_{i=1}^{h} i = \frac{h(h+1)}{2} = \frac{\log(n+1)[\log(n+1)+1]}{2} = \Theta((\log n)^2).$$

We omit the details of how the other results in Table 1 are obtained due to space considerations. We make three observations about these results.

First, branching favors right-looking factorization over left-looking factorization. In Table 1, there is only one case in which the left-looking core is smaller than the right-looking core, although the asymptotic complexity is the same. This happens when the elimination tree is a path, and the filled graph is fully connected, corresponding to dense factorization. The left-looking core is smaller than the right-looking core by a factor of two. There is also no asymptotic difference between the two algorithms for the 2-d and 3-d model problems, although the right-looking core is smaller than the left-looking core by a constant factor this time.

Second, branching helps reduce the size of the core with respect to the factor. In Table 1, the only case in which the core is as large as the factor itself is again when the elimination tree is a path and the filled graph is fully connected. As we branch, the core can become asymptotically smaller than the factor, even if the filled graph is fully connected. Note that such an asymptotic difference exists for 2-d model problems but not for 3-d model problems. This is related to the longer path sections between the branches in the latter elimination tree.

Third, multifrontal factorization tends to be favored by branching, although not as much as the right-looking factorization. Interestingly, the multifrontal algorithm can perform well, as in several cases in Table 1, but it can also be a poor choice. In order to show why we need to consider two more cases.

The first case is a generalized star. Assume that $q$ nodes form a clique in the filled graph and that each one of the remaining $n - q$ nodes is a leaf in the elimination tree, and is connected to every node in the clique. Then, we have

$$\begin{aligned} |L| &= \frac{q(q+1)}{2} + (n-q)(q+1) \\ |T| &= (n-q) \cdot \frac{q(q+1)}{2}. \end{aligned}$$

Choosing $q = \sqrt{n}$, for example, we find that $|L| = \Theta(n^{3/2})$, and $|T| = \Theta(n^2)$; therefore the amount of temporary data required by the multifrontal algorithm is asymptotically larger than the factor itself.

The second case is determined by unbalancing the elimination tree. Remember that we have only discussed balanced trees so far (the best case in terms of balance). Consider now binary trees with worst case balance, as shown in Figure 8.

For multifrontal factorization, the tree on the left corresponds to the best case because we need to store temporary data for at most two nodes, while

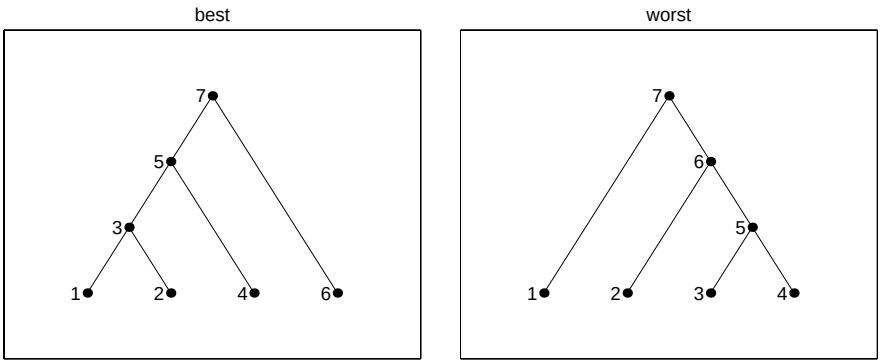


Figure 8: Unbalanced binary trees.

the tree on the right corresponds to the worst case because we need to store temporary data for $n$ nodes.

Table 2 shows the core complexity for the two unbalanced trees. Again, the complexity of the factor is also provided. In the best case the complexity of the multifrontal core is the same as the complexity of the left-looking and right-looking core. However, in the worst case the multifrontal core is asymptotically larger. In addition, the core can be asymptotically larger than the factor itself. We note that the core complexity of the multifrontal algorithm with an unbalanced elimination tree can be reduced by renumbering the children of each node in the elimination tree using an algorithm designed by Liu [9].

| balance | connect. | factor | left/right-looking core | multifrontal core |
|---|---|---|---|---|
| best | best | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| | worst | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| worst | best | $\Theta(n)$ | $\Theta(1)$ | $\Theta(n)$ |
| | worst | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^3)$ |

Table 2: The core complexity for some unbalanced elimination trees.

We turn now to the experimental results. As we mentioned, we have implemented simulation algorithms that compute the minimum size of the core for less regular problems.

We determined the minimum core size for various problems, but have selected three for this discussion: a $1023 \times 1023$ 2-d grid, a $63 \times 63 \times 63$ 3-d grid, and a linear programming problem that comes from multicommodity flow in a network called `ken13`. The last problem is not large but it represents a good example in which the multifrontal factorization is a poor choice.

We show the results in Figure 9. Our main choice for ordering is the node

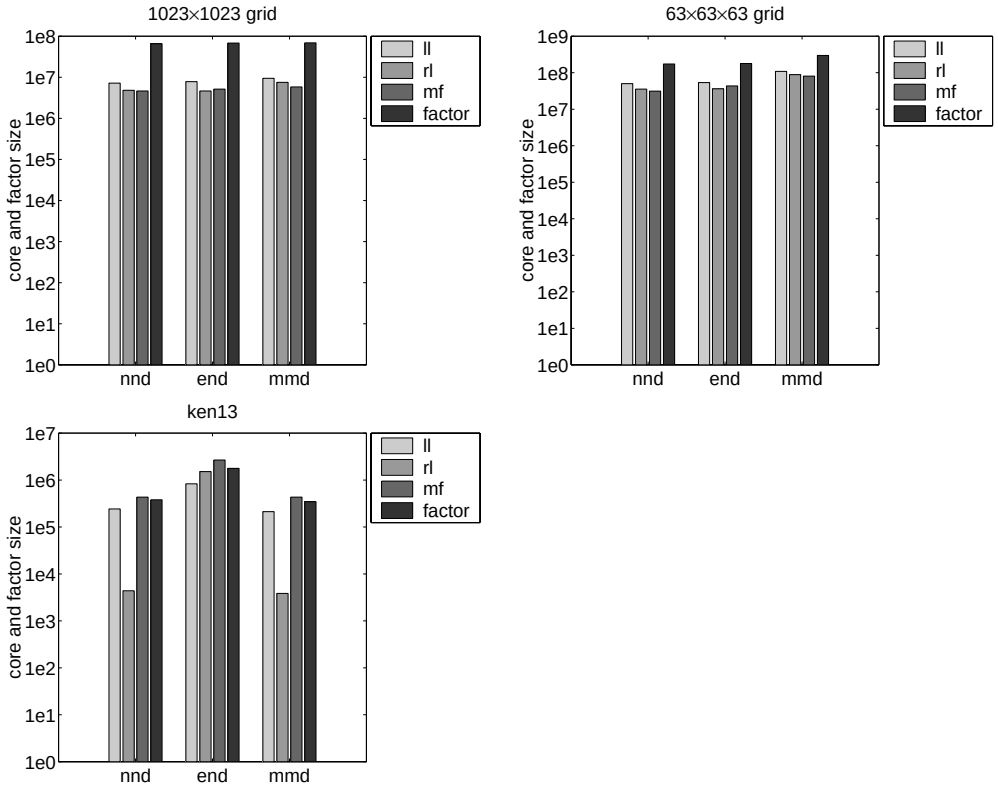


Figure 9: The minimum core size for a 2-d grid, a 3-d grid, and `ken13`, ordered by node nested dissection, edge nested dissection, and multiple minimum degree.

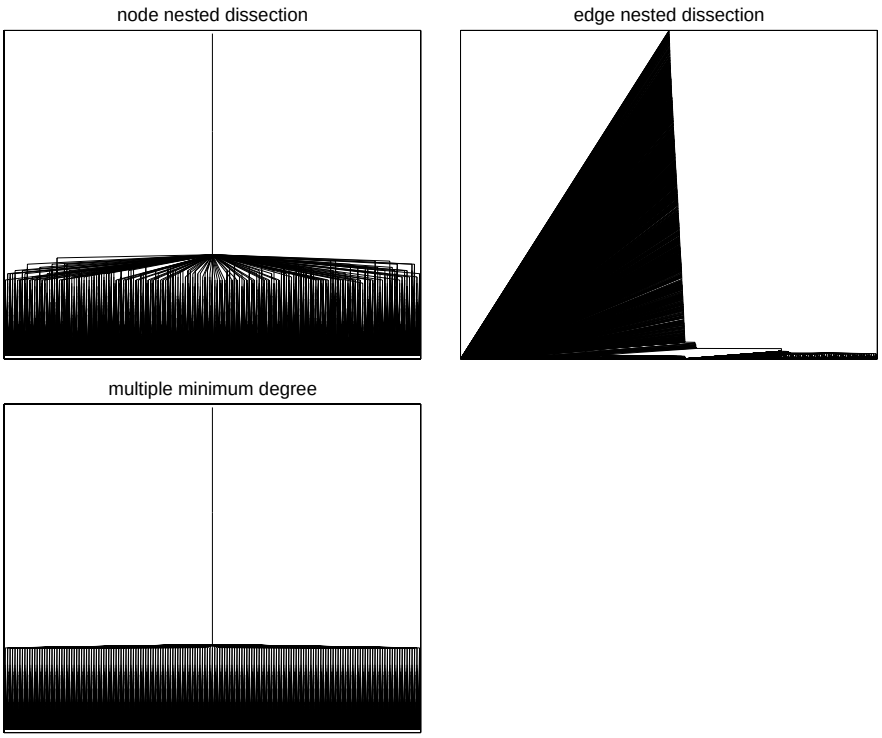


Figure 10: The elimination trees for `ken13`.

nested dissection algorithm from Metis [6], but we have also included results obtained with the edge nested dissection algorithm, also from Metis, as well as results obtained with the multiple minimum degree (MMD) algorithm [7].

For each one of the three problems we plot the left-looking, right-looking and multifrontal core, and the size of the factor $|L|$, in order to visualize the gap between $M_2$ and $M_3$. We also plot $|L|$ for each problem. The $y$-axis is plotted on a logarithmic scale.

Focusing on the two grids, note the difference between $M_2$ and $|L|$, which indicates that the factorization can be performed with minimal traffic using a core that is significantly smaller than the factor (a factor of ten for the 2-d grid, and a factor of four for the 3-d grid). Note also that the left-looking core tends to be larger than the right-looking and the multifrontal core.

The two grid examples are representative of a large number of problems from discretizations of partial differential equations. We determined the minimum core size for problems from various application areas like structural mechanics, electromagnetics, acoustics. We witnessed the same trend for these problems: the minimum core size is significantly smaller than the factor, with the right-looking and multifrontal algorithms performing better than the left-looking algorithm.

These trends are not general though, as our analysis shows. For the linear program `ken13`, the behavior is completely different. A first observation is that the multifrontal core is always larger than the factor, and hence the multifrontal factorization is a bad choice for this particular problem. A second observation is that for node nested dissection and multiple minimum degree the right-looking core is extremely small, which makes the right-looking algorithm the best choice in this case. This is not true for edge nested dissection, which is a bad ordering for `ken13`, due to the large size of the factor.

The results we obtained for `ken13` can be correlated quite well with our theoretical study. Looking at the elimination trees for this problem, shown in Figure 10, the trees that correspond to node nested dissection and multiple minimum degree are very similar to the generalized star. We already know that multifrontal factorization should be avoided in this case because it requires more storage than $|L|$. We also know that right-looking factorization is a much better choice than left-looking factorization in this situation. The tree that corresponds to edge nested dissection is also familiar to us; it is an unbalanced tree and it is unbalanced in the wrong direction from the multifrontal factorization perspective. Again, the multifrontal factorization should be avoided since it requires more storage than $|L|$.

# 5 Larger Traffic

As soon as the size of the core drops below $M_2$, minimum traffic is no longer possible, since the same entry (factor or temporary) may be read or written more than once. We are interested in reducing the traffic for a given core size, and the key to doing this is data reuse.

In linear algebra terms, the factorization is a level 3 operation and thus has a significant potential for data reuse. Table 3 lists the number of arithmetic operations and the number of data accesses for the sparsest (a diagonal matrix) and the densest factorizations (a dense matrix), as well as for the factorization of the 2-d and 3-d problems ordered by nested dissection. The ratio between the number of arithmetic operations and the number of data accesses, which indicates the potential for data reuse, is also listed in Table 3, showing that the denser the factorization, the larger the potential for data reuse.

| problem | work | data | ratio |
|---|---|---|---|
| sparsest | $\Theta(n)$ | $\Theta(n)$ | $\Theta(1)$ |
| 2-d | $\Theta(n^{3/2})$ | $\Theta(n \log n)$ | $\Theta(n^{1/2}/\log n)$ |
| 3-d | $\Theta(n^2)$ | $\Theta(n^{4/3})$ | $\Theta(n^{2/3})$ |
| densest | $\Theta(n^3)$ | $\Theta(n^2)$ | $\Theta(n)$ |

Table 3: The complexity of the arithmetic work and data accesses for factorization, for various problems.

The technique of choice for data reuse is blocking. For sparse matrices, the blocks must be defined within supernodes, where data can be reused.

Two major blocking alternatives are 1-d (along columns) and 2-d (along both columns and rows). Figure 11 depicts a RM/WM left-looking factorization algorithm that uses 1-d blocks, for example. The loops iterate over block columns this time. Column indices such as $j$ and $k$ are replaced by the block column indices $J$ and $K$ and the block nonzero structure of $L$ is described by analogous *Row* and *Col* sets. The DISCARD function is used to remove data from the core.

```
for K := 1 to N begin
    READ(L_K);
    for J in Row[K] \ {K} begin
        READ(L_J);
        UPDATE(L_J, L_K);
        DISCARD(L_J);
    end
    FACTOR(L_K);
    WRITE(L_K);
    DISCARD(L_K);
end
```

Figure 11: RM/WM left-looking factorization.

We determined the complexity of the traffic for the three factorization algorithms, for $M_1 \leq M < M_2$. We choose $M_1$ to be $2|col[j]|$, where $j$ is the column with the largest number of nonzero entries, because this value is valid for both

1- and 2-d blocks. This time we focused on the 2-d and 3-d problems ordered through nested dissection, since these are the only sparse problems which have a significant potential for data reuse in Table 1. We also implemented simulation algorithms that compute the exact amount of traffic for arbitrary problems.

| problem | without blocking | with 2-d blocking | with 1-d blocking |
|---|---|---|---|
| 2-d | $\Theta(n^{3/2})$ | $\Theta(n^{3/2}/\sqrt{M})$ | $\Theta(n^2/M)$ |
| 3-d | $\Theta(n^2)$ | $\Theta(n^2/\sqrt{M})$ | $\Theta(n^{8/3}/M)$ |

Table 4: Traffic complexity.

We begin again with the theoretical results. The analysis is highly idealized, as we consider that only the data accessed at any step is brought into the core. This is possible only with fine granularity data movement. Because of the algorithmic and software overhead of extracting only the data that needs to be accessed, data must move at coarse granularity in practice, which causes the traffic to be larger.

In order to perform the analysis it is useful to focus on multifrontal factorization first. A dense frontal matrix corresponds to each supernode and the traffic that corresponds to the factorization of a dense matrix can be easily computed. Consider a dense matrix of order $n$ that is sufficiently large with respect to $M$. Then the traffic is $\Theta(n^3)$ without blocking, $\Theta(n^4/M)$ for 1-d blocks, and $\Theta(n^3/\sqrt{M})$ for 2-d blocks. Asymptotically, the 2-d block expression is optimal for standard factorization algorithms. This is similar to the standard multiplication of two dense matrices of order $n$, $\Theta(n^3/\sqrt{M})$ traffic being optimal in this case as well, as shown by Hong and Kung [5], and more recently by Toledo [16]. The dense factorization traffic is also discussed by Toledo [15].

The structure of the 2-d and 3-d problems ordered through nested dissection allows us to describe through a recurrence equation the traffic for the multifrontal factorization. At each level of recursion we have the traffic that corresponds to the factorization of a frontal matrix plus the traffic that corresponds to the level below.

The recursive equations that describe the traffic for 2-d problems, without blocking, with 1-d blocks, and with 2-d blocks, are, respectively,

$$\begin{aligned} T(k) &= 4T(k/2) + \Theta(k^3), \\ T(k) &= 4T(k/2) + \Theta(k^3/\sqrt{M}), \\ T(k) &= 4T(k/2) + \Theta(k^4/M), \end{aligned}$$

where $k$ is the grid size ($k^2 = n$).

The solutions of the recurrences are presented in Table 4. The results extend to left-looking and right-looking factorization as well. It is easy to see that, asymptotically, 2-d blocks determine less traffic than 1-d blocks.

Turning to the experimental results, we now discuss the actual traffic for the three factorization algorithms with 1-d and 2-d blocks. We selected the

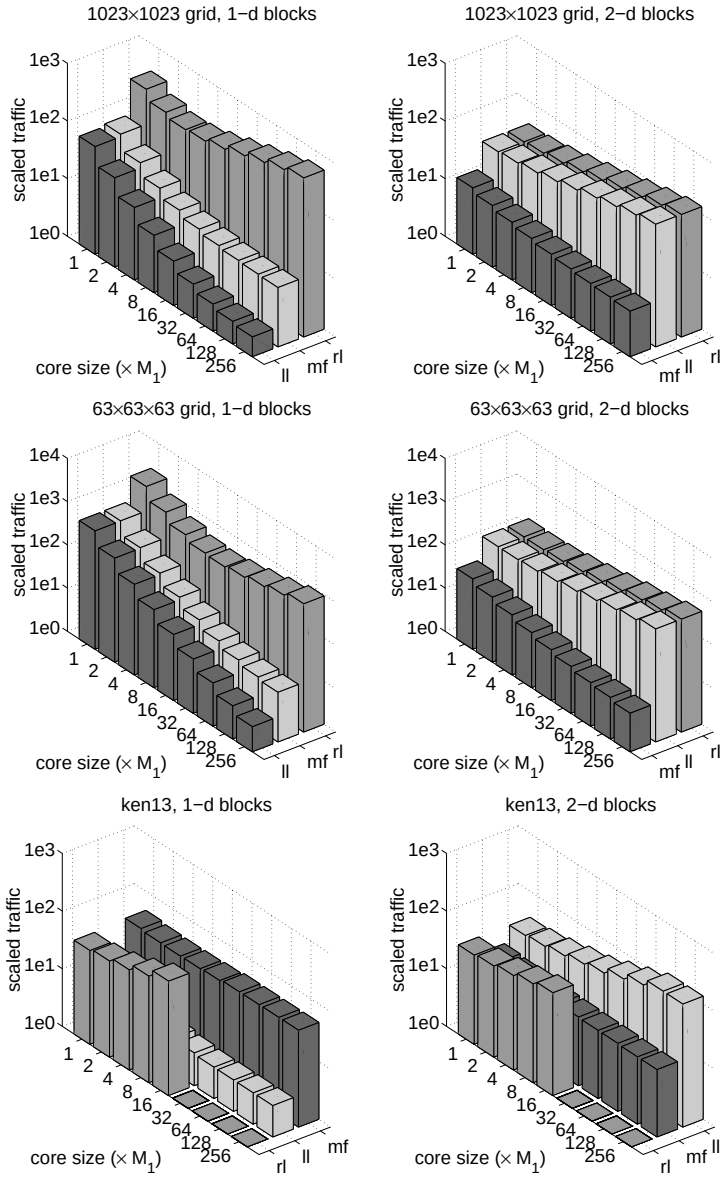


Figure 12: The traffic for a 2-d grid, a 3-d grid, and `ken13`, all ordered by node nested dissection.

same three problems from the previous section as examples and we show results obtained with the node nested dissection ordering from Metis. The results are presented in Figure 12. For each problem, the 1-d traffic is shown on the left and the 2-d traffic is shown on the right. We choose the core size $M$ to range from $M_1$ to $256 \times M_1$. We normalize the traffic with respect to $2|L|$, which represents the minimum traffic. Note that the traffic and memory sizes are plotted on logarithmic scales.

Focusing on the two grids first, note the difference between the left-looking and multifrontal traffic on the one hand and the right-looking traffic on the other hand, for 1-d blocks; also note the difference between the multifrontal traffic on the one hand and the left-looking and right-looking traffic on the other hand, for 2-d blocks. These differences are caused by the coarse granularity of data movement, which causes unnecessary traffic. This does not happen for left-looking factorization with 1-d blocks and it is not significant for multifrontal factorization no matter what the blocking is. This is why multifrontal factorization performs well with both type of blocks, while left-looking factorization performs well only with 1-d blocks. In these cases we see a low amount of traffic that decreases with the increase of $M$. In the other three cases the traffic is significantly larger and it increases with increasing core size $M$. This is counter-intuitive, but can be explained as follows. When larger memory is available, larger block sizes can be chosen, and in turn, the larger blocks can cause higher traffic by moving a larger amount of data that is not accessed.

For `ken13`, since the right-looking algorithm has a small value of $M_2$, as soon as $M$ becomes larger than $M_2$, traffic drops to the minimal value of $2|L|$. There is a significant difference between left-looking and multifrontal algorithms when 1-d blocks are used, the multifrontal traffic being larger. This is caused by the large amount of temporary data. We already know from the previous section that multifrontal factorization is not a good choice for this problem. On the other hand, due to the unnecessary data movement, it is the left-looking traffic that is larger when 2-d blocks are used.

# 6 Conclusions

A two-layer (disk/core) storage system determines several possible computational scenarios for the sparse Cholesky factorization. We identified two major out-of-core scenarios; the read-once/write-once (R1/W1) scenario in which we characterize the minimum core size that permits the minimum traffic; and the read-many/write-many (RM/WM) scenario, requiring a greater amount of traffic for smaller core sizes. In the latter case, we have characterized the traffic as a function of the core size. For both scenarios, we provide analytical results for model problems, and experimental results from simulation for irregular problems from computational partial differential equations and linear programming.

For the RM/WM scenario, the most common case in external memory factorizations for large-scale problems from discretized partial differential equations, our results show that multifrontal factorization with either 1- or 2-d blocking

or left-looking factorization with 1-d blocking are the best choices for an out-of-core direct solver. 2-d blocking has the advantage of asymptotically optimal traffic; however, the asymptotic behavior of 2-dimensional blocking manifests itself only for very large problems, and pivoting for numerical stability is easier to implement with 1-d blocking. The multifrontal factorization is more appealing in terms of an implementation because of its elegant computational pattern.

Yet, the multifrontal algorithm should not be used when the size of the temporary data that it creates is larger than the size of the factor. This situation occurs for some problems from linear programming and other application areas where there is no underlying geometrical mesh governing the computation, or for highly irregular geometries. In these problems, there is a small set of nodes whose removal disconnects the graph into several connected components. Then the core size required by the right-looking algorithm is sufficiently small that it can perform the computation in the R1/W1 scenario for relatively small core sizes, thus reducing the traffic to the minimum possible.

We have implemented fast simulation algorithms that compute the traffic in the RM/WM scenario given a factorization algorithm, an ordering, and a core size; simulation algorithms have also been implemented for computing the minimum core size in the R1/W1 scenario. Given a problem, the simulation algorithms can be used to decide which one of the ⟨ordering, algorithm, blocking⟩ triples would give the best results.

We have written an object-oriented direct solver software library called Oblio [4] that solves symmetric positive definite and indefinite systems of linear equations; we support both real- and complex-valued arithmetic. We plan to extend Oblio with out-of-core functionality, basing our algorithmic choices on the results that we have obtained in this paper. Our preliminary experiments with implicit-blocked and explicit-blocked data movement (the former with operating system support, the latter by managing files explicitly with our software) on an SGI Origin show that significant performance gains are obtained with explicit data movement. Consequently, we expect that a substantial effort will be needed to implement the external memory solver.

# References


[1] J. M. Abello and J. S. Vitter, editors. *External Memory Algorithms*. American Mathematical Society, 1999.

[2] C. Ashcraft. Personal comunication.

[3] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in sparse matrix methods for large linear systems on vector supercomputers. *Internat. J. Supercomput. Appl*, 1:10–30, 1987.

[4] F. Dobrian, G. K. Kumfert, and A. Pothen. The design of sparse direct solvers using object-oriented techniques. In H. P. Langtangen, A. M. Bruaset, and E. Quak, editors, *Advances in Software Tools in Scientific*

*Computing*, volume 50 of *Lecture Notes in Computational Science and Engineering*, pages 89–131. Springer-Verlag, 2000.

[5] J. W. Hong and H. T. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, 1981.

[6] G. Karypis and V. Kumar. METIS: Family of multilevel partitioning algorithms. http://www-users.cs.umn.edu/ karypis/metis.

[7] J. W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, June 1985.

[8] J. W. H. Liu. An application of generalized tree pebbling to sparse matrix factorization. Technical report, York University, April 1986.

[9] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12(3):249–264, September 1986.

[10] J. W. H. Liu. An adaptive general sparse out-of-core Cholesky factorization scheme. *SIAM Journal on Scientific and Statistical Computing*, 8(4):585–599, July 1987.

[11] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, January 1990.

[12] J. May. *Parallel I/O for High Performance Computers*. Morgan Kaufmann, 2000.

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

[14] E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, August 1999.

[15] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, October 1997.

[16] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 161–179. American Mathematical Society, 1999.