Winter 2001

External Memory Algorithms for Factoring Sparse Matrices

Florin Dobrian
Old Dominion University

EXTERNAL MEMORY ALGORITHMS FOR FACTORING SPARSE MATRICES

by

Florin Dobrian
B.S. June 1989, Bucharest Polytechnic Institute, Bucharest, Romania

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December 2001

Approved by:

Alex Pothen (Director)

Cleve Ashcraft (Member)

David Keyes (Member)

Stephan Olariu (Member)

Linda Stals (Member)

ABSTRACT

EXTERNAL MEMORY ALGORITHMS FOR FACTORING
SPARSE MATRICES

Florin Dobrian
Old Dominion University, 2001
Advisor: Dr. Alex Pothen

We consider the factorization of sparse symmetric matrices in the context of a two-layer storage system: disk/core. When the core is sufficiently large the factorization can be performed in-core. In this case we must read the input, compute, and write the output, in this sequence. On the other hand, when the core is not large enough, the factorization becomes out-of-core, which means that data movement and computation must be interleaved.

We identify two major out-of-core factorization scenarios: read-once/write-once (R1/W1) and read-many/write-many (RM/WM). The former requires minimum traffic, exactly as much as the in-core factorization: reading the input and writing the output. More traffic is required for the latter. We investigate three issues: the size of the core that determines the boundary between the two out-of-core scenarios, the in-core data structure reorganizations required by the R1/W1 factorization and the traffic required by the RM/WM factorization. We use three common factorization algorithms: left-looking, right-looking and multifrontal.

In the R1/W1 scenario, our results indicate that for problems with good separators, such as those coming from the discretization of partial differential equations, ordered with nested dissection, right-looking and multifrontal factorization perform slightly better than left-looking factorization. There are, however, applications for which multifrontal is a bad choice, requiring too much temporary storage. On the other hand, right-looking factorization should be avoided in the RM/WM scenario. Left-looking is a good choice, but only if data is blocked along one dimension. Multifrontal performs well for both one and two dimensional blocks as long as not too much storage is required.

We also explore a framework for a software implementation. We have implemented an in-core

solver that relies on some object-oriented constructs. Most of the code is written in C++, except for some kernels written in Fortran 77. We intend to add out-of-core functionality to the code and data movement is a major concern. Implicit data movement represents the easy way, but, as some of our experiments show, good performance can be achieved only with explicit data movement. This complicates the code and we expect a substantial effort in order to implement an efficient out-of-core solver.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER I
INTRODUCTION

We investigate the effect of the disk-core interface on the factorization of sparse symmetric matrices. This research is motivated by the need to solve large scale linear systems whose sizes are larger than the memory available on computers.

The factorization represents the decomposition of the coefficient matrix A from the linear system $Ax = b$. In its simplest form, the factorization equation for a symmetric matrix A is $A = LL^T$, where L is a lower triangular factor.

This computational procedure is generally straightforward, basically a triple nested loop. However, when A is large and sparse it is important to take advantage of its sparsity and avoid wasting computational resources by storing and operating only on nonzero entries. This significantly complicates the factorization. Further complications are determined by the hierarchical nature of the storage.

Assuming a finite core and an infinite disk, the entries of A (the input data) initially lie on the disk and the entries of L (the output data) must be stored on the disk as well. However, the computation can only take place in-core, therefore data must move between the two storage layers. Reading data from and writing data to the disk while computing determines an out-of-core factorization.

We identify two major out-of-core scenarios: read-once/write-once and read-many/write-many. In the former scenario the core is large enough and the data traffic is minimal, basically reading the input and writing the output. In the latter, the core is smaller and more data traffic is required.

For each scenario we consider three common column based factorization algorithms: left-looking, right-looking and multifrontal. Their different data access patterns determine significantly different behaviors in the out-of-core context.

The dissertation is organized as follows: in Chapter II we provide background information; Chapters III and IV are dedicated to the two out-of-core scenarios, while Chapter V is dedicated to an implementation framework; we conclude in Chapter VI.

The model journal used for this dissertation is *BIT*.

CHAPTER II
BACKGROUND

In this chapter we review the basic concepts required throughout the dissertation: the direct solution of sparse linear systems, sparse factorization algorithms, graphs as sparse matrix tools, supernodes, blocks. We also discuss computational scenarios that are determined by a hierarchical storage.

2.1 SOLVING SPARSE SYMMETRIC LINEAR SYSTEMS OF EQUATIONS

The problem of solving a linear system of equations is stated as follows: given a square nonsingular matrix A (the *coefficient matrix*) and a vector b (the *right-hand side*), compute the vector x (the *solution*) that satisfies

$$Ax = b. \tag{1}$$

The order of the system (the number of columns/rows of A) is denoted as n. In this dissertation we focus on symmetric systems, therefore A is symmetric as well.

There are two major approaches for solving linear systems [14, 55]: *direct* and *iterative*. Direct solvers are generally more robust, but they also require more storage for the computation.

A direct solver decomposes the original system into a sequence of simpler ones, which are easier to solve. The procedure has two steps: first, A is decomposed into a product of simpler matrices called *factors*; second, the systems formed with the factors as coefficient matrices are solved. These two steps are called the *factorization* and the *solve*, respectively.

When A is large and sparse we need to take advantage of its nonzero structure (the locations of the nonzero entries) and use computational resources judiciously. The problem is that the sparsity of A does not generally guarantee the sparsity of the factors and the factors tend to have significantly more nonzero entries that A. A third step is required in order to preserve sparsity. This permutes the rows/columns of A before the factorization and it is called *ordering*.

Like all numerical algorithms, the factorization raises the issue of numerical stability [14, 25, 42, 55], a stable factorization generally requiring pivoting (permuting the rows/columns during the

factorization). However, it is more convenient to analyze a factorization algorithm in the absence of pivoting. In this case the sparse factorization equation is

$$PAP^T = LL^T, \tag{2}$$

also known as the sparse Cholesky factorization [22]. Here P is a sparsity preserving permutation and L is the Cholesky factor, which is lower triangular (because of the shape of the factor the solve step is also known as the *triangular solve*).

The computation, described in Figure 1, proceeds as follows: first, a sparsity preserving permutation of the coefficient matrix is computed by an ordering algorithm; second, a factorization algorithm takes the reordered coefficient matrix and decomposes it into a product of factors; third, since the system can be equivalently rewritten as

$$PAP^T Px = Pb,$$

the solution is computed by the following sequence:

$$\begin{aligned} c &= Pb, \\ z &= L^{-1}c, \\ y &= L^{-T}z, \\ x &= P^T y. \end{aligned}$$

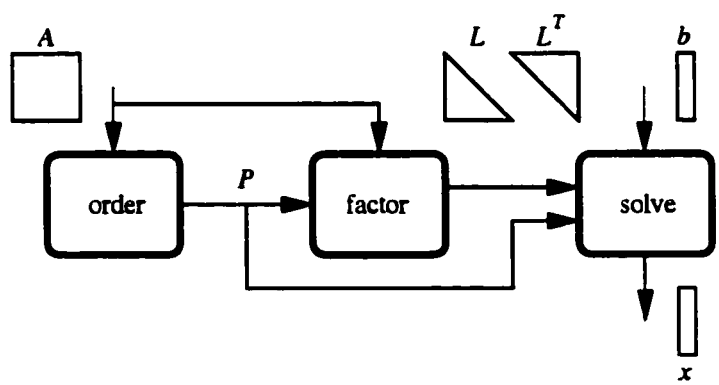


Figure 1: A black box representation of a sparse symmetric positive definite direct solver.

In order to isolate the factorization we can rewrite the Cholesky equation as

$$A = LL^T, \tag{3}$$

considering thus that A was already processed by some ordering algorithm that preserves sparsity.

The major storage requirements of the computation come from A and L, ranging between $\Theta(n)$ and $\Theta(n^2)$. In common situations A requires only $\Theta(n)$ storage but L requires significantly more.

Throughout the dissertation we use several examples of sparse systems, the two most common corresponding to model problems with two or three dimensions (2-d and 3-d). General 2-d and 3-d systems come from the discretization of partial differential equations over 2-d and 3-d domains. The geometry of these systems makes nested dissection ordering algorithms [30] asymptotically optimal and each class of systems can be analyzed using a model.

More precisely, the models we use correspond to 2-d regular grids with 9-point finite difference stencils and to 3-d regular grids with 27-point finite difference stencils, ordered by a geometrically perfect nested dissection algorithm. For the 2-d models the grids are k-by-k, thus $n = k^2$. Similarly, for the 3-d models the grids are k-by-k-by-k, thus $n = k^3$. In both cases $k = 2^l - 1$, where l is a positive integer. Figures 2 and 3 show the coefficient matrix and the factor for the 2-d model problem with $k = 7$, respectively. Figure 4 shows the coefficient matrix and the factor for the 3-d model problem with $k = 3$. Note that the nonzero structure of the lower triangle of A is included in the nonzero structure of L. This is an important property of the factorization (ignoring catastrophic cancellation) [22], the entries in the new locations being known as *fill entries* or simply as *fill* (the objective of the ordering step is to reduce the fill).

The storage requirements for L for 2-d and 3-d problems ordered by nested dissection are $\Theta(n \log n)$ and $\Theta(n^{4/3})$, respectively.

2.2 LEFT-LOOKING AND RIGHT-LOOKING FACTORIZATION

The factorization can be viewed as a triply nested loop, commonly organized along columns (the outer and the middle loop iterate on columns, the inner loop iterates on rows). The basic idea is to turn each column of A into a column of L, beginning with the leftmost column and ending with

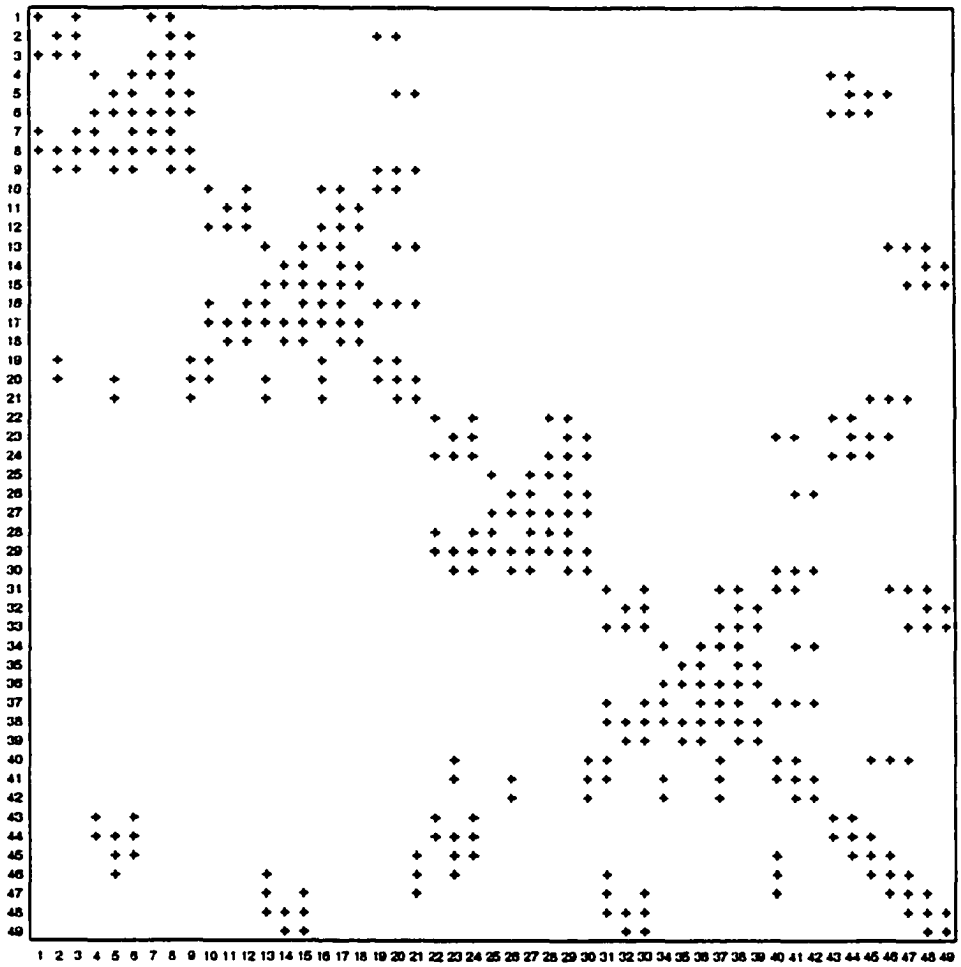


Figure 2: The coefficient matrix for a 2-d model problem ($k = 7$).

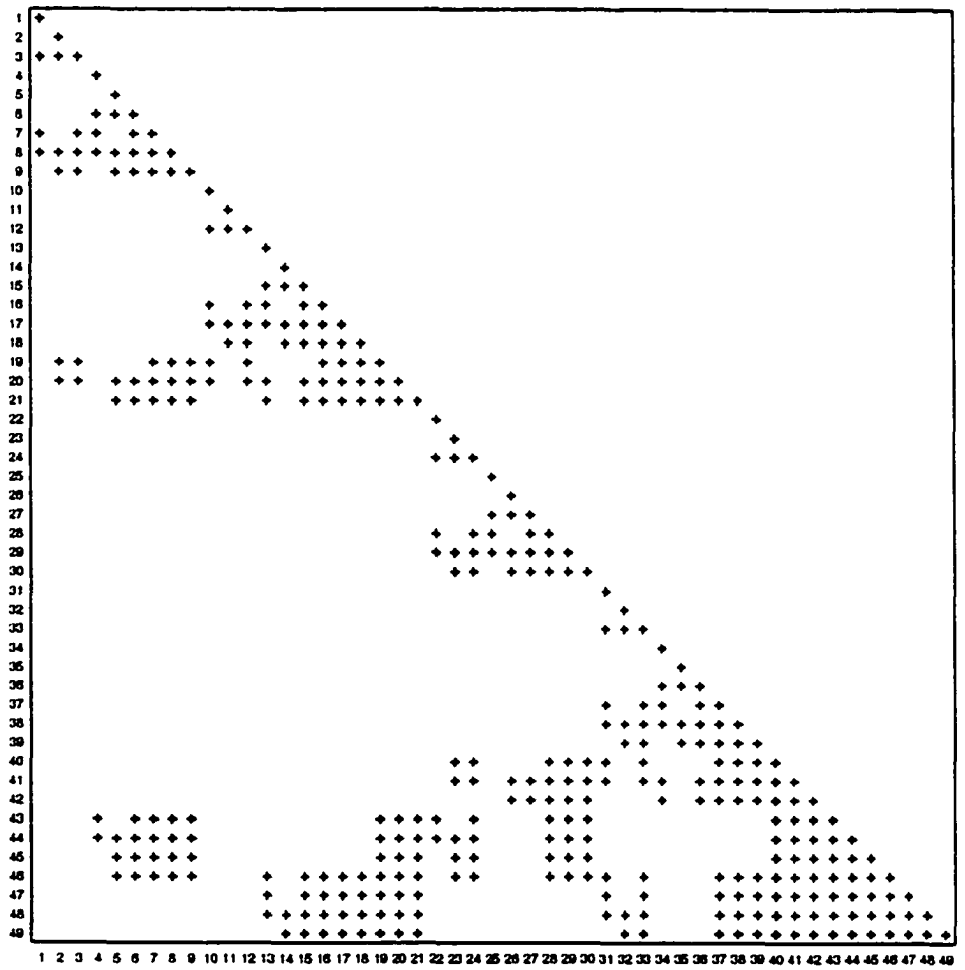


Figure 3: The factor for a 2-d model problem ($k = 7$).

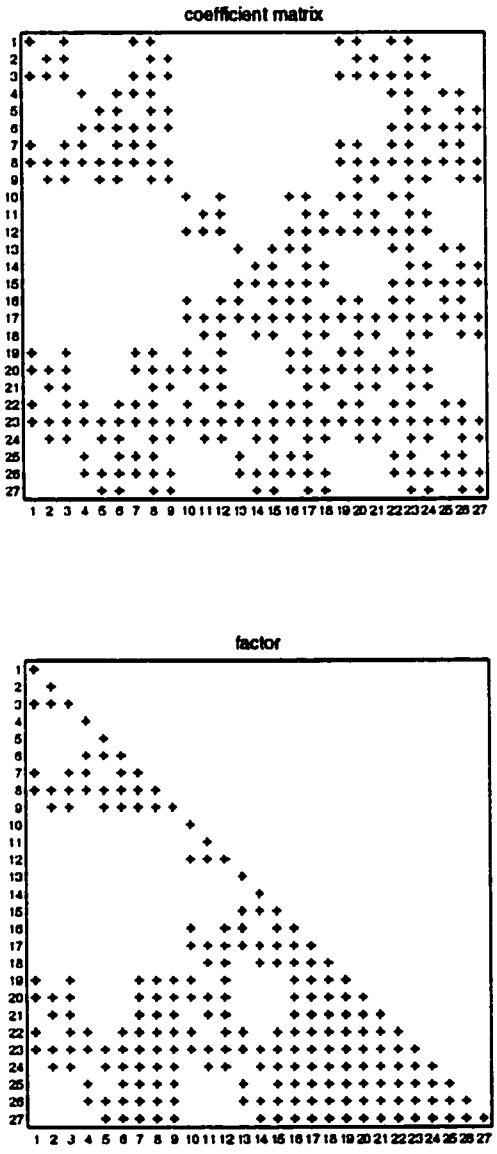


Figure 4: The coefficient matrix and the factor for a 3-d model problem ($k = 3$).

the rightmost one. Because of the symmetry, we refer to $A_{j:n,j}$ as column j of A. Column j of L is $L_{j:n,j}$.

By switching the outer loop and the middle loop of the factorization we obtain two common algorithms: *left-looking* and *right-looking*. In order to write these algorithms we need to expand equation (3). Column k of A is equal to the product between L and column k of L^T, as shown in Figure 5. We thus have

$$A_{k:n,k} = \sum_{j=1}^{k} L_{k:n,k} L_{k,j}. \tag{4}$$

Rewriting equation (4) in order to compute column k of L we obtain

$$L_{k:n,k} = A_{k:n,k} - \sum_{j=1}^{k-1} L_{k:n,k} L_{k,j}. \tag{5}$$

Equation (5) immediately leads us to Algorithm 1, which is the left-looking factorization. The name of the algorithm is determined by the fact that, while processing column k, we look back at columns 1 through $k-1$, as shown on the left side of Figure 6.

```
1.  for k := 1 to n begin
2.      for j := 1 to k − 1
3.          for i := k to n
4.              A_{i,k} := A_{i,k} − L_{i,j} L_{k,j};
5.      L_{k,k} := √A_{k,k};
6.      for i := k + 1 to n
7.          L_{i,k} := A_{i,k} / L_{k,k};
8.  end
```

Algorithm 1: General left-looking factorization.

Algorithm 2 is the right-looking factorization, obtained by switching the outer loop and the middle loop of the left-looking factorization. Similarly, the name of the algorithm is determined by the fact that, while processing column j, we look forward at columns $j+1$ through n, as shown on the right side of Figure 6.

Note the assumption made for the two algorithms: A and L use separate storage and the contents of A are destroyed after the execution. In practice none of these is necessary: A and L may use

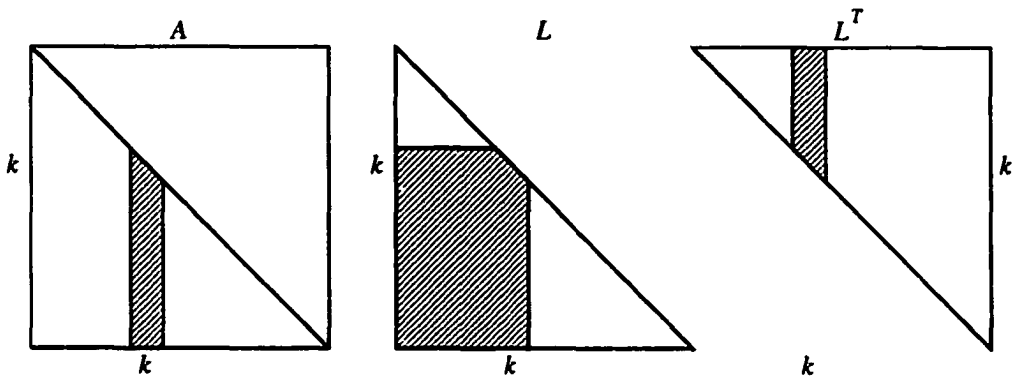


Figure 5: The computation of column k of A.

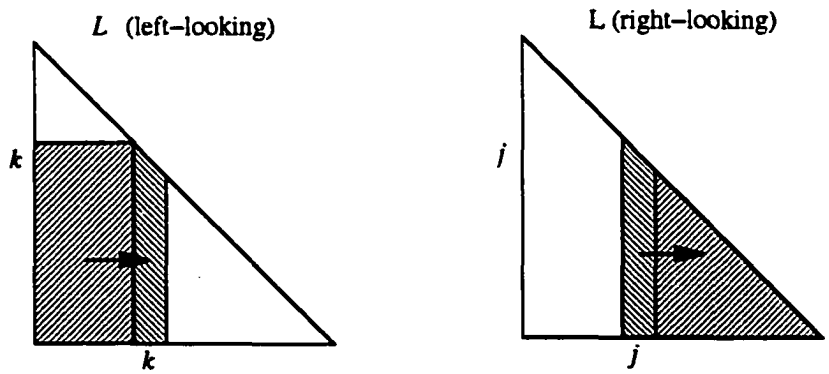


Figure 6: Data access patterns for left-looking and right-looking factorization.

```
1.  for j := 1 to n begin
2.      L_{j,j} := sqrt(A_{j,j});
3.      for i := j + 1 to n
4.          L_{i,j} := A_{i,j} / L_{j,j};
5.      for k := j + 1 to n
6.          for i := k to n
7.              A_{i,k} := A_{i,k} - L_{i,j} L_{k,j};
8.  end
```

Algorithm 2: General right-looking factorization.

separate storage while the contents of A are preserved, or A and L may share the same storage and thus L overwrites A. For convenience, we assume that L is already initialized with data from A before the factorization begins, therefore from this point on only L will show up in the algorithms.

Algorithms 1 and 2 perform two basic column tasks: modifying a column by a multiple of another column (lines 3–4 of Algorithm 1 and lines 6–7 of Algorithm 2) and scaling a column by a scalar (lines 5–7 of Algorithm 1 and lines 2–4 of Algorithm 2). These are called column factorization (FACTOR) and column update (UPDATE), respectively.

These tasks operate on factor columns, which we denote by subscripting L with the corresponding column indices, such as L_j and L_k. We omit row indices in order to keep the notation simple.

We also describe factorization algorithms expressed in terms of block tasks. We denote block columns (groups of columns) by subscripting L with the corresponding range of column indices, such as $L_{p:q}$, omitting row indices again. On the other hand, we denote block entries (horizontal slices of block columns) by subscripting L with the corresponding ranges of both row and column indices, such as $L_{s:t,p:q}$.

To justify the term column factorization note that we can factor any block column in general. If the column indices within the block column range from p to q then the block column factorization is denoted as FACTOR($L_{p:q}$). The whole factorization corresponds to $p = 1$ and $q = n$ while the factorization of column j corresponds to $p = q = j$. Reduced to column level, the whole factorization is thus a collection of column factorization and column update tasks.

Note that the FACTOR($L_{p:q}$) task groups all the column factorization and update tasks associated

with the columns from block column $L_{p:q}$. Similarly, the UPDATE($L_{p:q}, L_{s:t}$) groups all the column update operations between columns from block column $L_{p:q}$ and columns from block column $L_{s:t}$. The same ideas apply to block entry factorization algorithms, with the addition of row indices.

For column based factorization algorithms there is thus one column factorization task for each outer loop iteration and one column update task for each middle loop iteration. The column factorization tasks are performed in the increasing order of the column indices but the relative order between column factorization and column update tasks depends on the factorization algorithm. There is a whole range of possibilities, with left-looking and right-looking at the extremes. In left-looking factorization the update tasks to column k take place immediately before the factorization of column k. In right-looking factorization the update tasks from column j take place immediately after the factorization of column j.

Algorithms 3 and 4 represent the left-looking and right-looking factorization, respectively, rewritten in terms of column factorization and update tasks.

The sparse factorization algorithms are variations of the general ones that take advantage of the nonzero structure of L. This is described by the following sets:

$$\begin{aligned} row[k] &= \{j \mid j \leq k,\ L_{j,k} \neq 0\} \\ col[j] &= \{k \mid k \geq j,\ L_{k,j} \neq 0\}. \end{aligned}$$

Algorithms 5 and 6 represent the sparse left-looking and right-looking factorization, respectively, expressed in terms of column factorization and update tasks.

2.3 SPARSE MATRICES AND GRAPHS

Sparse matrix computations can be conveniently expressed in terms of graphs [21]. A one-to-one correspondence exists between symmetric matrices (ignoring numerical values) and undirected graphs. Given a symmetric matrix, its associated graph $G(A)$ is built as follows: for each row/column create a node; for each pair of nonzero off-diagonal entries create an edge between the nodes corresponding to the rows/columns in which the nonzero entries lie.

```
1.  for k := 1 to n begin
2.      for j := 1 to k − 1
3.          UPDATE(L_j, L_k);
4.      FACTOR(L_k);
5.  end
```

Algorithm 3: General left-looking factorization expressed in terms of FACTOR and UPDATE tasks.

```
1.  for j := 1 to n begin
2.      FACTOR(L_j);
3.      for k := j + 1 to n
4.          UPDATE(L_j, L_k);
5.  end
```

Algorithm 4: General right-looking factorization expressed in terms of FACTOR and UPDATE tasks.

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

Algorithm 5: Sparse left-looking factorization expressed in terms of FACTOR and UPDATE tasks.

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

Algorithm 6: Sparse right-looking factorization expressed in terms of FACTOR and UPDATE tasks.

We use an example in order to illustrate the relationship between sparse matrices and graphs. Consider matrix A and the corresponding factor L from Figure 7. $G(A)$ is shown in the top left corner of Figure 8.

Now consider the matrix $F = L + L^T$, known as the *filled matrix*. Obviously, this matrix is symmetric as well and an undirected graph $G(F)$ can be associated with it. There is an important relationship between $G(F)$ and $G(A)$: the set of edges of $G(A)$ is a subset of the set of edges of $G(F)$ [22]. $G(F)$ is usually denoted as $G^+(A)$ and it is called the *filled graph* of A. Those edges that are in $G^+(A)$ but not in $G(A)$ are called *fill edges* and they correspond to the fill entries [35]. $G^+(A)$ for the current example is shown in the top right corner of Figure 8. I used dotted lines for the fill edges in order to distinguish them from the original ones.

The number of edges in $G(A)$ is denoted as e, while the number of edges in $G^+(A)$ is denoted as e^+. They correspond to the number of entries that lie below the diagonal in A and L, respectively. The total number of entries in A is thus $n + 2e$. Similarly, the total number of entries in L is $n + e^+$.

The factorization can be equivalently viewed as a procedure that turns $G(A)$ into $G^+(A)$. Assume that the white color corresponds to A and the black color corresponds to L. Initially we have $G(A)$, with all the nodes and edges being white. The factorization traverses the graph in the order of the node labels (the node labels are the same as the column indices). Each time a node is visited it becomes black. The edges that link it with its neighbors also become black. In addition, all its neighbors are linked by black edges (some of which may already be there, just white colored). In the end, we obtain $G^+(A)$, with all the nodes and edges being black.

Another convenient tool for sparse factorization is the *elimination tree* [35], which is nothing but the transitive reduction [3, 4] of the directed version of $G^+(A)$ (in which each edge is directed from its lower numbered to its higher numbered endpoint). The elimination tree captures all the column dependencies in L and is therefore used by the factorization algorithms, which traverse the tree in postorder or, more generally, in topological order [11, 52].

Note that in general this structure is a forest, as the filled graph may not be connected. In this case the term *elimination forest* is used. Without any loss of generality, we assume that the filled

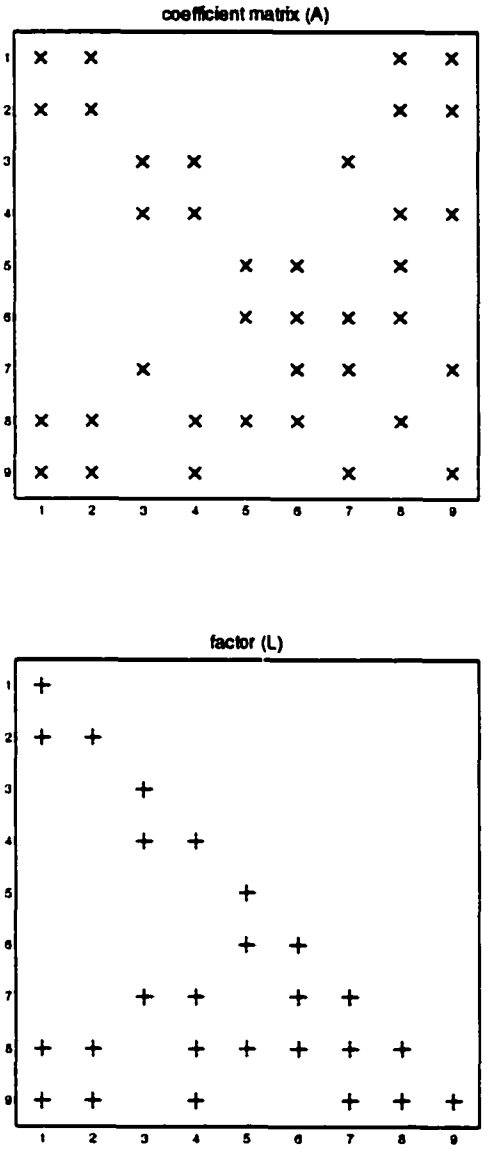


Figure 7: A sparse symmetric matrix and the corresponding factor.

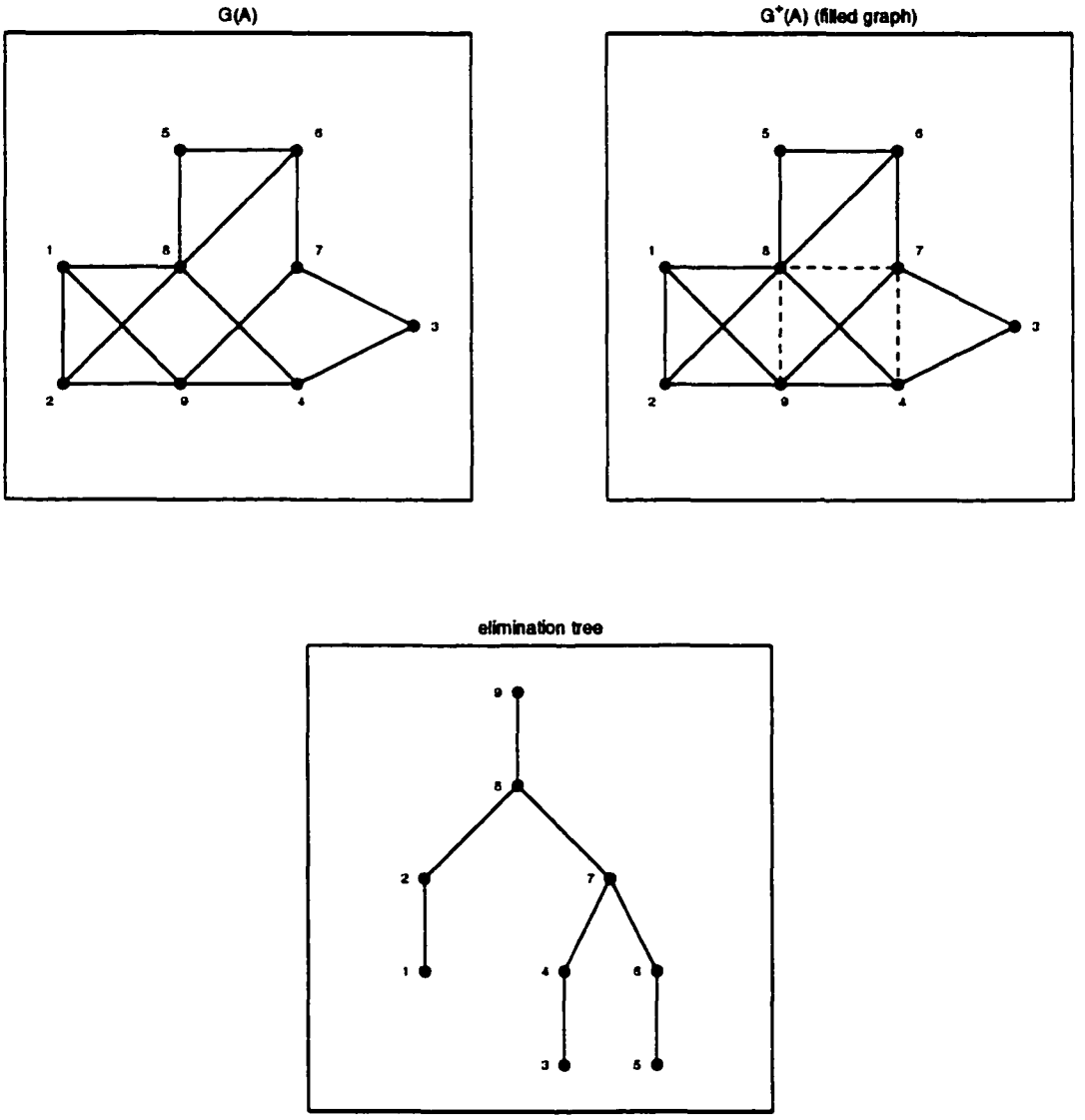


Figure 8: The graphs associated with A and the elimination tree.

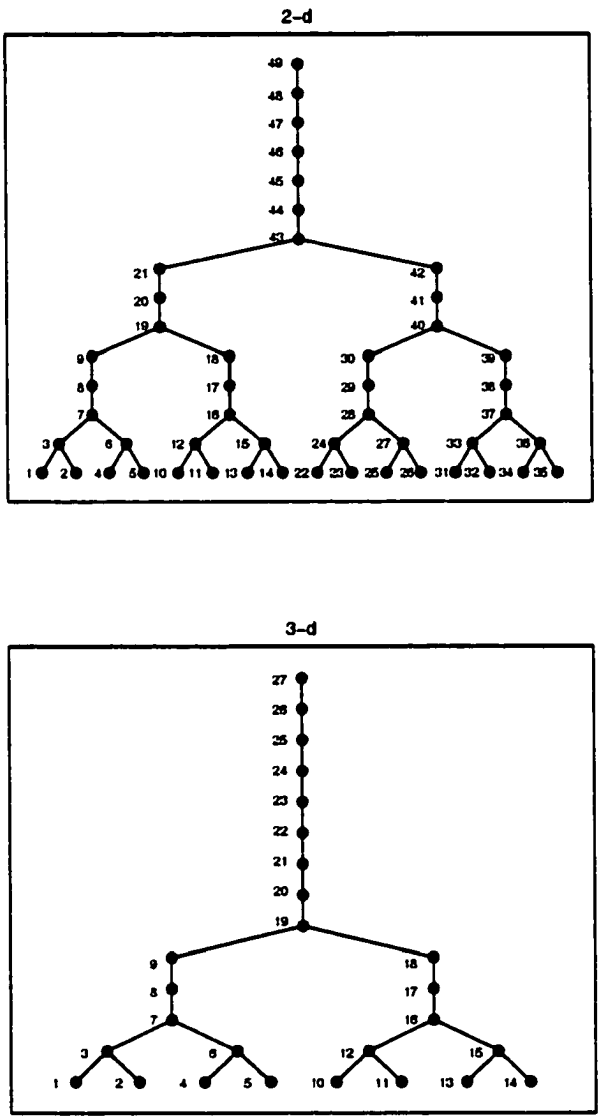


Figure 9: The elimination trees for the 2-d and 3-d model examples.

graph is connected, which corresponds to an *irreducible* coefficient matrix.

The elimination tree can be described as following: for each node j, *parent*[j] is either the parent node of j or 0, if j has no parent; for each node j, *children*[j] is the set of child nodes of j.

The elimination tree for the current example is shown at the bottom of Figure 8. Also, Figure 9 depicts the elimination trees for the two model examples from Section 2.1.

2.4 MULTIFRONTAL FACTORIZATION

The *multifrontal* factorization [18, 36] is different from the left-looking and right-looking factorization in that the UPDATE tasks are not performed directly between factor columns. Instead, updates are carried through a chain of temporary columns.

A temporary column T_k^j is created for task UPDATE(L_j, L_k). Task UPDATE(L_j, L_k) is then replaced by task UPDATE(L_j, T_k^j). In addition, a new type of task, called column assembly task (ASSEMBLE) is created. The ASSEMBLE(T_j^i, L_j) task, for example, adds the entries from temporary column T_j^i to factor column L_j. The addition is performed by matching the row indices.

The updates between factor columns are carried through temporary columns along elimination tree paths. They are passed from children to parents, thus from one level to the one above it.

Algorithm 7 represents the multifrontal factorization. It traverses the elimination tree in postorder and, at node j it creates a temporary column T_k^j for and, for each factor column L_j, it assembles the temporary columns that carry the updates from the children of j into L_j or into temporary columns that carry the updates from j, then it factors L_j and it updates the temporary columns that carry the updates from j. The CLEAR task initializes a temporary column with zero entries.

There is additional terminology for the multifrontal factorization. The temporary columns that carry the updates from j form a dense matrix that is called the *update matrix*, and L_j plus the temporary columns that carry the updates from j form a dense matrix that is called the *frontal matrix*. The update matrix is a submatrix of the frontal matrix. The multifrontal factorization can thus be viewed as a collection of partial factorizations of frontal matrices. Temporary data are

```
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
```

Algorithm 7: Multifrontal factorization.

passed from child to parent nodes using update matrices. The set of update matrices that exist at some point during the factorization is called the *update stack*. The name comes from the fact that, since the elimination tree is traversed in postorder, the update matrices can be stacked.

It is useful to visualize the data access patterns of the three factorization algorithms. Consider the elimination tree from Figure 10, 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 is accessed. If the factorization is right-looking, a subset of the nodes on the path from the current node to the root is accessed. If the factorization is multifrontal, only the current node and its children are accessed.

2.5 SUPERNODES

Dense matrix computations are generally more efficient than their sparse counterparts, for the latter require indirect addressing [6]. For factorization, the performance gap can be narrowed by using *supernodes* [37, 44].

The basic idea is to try to perform dense computations at least on clusters of factor columns, which is possible as long as the columns have the same nonzero structure. Such clusters are known as supernodes.

Supernodes are usually defined with respect to a postordered elimination tree. A *fundamen-*

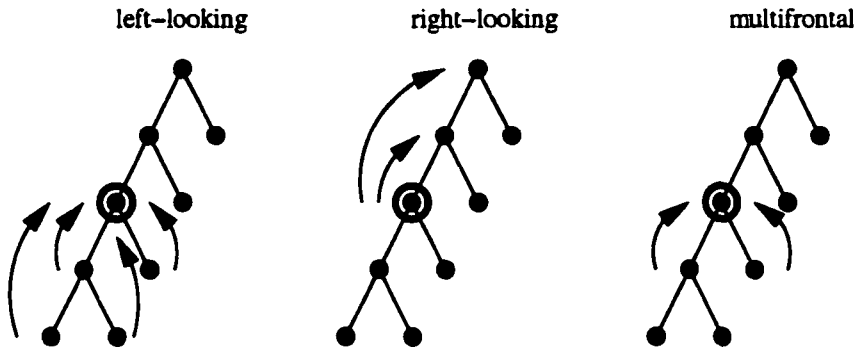


Figure 10: Data access patterns for sparse left-looking, right-looking and multifrontal factorization (the currently processed node is highlighted).

tal supernode groups a maximal number of consecutive columns $L_p, \ldots, L_q$ having the following properties:

- $col[j] = col[j-1]$, for $p < j \leq q$;
- in the elimination tree, the node that corresponds to L_{j-1} is the only child of the node that corresponds to L_j, for $p < j \leq q$.

Supernodes determine a column partition, which is unique for fundamental supernodes. It is possible to define *maximal supernodes*, which determine a slightly better partition, but a maximal supernode partition is not unique. We use fundamental supernodes for convenience.

In order to denote supernodes we use capital letters and the subscript s, such as J_s and K_s. The subscript distiguishes between supernodes and blocks (defined later). The total number of supernodes is denoted by N_s.

Note that the nodes of the elimination tree are clustered as well (merged into the supernodes) and in practice the supernodal elimination tree is used rather than the original one. This is still described by the *parent* and *children* data structures, updated according to the supernodes.

Figure 11 shows the fundamental supernode partition for the 3-d model problem example from Section 2.1.

In addition to the performance increase, supernodes help saving storage. A sparse symmetric direct solver usually stores only the *col* sets, the *row* sets being dynamically computed, if required. With supernodes, the *col* sets can be stored in a compressed way. The nonzero structure of each

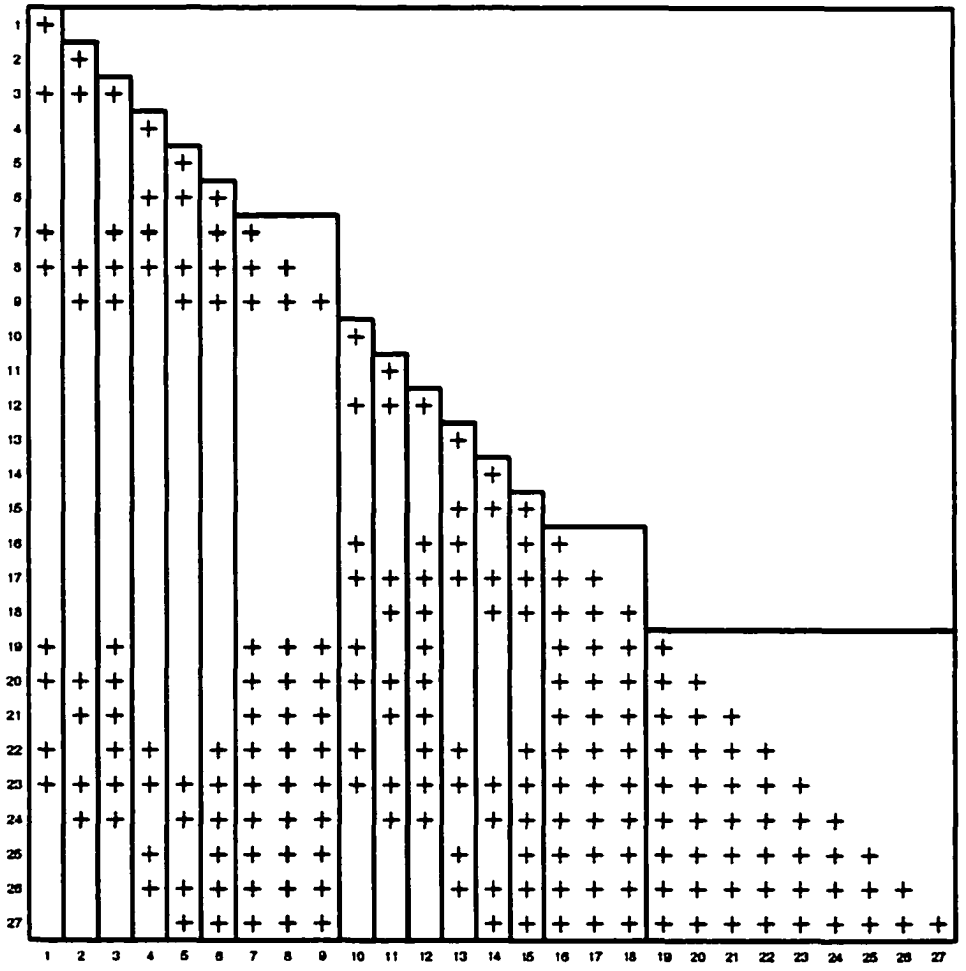


Figure 11: The fundamental supernode partition for the 3-d model example.

supernode is stored instead of the nonzero structure of each column. This can be done only for the *col* sets, not for the *row* sets. The compression reduces the number of row indices to a value that we denote as m. For 2-d and 3-d problems ordered by nested dissection $m = \Theta(n)$. Compare it with $\Theta(n \log n)$ and $\Theta(n^{4/3})$, required when the nonzero structure of L is not compressed.

2.6 THE FACTORIZATION IN A HIERARCHICAL STORAGE CONTEXT

When storage is hierarchical we face an additional issue: data movement. Consider a disk/core storage system. Assume that the core has a finite size M while the size of the disk is infinite and suppose the input is initially stored on the disk and that the output must be stored on the disk as well. The computation, however, can only take place on data that is stored within the core.

As a consequence, during the factorization input data must be moved into the core (read) and output data must be moved out of the core (written). This leads to several scenarios, as described in Figure 12. The horizontal axis corresponds to the core size and the values M_1^*, M_2^* and M_3^* depend on L and on the factorization algorithm, the asterisk being a placeholder for L (left-looking), R (right-looking) or M (multifrontal).

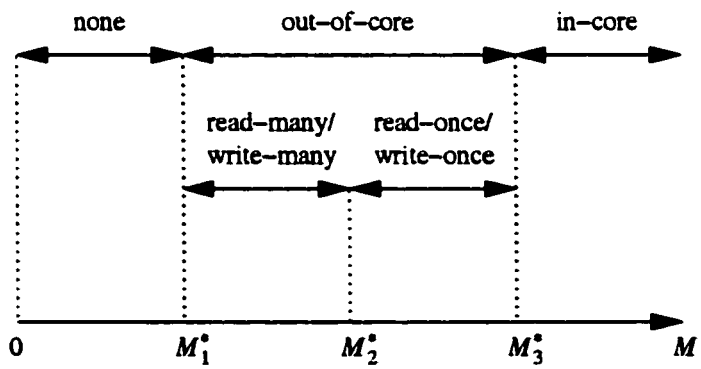


Figure 12: Factorization scenarios determined by a disk-core storage system.

In this dissertation we focus only on numerical values, therefore M_1^*, M_2^* and M_3^* do not take any other data into account. In practice these values must be larger because the core needs to accommodate additional data, such as the nonzero structure of the factor and the structure of the

elimination tree.

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 $2|col[j]|$, where j is the column with the largest number of entries. This is determined by column update and column assembly operations, which require two columns to be stored simultaneously in core. If the unit of data movement is the entry then $M_1^* = 3$. This is determined by update operations on individual entries, which require three entries to be stored simultaneously in core (see Algorithms 1 and 2 for example). It is reasonable to assume the former value overall ($2|col[j]|$, column j having the largest number of entries) because it covers both cases.

For $M \geq M_3^*$, the factorization can be performed in-core. The core is large enough to store all the data, therefore minimum traffic is required: reading the input at the beginning of the factorization and writing the output at the end of the factorization. For left-looking and right-looking factorization we have $M_3^L = M_3^R = |L|$. For multifrontal factorization the core must accommodate the temporary data as well. Denoting the maximum number of entries in the update stack by $|U|$, we can write $M_3^M = |L| + |U|$.

The case of interest for this dissertation is $M_1^* \leq M < M_3^*$, when the factorization can be performed only out-of-core. Data are moved not only at the beginning and at the end of the factorization, but also during the factorization.

Two major out-of-core scenarios can be identified. The first scenario, called *read-once/write-once* (R1/W1), corresponds to $M_2^* \leq M < M_3^*$. In this case the core is still large enough to allow the minimum traffic of the in-core factorization, with the difference is that this time data can move during the factorization as well. The second scenario, called *read-many/write-many* (RM/WM), corresponds to $M_1^* \leq M < M_2^*$. In this case the core is smaller and a larger traffic is required.

For R1/W1 factorization the input is read once and the output is written once. If any temporary data are used, as in multifrontal factorization, they exist only in core and thus are not subject to movement. For RM/WM factorization the input is also read once. The difference comes with the output and with the temporary data. If the factorization is left-looking then the output is written

once but may be read many times. If the factorization is right-looking then the output may be both read and written many times. For multifrontal factorization the output is never read, it is written once, and the temporary data may move in and out of the core several times.

Three natural questions can be asked for out-of-core factorization:

- what is the value of M_2^* (the boundary between the two scenarios)?
- how often do we need to reorganize the data structure that stores the numerical values and how much data do we need to move within the core if the factorization is R1/W1?
- what is the minimum amount of traffic for the RM/WM factorization?

The core minimization issue was investigated by Liu [32, 33, 34]. He assumed that A is already ordered by a fill reducing algorithm and he used topological orders of the elimination tree. He provided core minimization algorithms for left-looking and multifrontal factorization algorithms. Ashcraft proved similar results for right-looking algorithms [8]. For left-looking factorization Liu also studied the data reorganization issue [34].

Investigations related to the sparse factorization traffic tend to be rather experimental, focusing on execution time (the traffic is a key factor in the execution time.) In a recent study Rothberg and Schreiber [47] considered left-looking and multifrontal algorithms as well as left-looking/multifrontal hybrids. There is also related work by Rothberg and Gupta [46, 45], targeted at the higher layers of the memory hierarchy.

More theoretical investigations of the traffic exist for other types of computations. Aggarwal and Vitter [2], then Vitter and Shriver [57, 58] studied the traffic complexity of computations such as sorting, FFT, matrix transposition and standard matrix multiplication of dense matrices [11]. The traffic complexity of the standard matrix multiplication of dense matrices and of the FFT were also considered by Hong and Kung [26]. Another traffic complexity study for the FFT belongs to Cormen and Nicol [12]. On the sparse side, Ullman and Yannakakis [56] looked the traffic complexity of the transitive closure [11]. There is also a good discussion of out-of-core factorization concepts by Dongarra [17].

A good survey on out-of-core algorithms in numerical linear algebra was recently provided by Toledo [54]. Two recent books by Abello and Vitter [1] and by May [38] provide more information about the current interest in external memory algorithms.

2.7 BLOCKS

In Section 2.5 we talked about the performance issue that comes with sparse matrix computations, and we showed that supernodes are a solution to this problem. Another performance issue is determined by the hierarchical nature of the storage. In this case the performance is affected by the data movement, high performance corresponding to low traffic.

Considering the RM/WM scenario (for R1/W1 the traffic is minimal), the key issue is data reuse. A data item is said to be reused if it is accessed by two or more operations but it does not need to be moved into the core each time it is accessed, some operations finding it in core. This helps reduce the traffic.

The potential for data reuse depends on the computation at hand. It is high only when the number of operations is significantly larger than the number of accessed data items.

Consider three major types of dense linear algebra computations [23]: vector-vector, matrix-vector, and matrix-matrix. Suppose the vectors and the matrices are all of order n. Then vector-vector operations access $\Theta(n)$ data for $\Theta(n)$ arithmetic, matrix-vector operations access $\Theta(n^2)$ data for $\Theta(n^2)$ arithmetic, and matrix-matrix operations access $\Theta(n^2)$ data for $\Theta(n^3)$ arithmetic. Table 1 summarizes the complexity of the arithmetic work and data accesses for these three linear algebra computations.

As a consequence, only matrix-matrix computations have a potential for data reuse. The factorization is a matrix-matrix computation but, since we are interested in sparse factorization, the potential for data reuse depends on sparsity as well. At one extreme the filled graph is a clique and $\Theta(n^2)$ data is accessed for $\Theta(n^3)$ arithmetic. At the other extreme the filled graph is the same as the elimination tree and $\Theta(n)$ data is accessed for $\Theta(n)$ arithmetic. 2-d and 3-d problems ordered by nested dissection lie somewhere in the middle, with $\Theta(n \log n)$ data accessed for $\Theta(n^{3/2})$ arithmetic

and $\Theta(n^{4/3})$ data accessed for $\Theta(n^2)$ arithmetic, respectively. Table 2 summarizes the complexity of the arithmetic work and data accesses for the factorization in these four cases.

type	work	data	ratio
vector-vector	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
matrix-vector	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
matrix-matrix	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n)$

Table 1: The complexity of the arithmetic work and data accesses for three basic dense linear algebra operations.

type	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 2: The complexity of the arithmetic work and data accesses for factorization, for various problems.

Accordingly, the denser the filled graph, the higher the potential for data reuse. On the other hand, the denser the filled graph, the larger the factor. Thus, the potential for data reuse grows with the factor.

The technique of choice for data reuse is blocking [41]. The whole computation is decomposed into a set of block computations, each one of them operating on a subset of data items, or a data block. Given a particular block computation the corresponding data items are read before the block computation begins and written after the block computation ends. No data movement occurs during the block computation. This sets a limit on the block size, determined by the maximum number of blocks involved in a block computation.

For dense matrices blocking is simple. A data block is determined by grouping adjacent columns and/or rows. For sparse matrices blocking is a little more complicated. We need to focus on dense areas, where data can be reused. Accordingly, blocks must be defined within supernode boundaries. Except for that, the principle is the same: group adjacent columns and/or rows.

In order to denote blocks we use capital letters and the subscript b, such as J_b and K_b. The total number of column (or row) clusters determined by blocking is denoted by N_b.

Partitioning columns by blocks determines 1-d blocks (block columns). We denote a block column by subscripting L with the corresponding block index, such as L_{J_b}.

It is also possible to determine 2-d blocks (block entries) by applying the block partition to both columns and rows. We denote a block entry by subscripting L with the corresponding block indices, such as L_{J_b,K_b}.

The block nonzero structure of the factor is described by *Row* and *Col* sets, which are similar to the *row* and *col* sets:

$$\begin{aligned} Row[K_b] &= \{J_b \mid J_b \leq K_b,\ L_{J_b,K_b} \neq 0\} \\ Col[J_b] &= \{K_b \mid K_b \geq J_b,\ L_{K_b,J_b} \neq 0\}. \end{aligned}$$

We can look now at blocked factorization algorithms. For 1-d blocking the algorithms are very similar to their nonblocked counterparts. Algorithms 8 and 9 for example represent the 1-d blocked left-looking and right-looking factorization, respectively. In this case we use block column factorization and update tasks.

For 2-d blocking we basically need to index rows as well. Consider Algorithms 10 and 11 for example, which represent the 2-d blocked left-looking and right-looking factorization, respectively. In this case we use block entry factorization and update tasks.

Note that a block column factorization task involves one block column while a block entry factorization task generally involves two block entries. The taks FACTOR(L_{I_b,J_b}) for example involves block entries L_{J_b,J_b} and L_{I_b,J_b}. Similarly, a block column update task involves two block columns while a block entry update task generally involves three block entries. The task UPDATE(L_{I_b,J_b}, L_{I_b,K_b}) for example involves block entries L_{J_b,J_b}, L_{I_b,J_b} and L_{I_b,K_b}.

```
1.  for K_b := 1 to N_b begin
2.      for J_b in Row[K_b] \ {K_b}
3.          UPDATE(L_{J_b}, L_{K_b});
4.      FACTOR(L_{K_b});
5.  end
```

Algorithm 8: 1-d blocked sparse left-looking factorization.

```
1.  for J_b := 1 to N_b begin
2.      FACTOR(L_{J_b});
3.      for K_b in Col[J_b] \ {J_b}
4.          UPDATE(L_{J_b}, L_{K_b});
5.  end
```

Algorithm 9: 1-d blocked sparse right-looking factorization.

```
1.  for K_b := 1 to N_b begin
2.      for J_b in Row[K_b] \ {K_b}
3.          for I_b in (Col[J_b] \ {J_b}) ∩ Col[K_b]
4               UPDATE(L_{I_b,J_b}, L_{I_b,K_b});
5.      for I_b in Col[K_b]
6.          FACTOR(L_{I_b,K_b});
7.  end
```

Algorithm 10: 2-d blocked sparse left-looking factorization.

```
1.  for J_b := 1 to N_b begin
2.      for I_b in Col[J_b]
3.          FACTOR(L_{I_b,J_b});
4.      for K_b in Col[J_b] \ {J_b}
5.          for I_b in (Col[J_b] \ {J_b}) ∩ Col[K_b]
6.              UPDATE(L_{I_b,J_b}, L_{I_b,K_b});
7.  end
```

Algorithm 11: 2-d blocked sparse right-looking factorization.

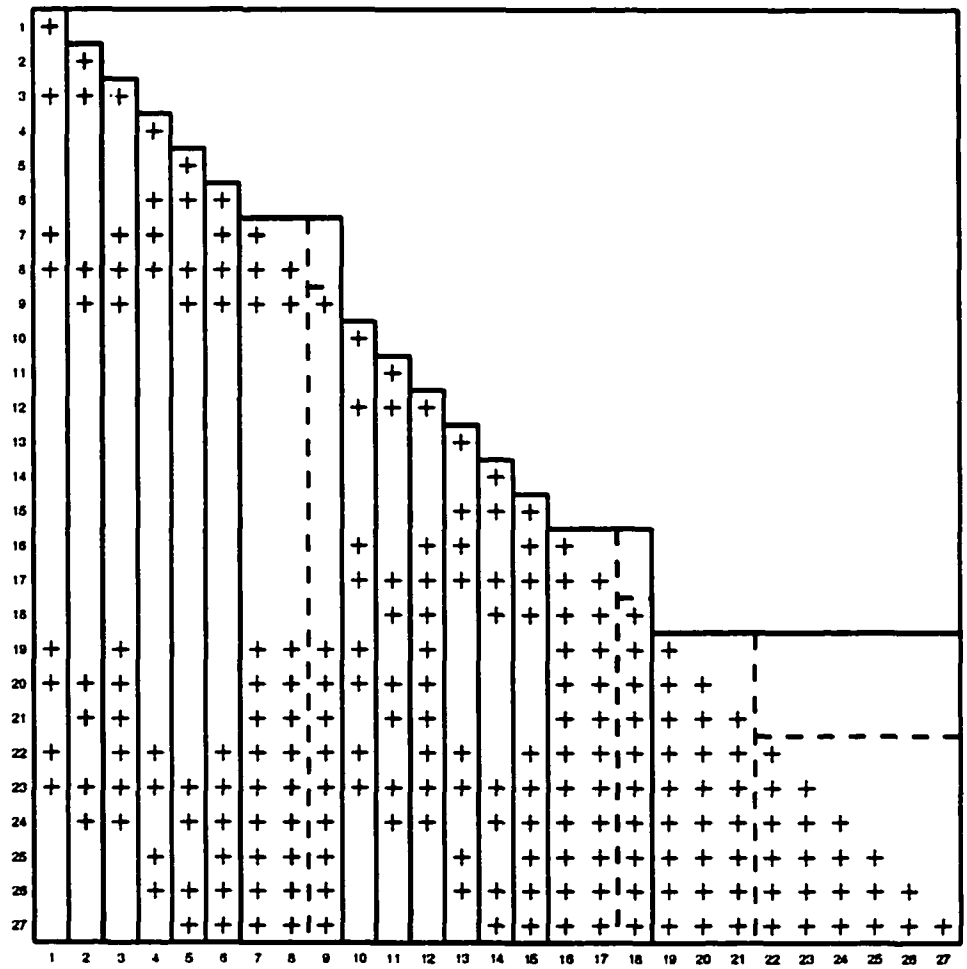


Figure 13: A 1-d block partition for the 3-d model example.

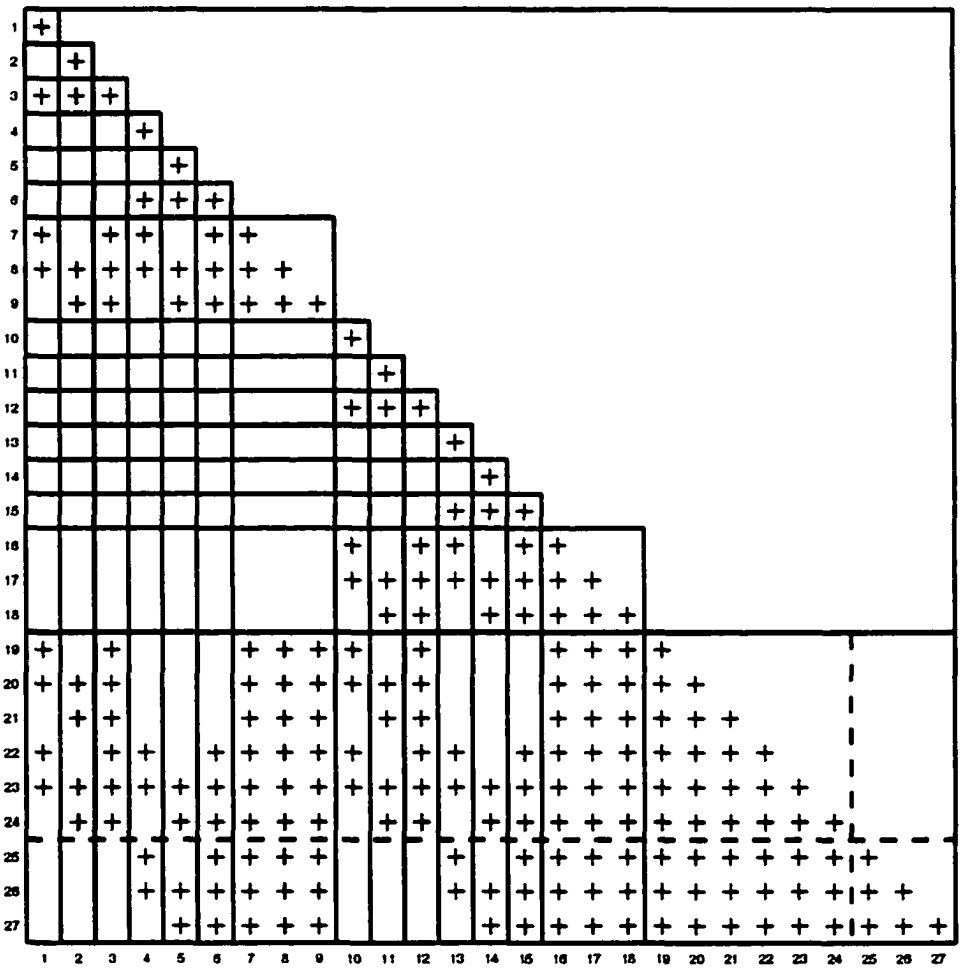


Figure 14: A 2-d block partition for the 3-d model example.

As a consequence, a block column task involves at most two block columns while a block entry task involves at most three block entries. Therefore a block column cannot be larger than half of the core and a block entry cannot be larger than one third of the core.

In order to illustrate blocking, consider the 3-d model example from Section 2.1 again, with the fundamental supernode partition described in Figure 11. Assume $M = 48$ (a maximum of 48 entries can be stored in core). Then a possible block partition that corresponds to 1-d blocking is shown in Figure 13. In a similar way, a block partition that corresponds to 2-d blocking is shown in Figure 14.

Finally, remember that blocking is associated with data movement. However, no data movement is shown in Algorithms 8 through 11. In this form these algorithms correspond to implicit data movement, performed automatically by the operating system through pages [51]. The other alternative is explicit data movement, performed by the programmer through files. With implicit data movement we can basically rely on the same data structures used by nonblocked algorithms, enhanced with pointers to the block boundaries. With explicit data movement we generally need to decompose the data, which leads to more sophisticated data structures. Implicit data movement is thus easier from the programming perspective but the control that we have with explicit data movement potentially determines better data reuse. The algorithms I discuss in Chapters III and IV use explicit data movement.

CHAPTER III

READ-ONCE/WRITE-ONCE FACTORIZATION

In the R1/W1 scenario the core is large enough to allow minimum traffic: reading the input and writing the output. Data movement can be, however, interleaved with the computation.

The first issue of interest is the minimum core size that allows minimum traffic. Liu investigated the core minimization problem for left-looking [34] and multifrontal [33] factorization using postorders. In both cases the core size is minimized by sorting the children of each elimination tree node in a particular way. He also pushed the investigation further, replacing postorders with the more general topological orders [32]. He considered only left-looking factorization in his analysis but he mentioned that the analysis applies to multifrontal factorization as well. Right-looking algorithms were considered by Ashcraft [8]. In this case the minimum core size is determined by the path with the maximum core requirement.

The second issue of interest is the number of core reorganizations, previously investigated by Liu in the context of left-looking factorization [34].

In this chapter we describe R1/W1 factorization algorithms and we study the minimum core size that allows minimum traffic, M_2^*. We simply refer to it as the minimum core size. We approach this issue theoretically as well as experimentally. For our theoretical study we consider various model problems for which we determine the complexity of M_2^*. Then we describe simulation algorithms that compute the exact value of M_2^* and we show results obtained with the simulation algorithms for selected problems.

At the end of the chapter we also investigate the core reorganizations. This time the study is purely experimental. We are interested in both the number of core reorganization and the amount of data that is moved within the core. We describe simulation algorithms that compute these quantities and, again, we show results obtained with the simulation algorithms for selected problems.

3.1 READ-ONCE/WRITE-ONCE FACTORIZATION ALGORITHMS

The general left-looking, right-looking and multifrontal algorithms can be easily adapted to the R1/W1 context. As an example, Algorithms 12 and 13 represent the R1/W1 left-looking and right-looking factorizations, respectively.

New tasks are present in the R1/W1 factorization algorithms. Two of them, READ and WRITE, manage the data movement between the disk and the core. Two other tasks, ALLOCATE and REORGANIZE, manage the allocation of the core storage. The ABORT task terminates the execution if the core is not large enough to allow minimum traffic.

The idea is to read the columns as they are needed. In R1/W1 left-looking factorization, column L_k is read at the beginning of step k. Column L_k is then updated, factored and written. As the factorization proceeds, entries at the beginning of column L_k that are no longer needed can be discarded. A test is performed before reading column L_k if there is still enough core for the entries in L_k then L_k is immediately read and the factorization proceeds; otherwise, the data that is already in core is reorganized. Reorganizing data means discarding entries that are no longer needed from each column that is in core and packing the remaining entries. When the REORGANIZE task finishes there is a larger chunk of free storage within the core. The test for available core space is performed twice. If it fails the second time it means that the core is not large enough to allow minimum traffic and the factorization should abort.

The R1/W1 right-looking factorization works in a similar way. This time during step j more than a column can be read. First there is column L_j, which is read at the beginning of step j. Then there is each column L_k, with $k \in col[j] \setminus \{j\}$, that is not already in core. Thus, at the beginning of step j column L_j may already be in core, read during a previous step because some other column updated it. As a consequence, a test must be performed before attempting to read a column, in order to determine if that column is already in core. If the column is already in core then the factorization proceeds. Otherwise the core storage availability test is performed as in the left-looking algorithm, potentially followed by a core reorganization as well as by an abnormal termination.

The R1/W1 multifrontal factorization, not shown here, follows a similar strategy. This time we

```
1.    for k := 1 to n begin
2.       if not enough core for L_k
3.          REORGANIZE();
4.       if not enough core for L_k
5.          ABORT();
6.       ALLOCATE(L_k);
7.       READ(L_k);
8.       for j in row[k] \ {k}
9.          UPDATE(L_j, L_k);
10.      FACTOR(L_k);
11.      WRITE(L_k);
12.   end
```

Algorithm 12: R1/W1 left-looking factorization.

```
1.    for j := 1 to n begin
2.       if L_j not in core begin
3.          if not enough core for L_j
4.             REORGANIZE();
5.          if not enough core for L_j
6.             ABORT();
7.          ALLOCATE(L_j);
8.          READ(L_j);
9.       end
10.      FACTOR(L_j);
11.      WRITE(L_j);
12.      for k in col[j] \ {j} begin
13.         if L_k not in core begin
14.            if not enough core for L_k
15.               REORGANIZE();
16.            if not enough core for L_k
17.               ABORT();
18.            ALLOCATE(L_k);
19.            READ(L_k);
20.         end
21.         UPDATE(L_j, L_k);
22.      end
23.   end
```

Algorithm 13: R1/W1 right-looking factorization.

also have to deal with temporary columns. These are kept in core as long as they are needed. When they are no longer needed they are simply discarded from the core. The core can still be reorganized, but this is meaningful only during the partial factorization of a frontal matrix. No room can be made while assembling temporary data. Only factor columns are affected while reorganizing the core. A temporary column is either fully stored in core or it does not exist in core at all.

3.2 THE COMPLEXITY OF THE MINIMUM CORE

We analyze the complexity of M_2^* for various model problems. We consider the following three factors:

- branching in the elimination tree;
- balance in the elimination tree;
- connectivity in the filled graph.

The following additional notation is required:

- h: the height of the elimination tree (the number of nodes along the longest tree path);
- d: the depth of a node in the elimination tree, measured from the root, where $d = 0$; the maximum depth is $h - 1$;
- H: the height of the supernodal elimination tree (the number of supernodes along the longest tree path);
- D: the depth of a supernode in the elimination tree, measured from the root, where $D = 0$; the maximum depth is $H - 1$.

Remember also that we denote the maximum number of entries in the update stack by $|U|$. We use $|F|$ as well, to denote the maximum number of entries in a frontal matrix. Both $|U|$ and $|F|$ are required in order to determine the complexity of M_2^M.

We begin with simple types of trees: path, balanced p-ary, star and unbalanced p-ary. The path and the star delimit the range of the balanced trees and the balanced p-ary trees fall in this

range. The range of connectivity is delimited by the sparsest filled graph, which is the same as the elimination tree, and by the densest filled graph, in which each descendant-ancestor pair (from the elimination tree) is connected. We also consider a more general version of the star. At the end we look at an idealized model of the trees that correspond to grids with r dimensions.

Some of the complexity results are expressed in terms of logarithms. Whenever the base of the logarithm is not specified it is considered to be equal to two.

Path Elimination Tree

Figure 15 shows a path elimination tree and the corresponding factors for lowest and highest connectivity, respectively.

Lemma 1 *If the elimination tree is a path, we have the following:*

- *for lowest connectivity,* $|L| = \Theta(n)$*; for highest connectivity,* $|L| = \Theta(n^2)$*;*
- *for lowest connectivity,* $M_2^L = \Theta(1)$*; for highest connectivity,* $M_2^L = \Theta(n^2)$*;*
- *for lowest connectivity,* $M_2^R = \Theta(1)$*; for highest connectivity,* $M_2^R = \Theta(n^2)$*;*
- *for lowest connectivity,* $M_2^M = \Theta(1)$*; for highest connectivity,* $M_2^M = \Theta(n^2)$*.*

Proof

The height of the elimination tree is equal to the number of nodes:

$$h = n.$$

- $|L|$

  For lowest connectivity, there are $n-1$ edges in the filled graph, thus

  $$|L| = 2n - 1.$$

  For highest connectivity, there are $n(n-1)/2$ edges in the filled graph, thus

  $$|L| = \frac{n(n-1)}{2} + n$$

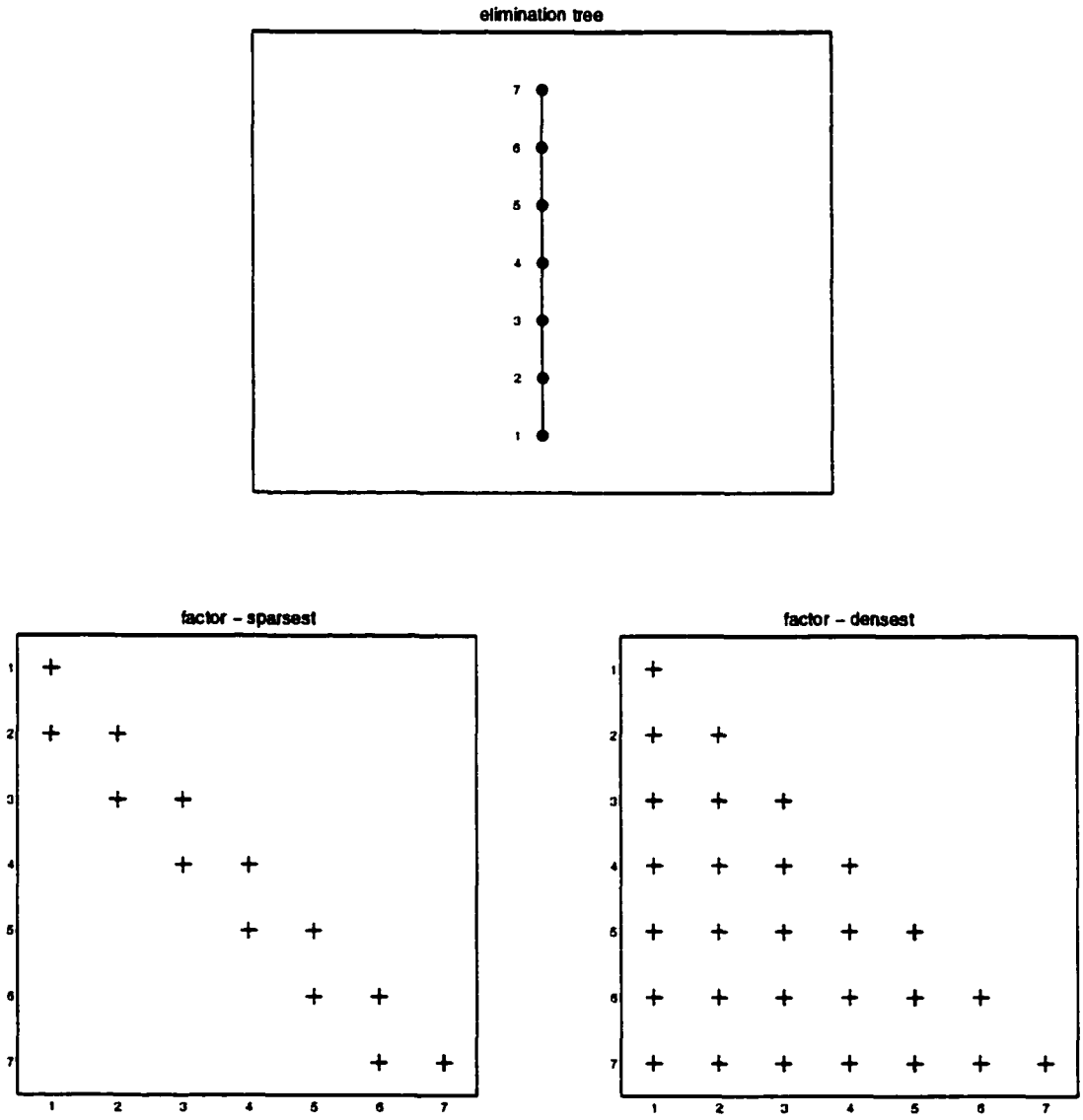


Figure 15: Path elimination tree, and corresponding factors for lowest and highest filled graph connectivity, respectively.

$$= \frac{n(n+1)}{2}$$
$$= \frac{n^2}{2} + O(n).$$

- M_2^L

For lowest connectivity, at most three entries need to be stored, thus

$$M_2^L = 3.$$

For highest connectivity we find the maximum value of the area that corresponds to the number of entries that need to be stored at each step. For convenience, we switch from discrete to continuous values. The area can be approximated as $n(n-x)$, x being the distance from 0 and ranging from 0 to n. The maximum is reached at $x = n/2$ and its value is $n^2/4$. More precisely, in discrete values, at most $n(n+2)/4$ entries must be stored if n is even, and at most $(n+1)^2/4$ entries must be stored if n is odd. We can thus write

$$M_2^L = \frac{n^2}{4} + O(n).$$

- M_2^R

For lowest connectivity, at most three entries need to be stored, thus

$$M_2^R = 3.$$

For highest connectivity we need to add the entries along the longest path, thus

$$M_2^R = \sum_{d=0}^{h-1}(d+1)$$
$$= \sum_{i=1}^{n} i$$
$$= \frac{n(n+1)}{2}$$
$$= \frac{n^2}{2} + O(n).$$

- M_2^M

For lowest connectivity, the update stack needs to store at most one entry and the largest frontal matrix has three entries, thus

$$\begin{aligned} |U| &= 1, \\ |F| &= 3. \end{aligned}$$

For highest connectivity, since the filled graph is a clique of size n, the supernodal elimination tree has a single supernode and the multifrontal factorization degenerates into left-looking factorization, thus

$$M_2^M = \frac{n^2}{4} + O(n). \quad \square$$

Balanced p-ary Elimination Tree

In a p-ary tree each node that is not a leaf has p children, where p is a positive integer larger than one. Figure 16 shows a balanced p-ary elimination tree with $p = 2$ (binary) and the corresponding factors for lowest and highest connectivity, respectively.

Lemma 2 *If the elimination tree is balanced p-ary, we have the following:*

- *for lowest connectivity,* $|L| = \Theta(n)$*; for highest connectivity,* $|L| = \Theta(n \log n)$*;*
- *for lowest connectivity,* $M_2^L = \Theta(1)$*; for highest connectivity,* $M_2^L = \Theta(n)$*;*
- *for lowest connectivity,* $M_2^R = \Theta(1)$*; for highest connectivity,* $M_2^R = \Theta((\log n)^2)$*;*
- *for lowest connectivity,* $M_2^M = \Theta(\log n)$*; for highest connectivity,* $M_2^M = \Theta((\log n)^3)$.

Proof

At depth d there are p^d nodes, therefore the total number of nodes is

$$\begin{aligned} n &= \sum_{d=0}^{h-1} p^d \\ &= \frac{p^h - 1}{p - 1}. \end{aligned}$$

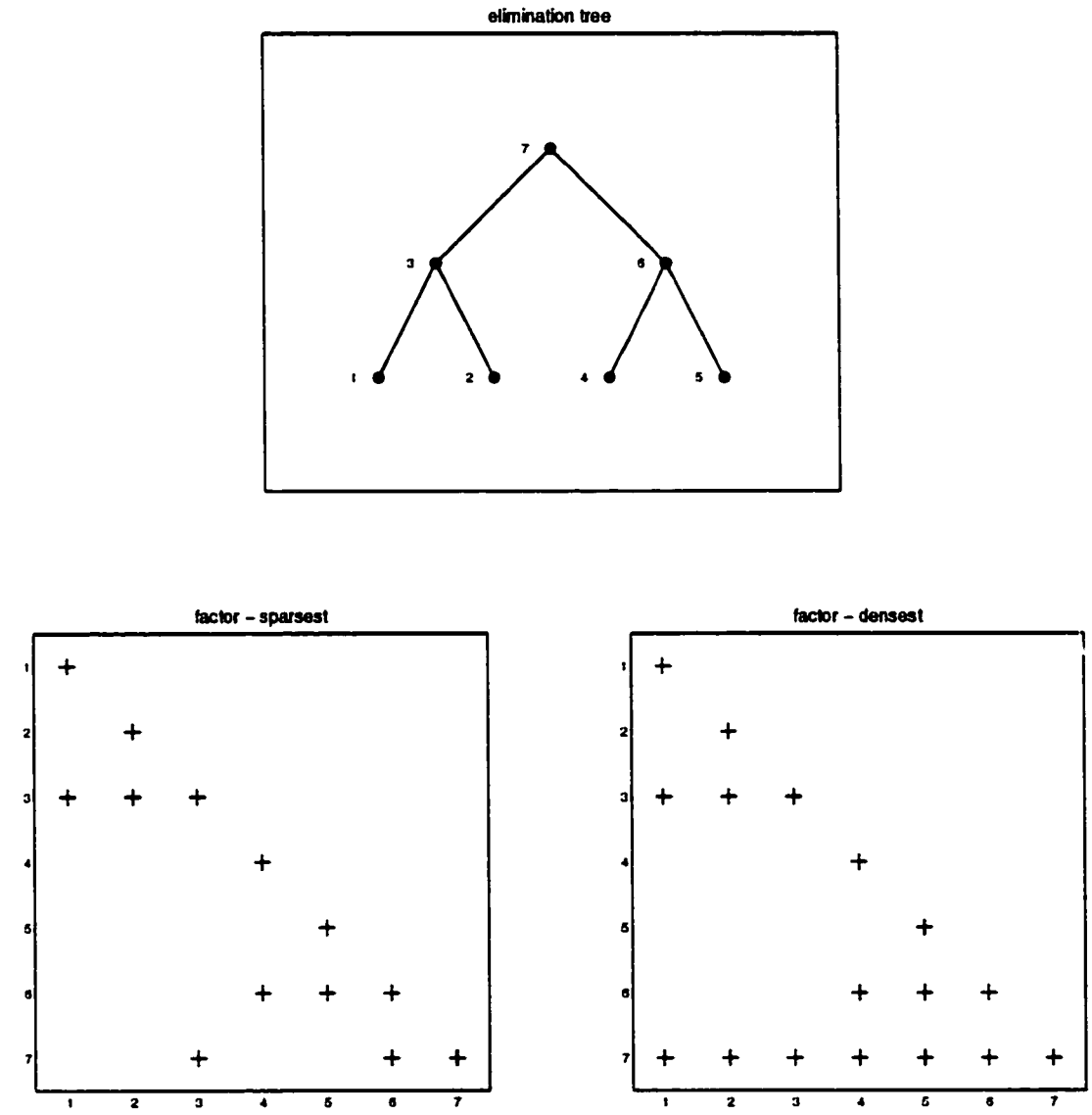


Figure 16: Balanced p-ary elimination tree with $p = 2$, and corresponding factors for lowest and highest filled graph connectivity, respectively.

The height of the elimination tree is thus

$$\begin{aligned} h &= \log_p((p-1)n+1) \\ &= (\log_p 2)\log((p-1)n+1) \\ &= (\log_p 2)\log n + O(1). \end{aligned}$$

- $|L|$

For lowest connectivity, there are $n-1$ edges in the filled graph, therefore

$$|L| = 2n-1.$$

For highest connectivity, there are dp^d edges connecting the nodes at depth d to their ancestors. The total number of edges is thus

$$\begin{aligned} e &= \sum_{d=0}^{h-1} dp^d \\ &= \left(h-1-\frac{1}{p-1}\right)\left(\frac{p^h-1}{p-1}-1\right)+\frac{h-1}{p-1}+h-1 \\ &= (\log_p 2)n\log n + O(n). \end{aligned}$$

We then have

$$|L| = (\log_p 2)n\log n + O(n).$$

- M_2^L

For lowest connectivity, at most $p+2$ entries need to be stored, thus

$$M_2^L = p+2.$$

For highest connectivity, for left-looking and multifrontal factorization, we use a recursive technique that follows Liu's strategy for the minimization of the core [33, 34]. We define the following:

- $M_2^L(d)$: the core required for a subtree rooted at depth d;

- $M_2^{L-}(d)$: the core required for a subtree rooted at depth d, immediately before the subtree's root is processed;
- $M_2^{L+}(d)$: the core required for a subtree rooted at depth d, immediately after the subtree's root is processed.

We can compute $M_2^{L-}(d)$ and $M_2^{L+}(d)$. Just before the subtree's root is processed, $(d+1)$ entries must be stored for each node in the subtree. As soon as the root is processed, only d entries must be stored for each node in the subtree. The number of nodes in a subtree rooted at depth d is

$$\begin{aligned}\sum_{i=d}^{h-1} p^{i-d} &= \frac{1}{p^d}\sum_{i=d}^{h-1} p^i \\ &= \frac{1}{p^d}\left(\sum_{i=0}^{h-1} p^i - \sum_{i=0}^{d-1} p^i\right) \\ &= \frac{1}{p^d}\left(\frac{p^h-1}{p-1} - \frac{p^d-1}{p-1}\right) \\ &= \frac{p^h-p^d}{p^d(p-1)}.\end{aligned}$$

We thus have

$$\begin{aligned}M_2^{L-}(d) &= (d+1)\frac{p^h-p^d}{p^d(p-1)}. \\ M_2^{L+}(d) &= d\frac{p^h-p^d}{p^d(p-1)}.\end{aligned}$$

Note that $M_2^{L-}(0) = n$ and $M_2^{L-}(h-1) = h$. We can now write the following recursion

$$\begin{aligned}M_2^L(d) &= \max\{M_2^{L-}(d),\ (p-1)M_2^{L+}(d+1) + M_2^L(d+1)\},\ 0 \le d < h-1, \\ M_2(h-1) &= M_2^{L-}(n, h-1).\end{aligned}$$

Obviously, $M_2^L = M_2^L(0)$, thus in order to compute M_2^L we need to solve the recursion above. We can expand it as following:

$$\begin{aligned}M_2^L &= M_2^L(0) \\ &= \max\{M_2^{L-}(0),\end{aligned}$$

$$(p-1)M_2^{L+}(1) + M_2^{L-}(1),$$

$$(p-1)M_2^{L+}(1) + (p-1)M_2^{L+}(2) + M_2^{L-}(2),$$

$$\vdots$$

$$(p-1)M_2^{L+}(1) + \cdots + (p-1)M_2^{L+}(h-1) + M_2^{L-}(h-1)\}.$$

The maximum value is reached somewhere in the middle. Instead of computing it precisely, we determine lower and upper bounds.

A lower bound can be immediately obtained from the first row of the expanded recursion:

$$\begin{aligned} M_2^L &> M_2^{L-}(0) \\ &= n. \end{aligned}$$

A simple upper bound can be obtained by considering that, for each depth d, we do not need to store more than $pM_2^{L-}(d)$ entries. We can be a little more precise though. Consider replacing each occurrence of $M_2^{L+}(d)$ in the expanded recursion by $M_2^{L-}(d)$, knowing that $M_2^{L+}(d) < M_2^{L-}(d)$. Also, on the last row,

$$\begin{aligned} M_2^{L-}(h-1) &= h \\ &\leq n \\ &= M_2^{L-}(0) \\ &< (p-1)M_2^{L-}(0). \end{aligned}$$

Thus, if we also replace the last $M_2^{L-}(h-1)$ on the last row by $(p-1)M_2^{L-}(0)$, the maximum is reached on the last line and we have

$$\begin{aligned} M_2^L &< \sum_{d=0}^{h-1}(p-1)M_2^{L-}(d) \\ &= \sum_{d=0}^{h-1}(p-1)\frac{(d+1)(p^h-p^d)}{p^d(p-1)} \\ &= p^h\sum_{d=0}^{h-1}\frac{d+1}{p^d} - \sum_{d=0}^{h-1}(d+1) \end{aligned}$$

$$
\begin{aligned}
&= p^{h+1}\sum_{d=0}^{h-1}\frac{d+1}{p^{d+1}} - \sum_{d=0}^{h-1}(d+1) \\
&= p^{h+1}\sum_{i=1}^{h}\frac{i}{p^i} - \sum_{i=1}^{h} i \\
&= p^{h+1}\frac{p}{(p-1)^2}\left(1 - \frac{(p-1)h+p}{p^{h+1}}\right) - \frac{h(h+1)}{2} \\
&= \frac{p^{h+2}}{(p-1)^2} + O(h^2) \\
&= \frac{p^2 p^h}{(p-1)^2} + O(h^2) \\
&= \frac{p^2(p-1)n + p^2}{(p-1)^2} + O((\log n)^2) \\
&= \frac{p^2}{p-1}n + O((\log n)^2).
\end{aligned}
$$

- M_2^R

For lowest connectivity, at most three entries need to be stored, thus

$$M_2^R = 3.$$

For highest connectivity we need to add the entries along the longest path, thus

$$
\begin{aligned}
M_2^R &= \sum_{d=0}^{h-1}(d+1) \\
&= \sum_{i=1}^{h} i \\
&= \frac{h(h+1)}{2} \\
&= (\log_p 2)^2(\log n)^2 + O(\log n).
\end{aligned}
$$

- M_2^M

For lowest connectivity, an update matrix has one entry (except at the root) and a maximum of $(p-1)(h-1)+1$ update matrices can be stacked, thus

$$
\begin{aligned}
|U| &= (p-1)(h-1)+1 \\
&= (p-1)(((\log_p 2)\log((p-1)n+1) - 1) + 1 \\
&= (p-1)(\log_p 2)\log n + O(1).
\end{aligned}
$$

The largest frontal matrix has three entries, thus

$$|F| = 3.$$

For highest connectivity, we use the recursive technique. We define the following:

- $u(d)$: the number of entries in an update matrix at depth d;
- $U(d)$: the maximum number of entries in the update stack, corresponding to a subtree rooted at depth d.

We can now write the following recursion

$$\begin{aligned} U(d) &= \max\{u(d),\ (p-1)u(d+1) + U(d+1)\},\ 0 \le d < h-1, \\ U(h-1) &= u(h-1). \end{aligned}$$

This time, $|U| = U(0)$. We can expand the recursion as following:

$$\begin{aligned} |U| &= U(0) \\ &= \max\{u(0), \\ &\qquad pu(1), \\ &\qquad (p-1)u(1) + pu(2), \\ &\qquad \vdots \\ &\qquad (p-1)u(1) + \cdots + pu(h-1)\}. \end{aligned}$$

Note that an update matrix that corresponds to a node at depth d has $d(d+1)/2$ entries, therefore the number of entries in an update matrix grows with the depth. The maximum stack size is then reached right after processing the last leaf, thus

$$\begin{aligned} |U| &= \sum_{d=1}^{h-1}(p-1)\frac{d(d+1)}{2} + \frac{h(h-1)}{2} \\ &= \frac{p-1}{2}\left(\sum_{d=1}^{h-1} d^2 + \sum_{d=1}^{h-1} d\right) + \frac{h(h-1)}{2} \\ &= \frac{p-1}{2}\left(\frac{h(h-1)(2h-1)}{6} + \frac{h(h-1)}{2}\right) + \frac{h(h-1)}{2} \end{aligned}$$

$$
\begin{aligned}
&= \frac{(p-1)h^3}{6} + O(h^2) \\
&= \frac{(p-1)(\log_p 2)^3}{6}(\log n)^3 + O((\log n)^2).
\end{aligned}
$$

The maximum number of entries in a frontal matrix is

$$
\begin{aligned}
|F| &= \frac{h(h+1)}{2} \\
&= \frac{h^2}{2} + O(h) \\
&= \frac{(\log_p 2)^2}{2}(\log n)^2 + O(\log n). \quad \square
\end{aligned}
$$

Star Elimination Tree

Figure 17 shows a star elimination tree and the corresponding factor. In this case the filled graph is the same as the elimination tree, thus the connectivity is fixed.

Lemma 3 *If the elimination tree is a star, we have the following:*

- $|L| = \Theta(n)$;
- $M_2^L = \Theta(n)$;
- $M_2^R = \Theta(1)$;
- $M_2^M = \Theta(n)$.

Proof

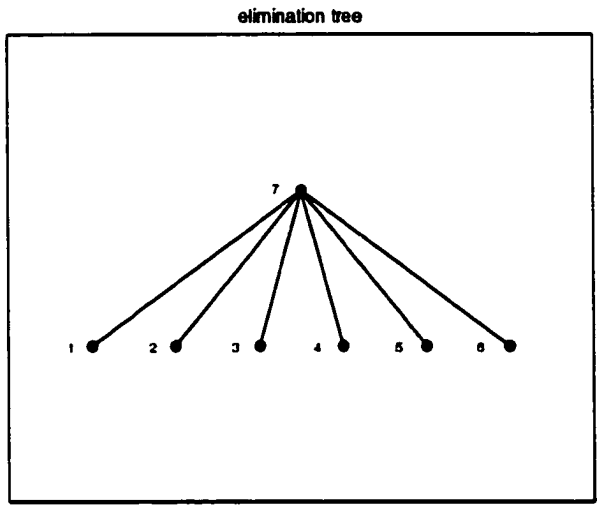
There are just two levels of nodes:

$$h = 2.$$

- $|L|$

  There are n edges in the filled graph, thus

$$|L| = 2n - 1.$$



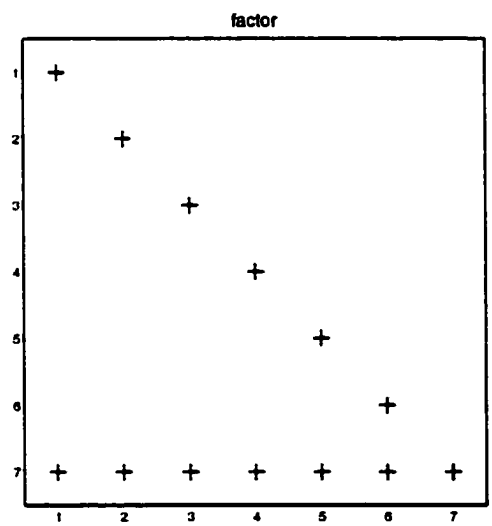


Figure 17: Star elimination tree and corresponding factor.

- M_2^L

  At most $n+1$ entries need to be stored, thus

$$M_2^L = n+1.$$

- M_2^R

  At most three entries need to be stored, thus

$$M_2^R = 3.$$

- M_2^M

  An update matrix has one entry (except at the root) and a maximum of $n-1$ update matrices can be stacked, thus

$$|U| = n-1.$$

  A frontal matrix can have at most three entries, thus

$$|F| = 3. \quad \square$$

The complexity results for the balanced elimination trees are summarized in Table 3.

branching	connectivity	$\|L\|$	M_2^L	M_2^R	M_2^M
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		$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$

Table 3: Factor and core complexity for the balanced elimination trees.

Unbalanced p-ary Elimination Tree

This represents the most unbalanced p-ary tree, in which interior nodes have p children, out of which $p-1$ are leaves. As before, p is a positive integer larger than one. Figures 18 and 19 show two isomorphic unbalanced p-ary elimination trees with $p = 2$ (binary), called arrow-tail and arrow-head, respectively, and the corresponding factors for lowest and highest connectivity.

Lemma 4 *If the elimination tree is unbalanced p-ary, we have the following:*

- *for lowest connectivity,* $|L| = \Theta(n)$*; for highest connectivity,* $|L| = \Theta(n^2)$*;*
- *for lowest connectivity,* $M_2^L = \Theta(1)$*; for highest connectivity,* $M_2^L = \Theta(n^2)$*;*
- *for lowest connectivity,* $M_2^R = \Theta(1)$*; for highest connectivity,* $M_2^R = \Theta(n^2)$*;*
- *for arrow-tail: for lowest connectivity,* $M_2^M = \Theta(1)$*; for highest connectivity,* $M_2^M = \Theta(n^2)$*;*
- *for arrow-head: for lowest connectivity,* $M_2^M = \Theta(n)$*; for highest connectivity,* $M_2^M = \Theta(n^3)$.

Proof

At depth $d = 0$ there is a single node and at depth $d > 0$ there are p nodes, therefore the total number of nodes is

$$\begin{aligned} n &= 1 + p(h-1) \\ &= ph - p + 1. \end{aligned}$$

The height of the elimination tree is thus

$$\begin{aligned} h &= \frac{n+p-1}{p} \\ &= \frac{n}{p} + O(1). \end{aligned}$$

- $|L|$

  For lowest connectivity, there are $n-1$ edges in the filled graph, thus

$$|L| = 2n - 1.$$

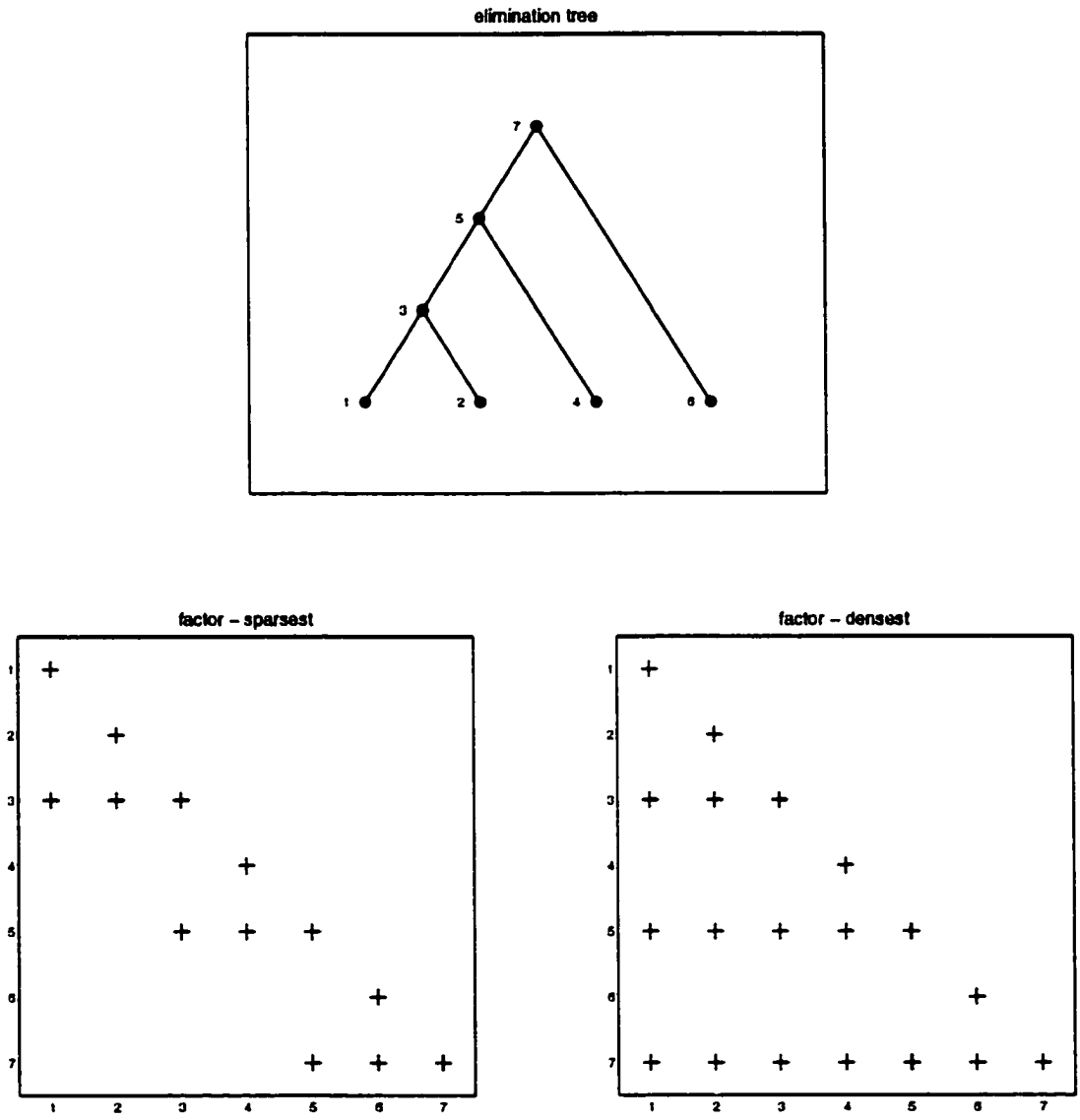


Figure 18: Arrow-tail p-ary elimination tree with $p = 2$, and corresponding factors for lowest and highest filled graph connectivity, respectively.

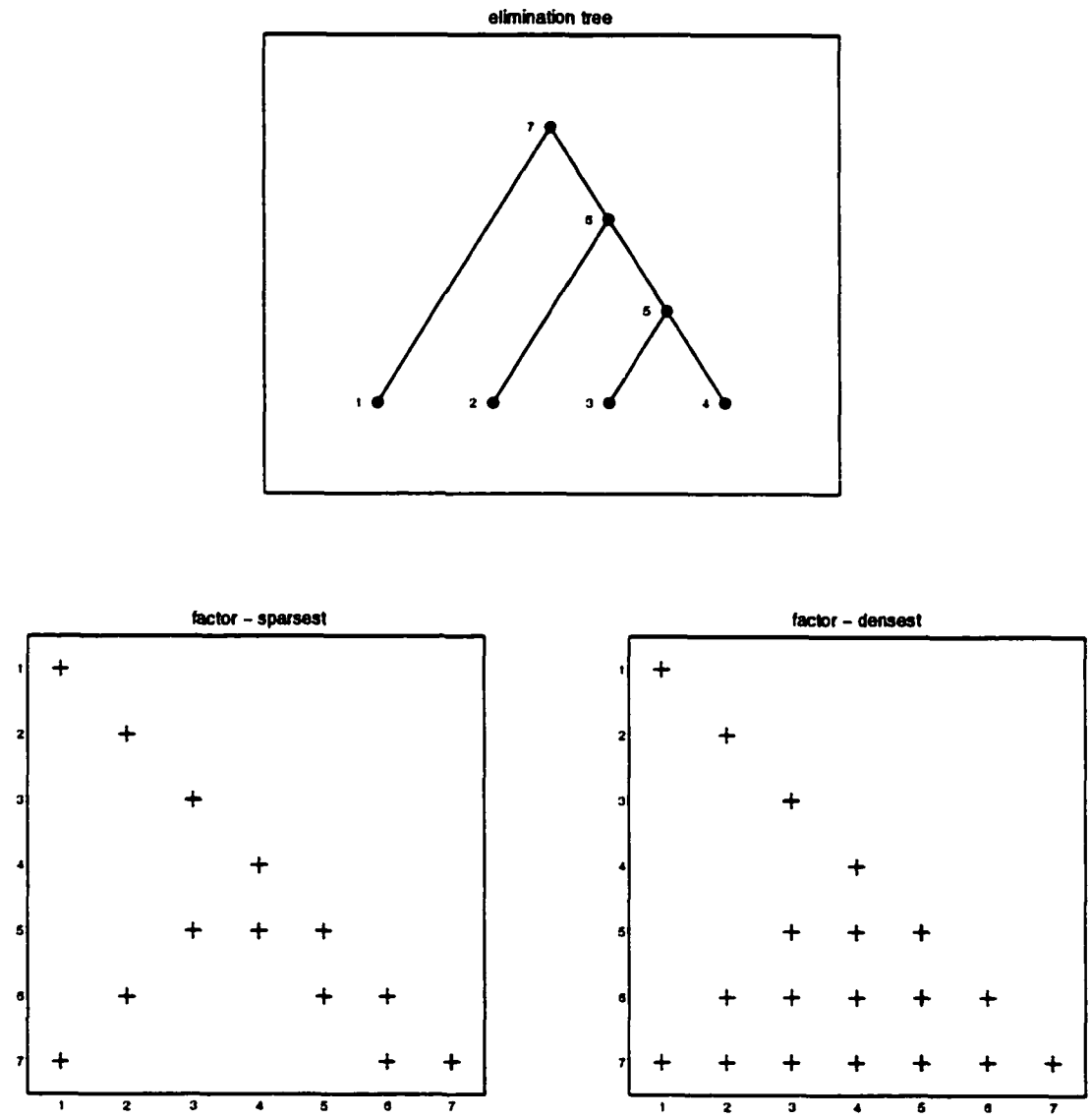


Figure 19: Arrow-head p-ary elimination tree with $p = 2$, and corresponding factors for lowest and highest filled graph connectivity, respectively.

For highest connectivity, there are pd edges connecting the nodes at depth d to their ancestors. The total number of edges is thus

$$\begin{aligned} e &= \sum_{d=1}^{h-1} pd \\ &= \frac{ph(h-1)}{2} \\ &= \frac{ph^2}{2} + O(h) \\ &= \frac{n^2}{2p} + O(n). \end{aligned}$$

We then have

$$|L| = \frac{n^2}{2p} + O(n).$$

- M_2^L

For lowest connectivity, at most $p+2$ entries need to be stored, thus

$$M_2^L = p + 2.$$

For highest connectivity we find the maximum value of the area that corresponds to the number of entries that need to be stored at each step, as we did for the path elimination tree. Again, for convenience, we switch from discrete to continuous values.

For the arrow-tail elimination tree the area can be approximated as $n(n-x)/2$, x being the distance from 0 and ranging from 0 to n (half of the area from the path elimination tree). The maximum is reached at $x = n/2$ and its value is $n^2/8$. As a consequence, we can write

$$M_2^L = \frac{n^2}{8} + O(n).$$

For the arrow-head elimination tree the area can be initially approximated as $x^2/2$, x being the distance from 0 and ranging from 0 to $n/2$. The maximum is reached at $x = n/2$ and its value is $n^2/8$. After that, the area becomes $(n/2-x)^2/2 + 2x(n/2-x)$, x being the distance from $n/2$ and ranging from 0 to $n/2$. The maximum is reached at $x = n/6$ and its value is $n^2/6$. The overall maximum is thus $n^2/6$ and we can write

$$M_2^L = \frac{n^2}{6} + O(n).$$

- M_2^R

For lowest connectivity, at most three entries need to be stored, thus

$$M_2^R = 3.$$

For highest connectivity we need to add the entries along the longest path, thus

$$\begin{aligned} M_2^R &= \sum_{d=0}^{h-1}(d+1) \\ &= \sum_{i=1}^{h} i \\ &= \frac{h(h+1)}{2} \\ &= \frac{h^2}{2} + O(h) \\ &= \frac{n^2}{2p^2} + O(n). \end{aligned}$$

- M_2^M, arrow-tail

For lowest connectivity, an update matrix has one entry (except at the root) and a maximum of p matrices can be stacked, thus

$$|U| = p.$$

The largest frontal matrix has three entries, thus

$$|F| = 3.$$

For highest connectivity, an update matrix that corresponds to a node at depth d has $d(d+1)/2$ entries. At most p update matrices can be stacked and the largest ones are at depth $d = h-1$, thus

$$\begin{aligned} |U| &= 2p\frac{h(h-1)}{2} \\ &= ph(h-1) \\ &= ph^2 + O(h) \\ &= \frac{n^2}{p} + O(n). \end{aligned}$$

- M_2^M, arrow-head

For lowest connectivity, an update matrix has one entry (except at the root) and a maximum of $(p-1)(h-1)+1$ matrices can be stacked, thus

$$\begin{aligned} |U| &= (p-1)(h-1)+1 \\ &= (p-1)\left(\frac{n+p-1}{p}-1\right)+1 \\ &= \frac{p-1}{p}n+O(1). \end{aligned}$$

The largest frontal matrix has three entries, thus

$$|F| = 3.$$

For highest connectivity, an update matrix that corresponds to a node at depth d has $d(d+1)/2$ entries. The maximum stack size is reached right after processing the last leaf, thus

$$\begin{aligned} |U| &= \sum_{d=1}^{h-1}(p-1)\frac{d(d+1)}{2}+\frac{h(h-1)}{2} \\ &= \frac{p-1}{2}\left(\sum_{d=1}^{h-1}d^2+\sum_{d=1}^{h-1}d\right)+\frac{h(h-1)}{2} \\ &= \frac{p-1}{2}\left(\frac{h(h-1)(2h-1)}{6}+\frac{h(h-1)}{2}\right)+\frac{h(h-1)}{2} \\ &= \frac{(p-1)h^3}{6}+O(h^2) \\ &= \frac{p-1}{6p^3}n^3+O(n^2). \end{aligned}$$

The maximum number of entries in a frontal matrix is

$$\begin{aligned} |F| &= \frac{h(h+1)}{2} \\ &= \frac{h^2}{2}+O(h) \\ &= \frac{n^2}{2p^2}+O(n). \quad \square \end{aligned}$$

Generalized Star Elimination Tree

We can generalize the star tree by assuming that:

- q out of n nodes from a clique in the filled graph, where $q = n^\alpha$ and $0 < \alpha < 1$;

- each one of the remaining $n-q$ nodes is connected to each one of the q nodes from the clique;
- the q-node clique is the root of the supernodal elimination tree and the remaining $n-q$ nodes are the leaves.

Figure 20 shows a generalized star elimination tree and the corresponding factor.

Lemma 5 *If the elimination tree is a generalized star, we have the following:*

- $|L| = \Theta(n^{1+\alpha})$;
- $M_2^L = \Theta(n^{1+\alpha})$;
- $M_2^R = \Theta(n^{2\alpha})$;
- $M_2^M = \Theta(n^{1+2\alpha})$.

Proof

The height of the elimination tree is

$$h = q+1.$$

- $|L|$

  There are $q(q-1)/2$ edges in the clique and $(n-q)q$ edges connecting the remaining $n-q$ nodes to the nodes in the clique, thus

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

- M_2^L

  The maximum number of entries in core is reached just before the first node in the clique is processed, thus

$$\begin{aligned} M_2^L &= (n-q)q + q \\ &= n^{1+\alpha} + O(n^{2\alpha}). \end{aligned}$$

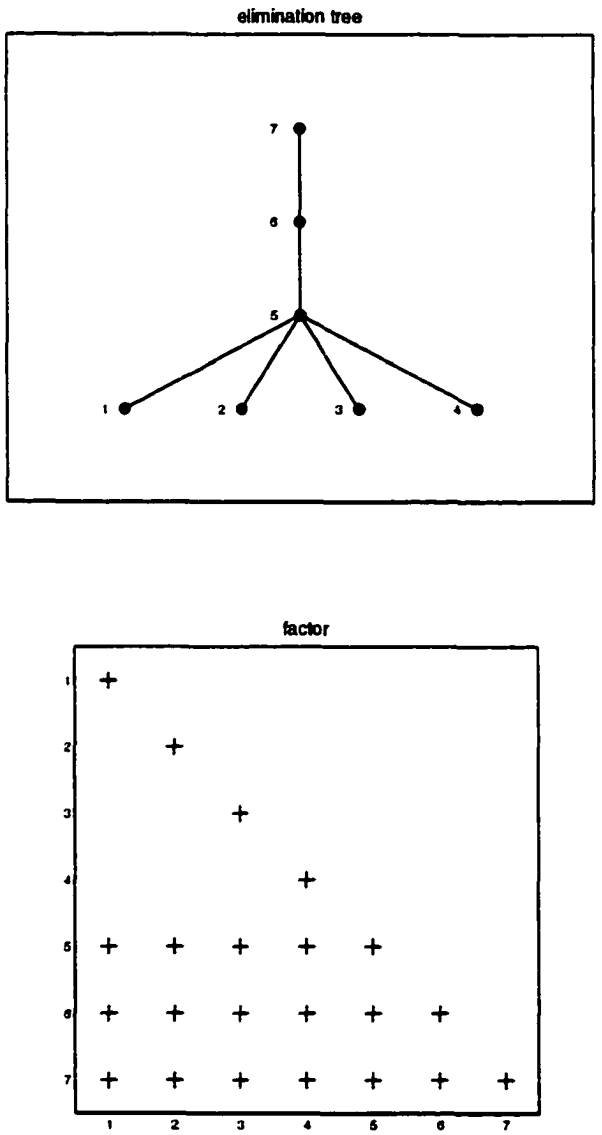


Figure 20: Generalized star elimination tree and corresponding factor.

- M_2^R

  We need to add the entries along the longest path, thus

$$\begin{aligned} M_2^R &= \frac{h(h+1)}{2} \\ &= \frac{(q+1)(q+2)}{2} \\ &= \frac{n^{2\alpha}}{2} + O(n^{\alpha}). \end{aligned}$$

- M_2^M

  For each leaf, the update matrix has $q(q+1)/2$ entries. At most $n-q$ update matrices can be stacked, thus

$$\begin{aligned} |U| &= (n-q)\frac{q(q+1)}{2} \\ &= \frac{n^{1+2\alpha}}{2} + O(n^{3\alpha}). \end{aligned}$$

  The largest frontal matrix corresponds to a leaf, thus

$$\begin{aligned} |F| &= \frac{(q+1)(q+2)}{2} \\ &= \frac{n^{2\alpha}}{2} + O(n^{\alpha}). \quad \square \end{aligned}$$

As an example, consider $q = n^{1/2}$. Then we have

$$\begin{aligned} |L| &= \Theta(n^{3/2}), \\ M_2^L &= \Theta(n^{3/2}), \\ M_2^R &= \Theta(n), \\ M_2^M &= \Theta(n^2). \end{aligned}$$

The complexity results for the unbalanced elimination trees and for the generalized star are summarized in Table 4.

Supernodal Elimination Tree for r Dimensions

We turn now to a model that idealizes the trees that correspond to grids with r dimensions ordered with nested dissection, with r integer, $r \geq 2$. These trees generalize the common trees that corre-

problem	connectivity	$\|L\|$	M_2^L	M_2^R	M_2^M
arrow-tail	best	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
	worst	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
arrow-head	best	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
	worst	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^3)$
generalized star		$\Theta(n^{1+\alpha})$	$\Theta(n^{1+\alpha})$	$\Theta(n^{2\alpha})$	$\Theta(n^{1+2\alpha})$

Table 4: Factor and core complexity for the unbalanced elimination trees and for the generalized star.

spond to 2-d and 3-d applications. The difference between the actual trees and the models is within constant factors.

A supernode corresponds to the set of separators that form a hypercross in an actual r-d problem. We assume that at depth D there are $r(k/2^D)^{r-1}$ nodes in a supernode, where, for convenience, $k = 2^l$ and l is a nonnegative integer.

Each interior supernode has $p = 2^r$ children. At depth D there are thus $p^D = 2^{rD}$ supernodes.

A supernode is connected to at most two of its ancestors. For each such connection a clique is formed with $r(k/2^D)^{r-1}$ nodes from that supernode and $r(k/2^D)^{r-1}$ nodes from its ancestor.

Lemma 6 *For the idealized r-d supernodal elimination tree described above, we have the following:*

- *if $r = 2$ then $|L| = \Theta(n \log n)$; if $r > 2$ then $|L| = \Theta(n^{2(r-1)/r})$;*
- $M_2^L = \Theta(n^{2(r-1)/r})$;
- $M_2^R = \Theta(n^{2(r-1)/r})$;
- $M_2^M = \Theta(n^{2(r-1)/r})$.

Proof

- $|L|$

We need to compute n, the number of nodes, and e, the number of edges. For the number of nodes we can write

$$\begin{aligned} n &= \sum_{D=0}^{H-1} p^D r \left(\frac{k}{2^D}\right)^{r-1} \\ &= \sum_{D=0}^{l} r 2^{rD} \frac{k^{r-1}}{2^{(r-1)D}} \\ &= rk^{r-1} \sum_{D=0}^{l} 2^D \\ &= rk^{r-1}(2^{l+1}-1) \\ &= rk^{r-1}(2k-1) \\ &= 2rk^r - rk^{r-1} \\ &= 2rk^r + O(k^{r-1}). \end{aligned}$$

We can thus approximate n as $2rk^r$ and k as $(n/(2r))^{1/r}$. Note that in real applications we would have $n = k^r$.

We bound the number of edges. A lower bound is given by the number of edges within the supernodes. At depth D, there are $(r(k/2^D)^{r-1}(r(k/2^D)^{r-1}-1))/2$ edges within a supernode. We can thus write

$$\begin{aligned} e &> \sum_{D=0}^{H-1} p^D \frac{1}{2} r \left(\frac{k}{2^D}\right)^{r-1} \left(r\left(\frac{k}{2^D}\right)^{r-1} - 1\right) \\ &= \frac{1}{2} \sum_{D=0}^{l} 2^{rD} r \left(\frac{k}{2^D}\right)^{r-1} \left(r\left(\frac{k}{2^D}\right)^{r-1} - 1\right) \\ &= \frac{1}{2} \left(\sum_{D=0}^{l} r^2 2^{rD} \left(\frac{k}{2^D}\right)^{2(r-1)} - \sum_{D=0}^{l} r 2^{rD} \left(\frac{k}{2^D}\right)^{r-1}\right) \\ &= \frac{r^2}{2} \sum_{D=0}^{l} 2^{rD} \frac{k^{2(r-1)}}{2^{2(r-1)D}} - \frac{r}{2} \sum_{D=0}^{l} 2^{rD} \frac{k^{r-1}}{2^{(r-1)D}} \\ &= \frac{r^2 k^{2(r-1)}}{2} \sum_{D=0}^{l} \frac{1}{2^{(r-2)D}} - \frac{rk^{r-1}}{2} \sum_{D=0}^{l} 2^D \\ &= \frac{r^2 k^{2(r-1)}}{2} \sum_{D=0}^{l} \frac{1}{2^{(r-2)D}} - \frac{rk^{r-1}}{2}(2^{l+1}-1) \\ &= \frac{r^2 k^{2(r-1)}}{2} \sum_{D=0}^{l} \frac{1}{2^{(r-2)D}} - \frac{rk^{r-1}}{2}(2k-1) \end{aligned}$$

$$= \frac{r^2k^{2(r-1)}}{2}\sum_{D=0}^{l}\frac{1}{2^{(r-2)D}} + O(k^r).$$

If $r = 2$ this becomes

$$\begin{aligned} e &> 2k^2\sum_{D=0}^{l} 1 + O(k^2) \\ &= 2k^2(l+1) + O(k^2) \\ &= 2k^2(\log k + 1) + O(k^2) \\ &= 2k^2\log k + O(k^2) \\ &= \frac{n}{2}\log\left(\frac{n}{4}\right)^{\frac{1}{2}} + O(n) \\ &= \frac{n}{4}(\log n - 2) + O(n) \\ &= \frac{1}{4}n\log n + O(n). \end{aligned}$$

If $r > 2$ we have

$$\begin{aligned} e &> \frac{r^2k^{2(r-1)}}{2}\frac{1}{2^{(r-2)l}}\frac{2^{(r-2)(l+1)}-1}{2^{r-2}-1} + O(k^r) \\ &= \frac{r^2k^{2(r-1)}}{2}\frac{1}{k^{r-2}}\frac{2^{r-2}k^{r-2}-1}{2^{r-2}-1} + O(k^r) \\ &= \frac{r^2}{2}\frac{2^{r-2}}{2^{r-2}-1}k^{2(r-1)} + O(k^r) \\ &= \frac{r^2 2^{r-3}}{2^{r-2}-1}k^{2(r-1)} + O(k^r) \\ &= \frac{r^2 2^{r-3}}{2^{r-2}-1}\frac{1}{(2r)^{2(r-1)/r}}n^{2(r-1)/r} + O(n). \end{aligned}$$

For an upper bound, remember that a supernode is connected to at most two of its ancestors. Again, there are $(r(k/2^D)^{r-1}(r(k/2^D)^{r-1}-1))/2$ edges within a supernode at depth D. In addition, there are at most $2r^2(k/2^D)^{2(r-1)}$ edges connecting a supernode at depth D to its ancestors. We can thus write

$$\begin{aligned} e &< \sum_{D=0}^{H-1} p^D\left(\frac{1}{2}r\left(\frac{k}{2^D}\right)^{r-1}\left(r\left(\frac{k}{2^D}\right)^{r-1}-1\right) + 2r^2\left(\frac{k}{2^D}\right)^{2(r-1)}\right) \\ &= \frac{5r^2k^{2(r-1)}}{2}\sum_{D=0}^{l}\frac{1}{2^{(r-2)D}} + O(k^r). \end{aligned}$$

We skipped the intermediate steps because they are similar to those from the lower bound. If

$r = 2$ this becomes

$$\begin{aligned} e &< \frac{10k^2}{2}\sum_{D=0}^{l} 1 + O(k^2) \\ &= \frac{5}{8}n\log n + O(n). \end{aligned}$$

If $r > 2$ we have

$$\begin{aligned} e &< \frac{5r^2k^{2(r-1)}}{2}\frac{1}{2^{(r-2)l}}\frac{2^{(r-2)(l+1)}-1}{2^{r-2}-1} + O(k^r) \\ &= \frac{5r^2 2^{r-3}}{2^{r-2}-1}\frac{1}{(2r)^{2(r-1)/r}}n^{2(r-1)/r} + O(n). \end{aligned}$$

- M_2^L

For the lower bound we can consider the factorization of the root, thus

$$\begin{aligned} M_2^L &> \frac{k^{2(r-1)}}{4} + O(k^{r-1}) \\ &= \frac{1}{4}\frac{1}{(2r)^{2(r-1)/r}}n^{2(r-1)/r} + O(n^{(r-1)/r}). \end{aligned}$$

For an upper bound we use the recursive technique, adapted to supernodal elimination trees. This time we get an upper bound for $M_2^{L-}(D)$, then bound M_2^L by adding $pM_2^{L-}(D)$ for each depth D. For $M_2^{L-}(D)$ it is not difficult to verify that from a subtree rooted at depth D, the number of supernodes from depth i, $i \geq D$, that update the subtree's root or it's ancestors is bounded from above by $4r2^{(r-1)(i-D)}$. The contribution from each such supernode is bounded from above by $3r^2(k/2^i)^{2(r-1)}$. We then have

$$\begin{aligned} M_2^{L-}(D) &< \sum_{i=D}^{H-1} 4r2^{(r-1)(i-D)}3r^2\left(\frac{k}{2^i}\right)^{2(r-1)} \\ &= 12r^3\sum_{i=D}^{H-1}\frac{2^{(r-1)i}}{2^{(r-1)D}}\frac{k^{2(r-1)}}{2^{2(r-1)i}} \\ &= \frac{12r^3}{2^{(r-1)D}}k^{2(r-1)}\sum_{i=D}^{H-1}\frac{1}{2^{(r-1)i}} \\ &= \frac{12r^3}{2^{(r-1)D}}k^{2(r-1)}\left(\sum_{i=0}^{H-1}\frac{1}{2^{(r-1)i}} - \sum_{i=0}^{D-1}\frac{1}{2^{(r-1)i}}\right) \\ &= \frac{12r^3}{2^{(r-1)D}}k^{2(r-1)}\left(\frac{1}{2^{(r-1)(H-1)}}\frac{2^{(r-1)H}-1}{2^{r-1}-1}\frac{1}{2^{(r-1)(D-1)}}\frac{2^{(r-1)D}-1}{2^{r-1}-1}\right) \\ &= \frac{12r^3}{2^{(r-1)D}}k^{2(r-1)}\frac{1}{2^{r-1}-1}\left(\frac{1}{2^{(r-1)(D-1)}} - \frac{1}{2^{(r-1)(H-1)}}\right) \end{aligned}$$

$$= \frac{12r^3}{2^{r-1}-1}k^{2(r-1)}\left(\frac{1}{2^{(r-1)(2D-1)}} - \frac{1}{2^{(r-1)(D+H-1)}}\right).$$

Then we can write

$$\begin{aligned}
M_2^L &< \sum_{D=0}^{H-1} pM_2^{L-}(D) \\
&= \frac{12r^3 2^r}{2^{r-1}-1}k^{2(r-1)}\sum_{D=0}^{H-1}\frac{1}{2^{(r-1)(2D-1)}} - \frac{12r^3 2^r}{2^{r-1}-1}k^{2(r-1)}\sum_{D=0}^{H-1}\frac{1}{2^{(r-1)(D+H-1)}} \\
&= \frac{12r^3 2^r}{2^{r-1}-1}k^{2(r-1)}2^{r-1}\sum_{D=0}^{H-1}\frac{1}{2^{2(r-1)D}} - \frac{12r^3 2^r}{2^{r-1}-1}k^{2(r-1)}\frac{1}{2^{(r-1)(H-1)}}\sum_{D=0}^{H-1}\frac{1}{2^{(r-1)D}} \\
&= \frac{12r^3 2^{2r-1}}{2^{r-1}-1}k^{2(r-1)}\frac{1}{2^{2(r-1)l}}\frac{2^{2(r-1)(l+1)}-1}{2^{2(r-1)}-1} \\
&\quad -\frac{12r^3 2^r}{2^{r-1}-1}k^{2(r-1)}\frac{1}{2^{(r-1)l}}\frac{1}{2^{(r-1)l}}\frac{2^{(r-1)(l+1)}-1}{2^{(r-1)}-1} \\
&= \frac{12r^3 2^{2r-1}}{2^{r-1}-1}k^{2(r-1)}\frac{2^{2(r-1)}}{2^{r-1}-1}k^{2(r-1)} + O(k^{r-1}) \\
&= \frac{12r^3 2^{2r-1}}{2^{r-1}-1}k^{2(r-1)}\frac{2^{2(r-1)}}{2^{r-1}-1}\frac{1}{(2r)^{2(r-1)/r}}n^{2(r-1)/r} + O(n^{(r-1)/r}).
\end{aligned}$$

- M_2^R

For a lower bound we assume connectivity only within supernodes. The largest supernode lies at the root, thus we can write

$$\begin{aligned}
M_2^R &> \frac{rk^{r-1}(rk^{r-1}+1)}{2} \\
&= \frac{r^2k^{2(r-1)}}{2} + O(k^{r-1}) \\
&= \frac{r^2}{2}\frac{1}{(2r)^{2(r-1)/r}}n^{2(r-1)/r} + O(n^{(r-1)/r}).
\end{aligned}$$

For an upper bound we assume full connectivity along a path. We need to compute the height of the elimination tree first:

$$\begin{aligned}
h &= \sum_{D=0}^{H-1} r\left(\frac{k}{2^D}\right)^{r-1} \\
&= r\sum_{D=0}^{l}\frac{k^{r-1}}{2^{(r-1)D}} \\
&= rk^{r-1}\sum_{D=0}^{l}\frac{1}{2^{(r-1)D}} \\
&= rk^{r-1}\frac{1}{2^{(r-1)l}}\frac{2^{(r-1)(l+1)}-1}{2^{r-1}-1}
\end{aligned}$$

$$
\begin{aligned}
&= rk^{r-1}\frac{1}{k^{r-1}}\frac{2^{r-1}k^{r-1}-1}{2^{r-1}-1} \\
&= \frac{r2^{r-1}}{2^{r-1}-1}k^{r-1}+O(1) \\
&= \frac{r2^{r-1}}{2^{r-1}-1}\frac{1}{(2r)^{(r-1)/r}}n^{(r-1)/r}+O(1).
\end{aligned}
$$

Adding the entries along a path we get

$$
\begin{aligned}
M_2^R &< \frac{h(h+1)}{2} \\
&= \frac{h^2}{2}+O(h) \\
&= \frac{1}{2}\left(\frac{r2^{r-1}}{2^{r-1}-1}\right)^2\frac{1}{(2r)^{2(r-1)/r}}n^{2(r-1)/r}+O(n^{(r-1)/r}).
\end{aligned}
$$

- M_2^M

For a lower bound, consider an update matrix at depth $D = 1$. There are $k^{r-1}(k^{r-1}+1)/2$ entries in it, thus

$$
\begin{aligned}
|U| &> \frac{rk^{r-1}(rk^{r-1}+1)}{2} \\
&= \frac{r^2k^{2(r-1)}2}{+}O(k^{r-1}) \\
&= \frac{r^2}{2}\frac{1}{(2r)^{2(r-1)/r}}n^{2(r-1)/r}+O(n^{(r-1)/r}).
\end{aligned}
$$

Using the recursive technique, no more than p update matrices can be stacked from any particular depth. There are at most $2r(k/2^D)^{r-1}(2r(k/2^D)^{r-1}+1)/2$ entries in an update matrix at depth D, thus

$$
\begin{aligned}
|U| &< \sum_{D=0}^{H-1} p\frac{1}{2}2r\left(\frac{k}{2^D}\right)^{r-1}\left(2r\left(\frac{k}{2^D}\right)^{r-1}+1\right) \\
&= \frac{p}{2}\sum_{D=0}^{l}\left(4r^2\left(\frac{k}{2^D}\right)^{2(r-1)}+2r\left(\frac{k}{2^D}\right)^{r-1}\right) \\
&= \frac{2^r}{2}\left(\sum_{D=0}^{l}4r^2\frac{k^{2(r-1)}}{2^{2(r-1)D}}+\sum_{D=0}^{l}2r\frac{k^{r-1}}{2^{(r-1)D}}\right) \\
&= 4r^2 2^{r-1}k^{2(r-1)}\sum_{D=0}^{l}\frac{1}{2^{2(r-1)D}}+2r2^{r-1}k^{r-1}\sum_{D=0}^{l}\frac{1}{2^{(r-1)D}} \\
&= r^2 2^{r+1}k^{2(r-1)}\frac{1}{2^{2(r-1)l}}\frac{2^{2(r-1)(l+1)}-1}{2^{2(r-1)}-1}+r2^r k^{r-1}\frac{1}{2^{(r-1)l}}\frac{2^{(r-1)(l+1)}-1}{2^{r-1}-1} \\
&= r^2 2^{r+1}k^{2(r-1)}\frac{1}{k^{2(r-1)}}\frac{2^{2(r-1)}k^{2(r-1)}-1}{2^{2(r-1)}-1}+r2^r k^{r-1}\frac{1}{k^{r-1}}\frac{2^{(r-1)}k^{(r-1)}-1}{2^{r-1}-1}
\end{aligned}
$$

$$= r^2 2^{r+1} \frac{2^{2(r-1)}}{2^{2(r-1)}-1} k^{2(r-1)} + O(k^{r-1})$$
$$= r^2 2^{r+1} \frac{2^{2(r-1)}}{2^{2(r-1)}-1} \frac{1}{(2r)^{2(r-1)/r}} n^{2(r-1)/r} + O(n^{(r-1)/r}).$$

No more than $3rk^{r-1}(3rk^{r-1}+1)/2$ entries can be in a frontal matrix, thus

$$\begin{aligned} |F| &< \frac{3rk^{r-1}(3rk^{r-1}+1)}{2} \\ &= \frac{9}{2} r^2 k^{2(r-1)} + O(k^{r-1}) \\ &= \frac{9}{2} r^2 \frac{1}{(2r)^{2(r-1)/r}} n^{2(r-1)/r} + O(n^{(r-1)/r}). \quad \square \end{aligned}$$

As a consequence, for 2-d trees the complexity of the factor is $\Theta(n \log n)$ while the complexity of the core is $\Theta(n)$, for all three algorithms. For 3-d trees both the complexity of the factor and the complexity of the core are $\Theta(n^{4/3})$. This information is summarized in Table 5.

problem	$\|L\|$	M_2^L	M_2^R	M_2^M
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 5: Factor and core complexity for 2-d and 3-d trees.

3.3 COMPUTING THE CORE THROUGH SIMULATION

We have implemented simulation algorithms that compute the exact values of M_2^L, M_2^R and M_2^M. As an example, we show the algorithms that correspond to left-looking and right-looking factorization. Algorithms 14 and 15 compute M_2^L and M_2^R, respectively.

In order to keep track of the current number of entries within the core we use the *corecnt* variable, whose value is initially zero. Each time a column is read *corecnt* is incremented by the number of entries in that column. After a column is factored or updated *corecnt* is decremented by one, accounting for the fact that one entry can be discarded from the core.

```
1.    corecnt := 0;
2.    coresize := 0;
3.    for k := 1 to n begin
4.        corecnt := corecnt + |L_k|;
5.        if coresize < corecnt
6.            coresize := corecnt;
7.        for j in row[k] \ {k}
8.            corecnt := corecnt − 1;
9.        corecnt := corecnt − 1;
10.       end
11.   end
```

Algorithm 14: Simulation algorithm that computes M_2^L.

```
1.    corecnt := 0;
2.    coresize := 0;
3.    for j := 1 to n
4.        incore[j] := false;
5.    for j := 1 to n begin
6.        if incore[j] = false begin
7.            corecnt := corecnt + |L_j|;
8.            if coresize < corecnt
9.                coresize := corecnt;
10.           incore[j] := true;
11.       end
12.       corecnt := corecnt − 1;
13.       for k in col[j] \ {j} begin
14.           if incore[k] = false begin
15.               corecnt := corecnt + |L_k|;
16.               if coresize < corecnt
17.                   coresize := corecnt;
18.               incore[k] := true;
19.           end
20.           corecnt := corecnt − 1;
21.       end
22.   end
```

Algorithm 15: Simulation algorithm that computes M_2^R.

The minimum core size is computed in the *coresize* variable. Initially, its value is also zero, and it is updated each time *corecnt* becomes larger than *coresize*.

In Algorithm 15 we also use an array of flags, *incore*, in order to keep track of the columns that were read.

The time complexity of both algorithms is $\Theta(|L|)$. The space complexity depends on the implementation. If the data structures are column-based then the space complexity is $\Theta(|L|)$ as well. On the other hand, if we use the compressed supernode-based data structures then we can get a lower space complexity. For 2-d and 3-d problems ordered with nested dissection, for example, the space complexity becomes $\Theta(n)$.

We have computed the minimum core for various 2-d and 3-d problems, using the nested dissection ordering algorithm from MeTiS [27], as well as Liu's multiple minimum degree ordering algorithm [22, 31]. We show results obtained for 2-d and 3-d models, which are representative. For the models we actually used a geometrically perfect nested dissection order.

The results are shown in Figure 21. For each class of models we plot $|L|$, M_2^L, M_2^R and M_2^L as functions of n. We actually scale the data by n and we plot on a logarithmic scale.

Note that the minimum core is always smaller than the factor, for 2-d models the difference being quite significant. In addition, note the asymptotic difference between the minimum core and the factor for 2-d models. The factor plot has a logarithmic shape, while the minimum core has a constant shape. On the other hand, the minimum core grows as fast as the factor for 3-d models. This behavior correlates with our analysis of the r-d supernodal elimination trees.

Note also the difference between M_2^R and M_2^L on one side, and M_2^L on the other side. This suggests that, for R1/W1 factorization, we should use the right-looking or the multifrontal algorithm, rather than left-looking algorithm.

Although 2-d and 3-d problems represent a large class of applications, there are problems for which we can witness a very different behavior. As an example, we selected **ken13**, a problem of multicommodity flow in a network.

In Figure 22 we plot $|L|$, M_2^L, M_2^R and M_2^L for **ken13**, for both nested dissection and multiple

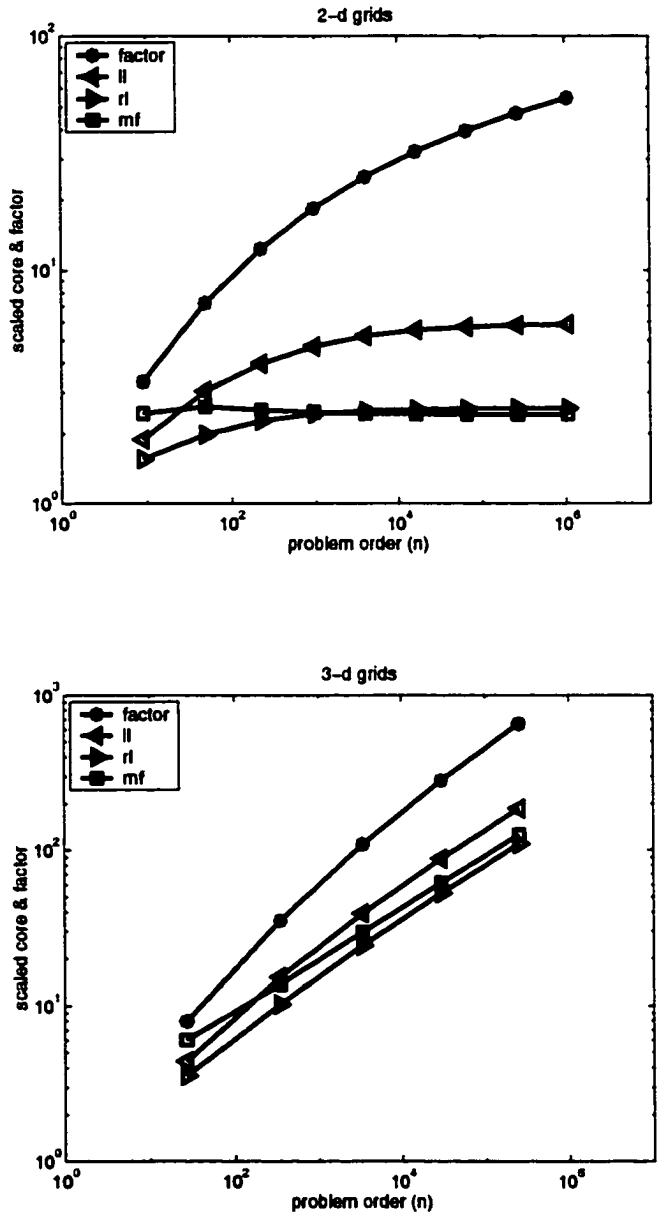


Figure 21: The minimum core size and the factor size for 2-d and 3-d model problems ordered with nested dissection.

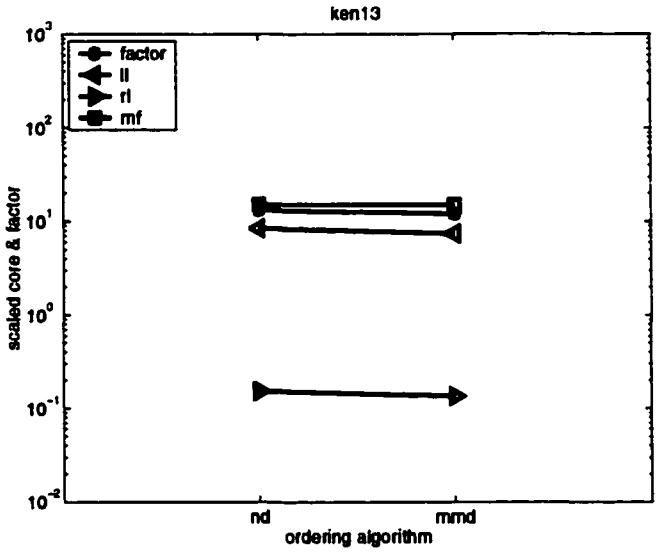


Figure 22: The minimum core size and the factor size for the ken13 problems ordered with nested dissection and multiple minimum degree.

minimum degree. Two observations need to be made here. First, M_2^R is much smaller than $|L|$, M_2^L and M_2^M. Second, M_2^M is larger than $|L|$.

The right-looking algorithm is thus by far the best choice for the a R1/W1 factorization in this case. On the other hand, the multifrontal factorization should really be avoided for such a problem.

It is not difficult to understand the results we obtained for ken13. In Figure 23 we show the elimination trees for this problem. These trees resemble the generalized star, for which we already know that right-looking factorization is the best alternative for R1/W1 factorization and that multifrontal factorization should be avoided because it requires too much temporary data.

Our results correlate with an observation made by Rothberg and Schreiber in their study of out-of-core factorization [47]. While discussing the multifrontal technique as a common out-of-core factorization algorithm they highlighted the fact that the update stack is typically much smaller than $|L|$ for 2-d problems and somewhat closer to $|L|$ for 3-problems. In addition, they mentioned the fact that the update stack can actually be larger than $|L|$ for some linear programming problems.

To conclude, for 2-d and 3-d applications the right-looking and multifrontal algorithms perform slightly better than the left-looking algorithm. On the other hand, there are applications for which multifrontal is a bad choice.

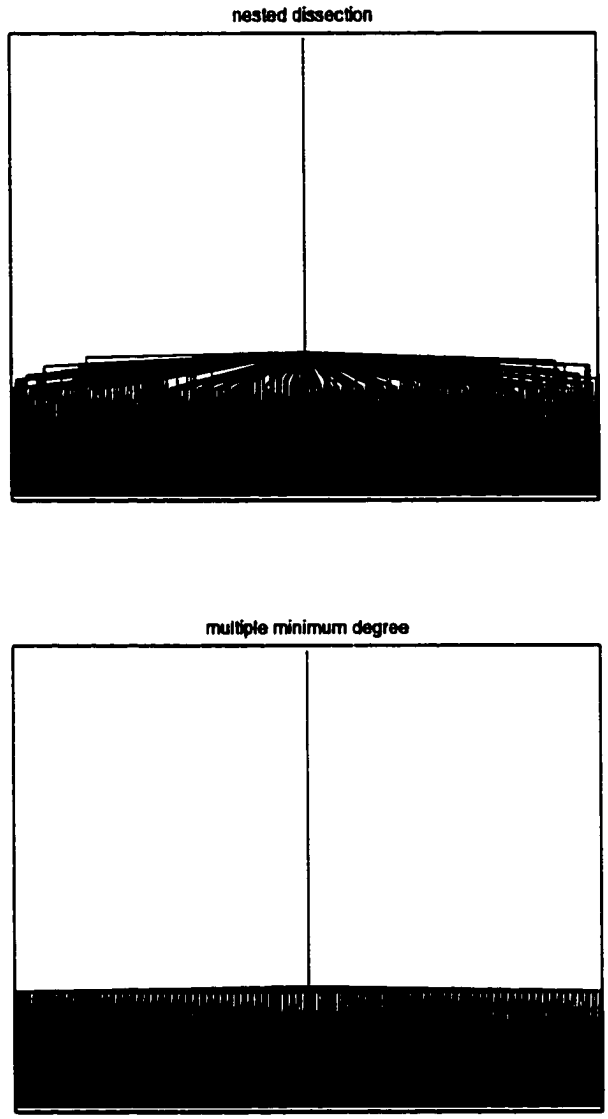


Figure 23: The elimination trees for **ken13** ordered with nested dissection and multiple minimum degree.

3.4 COMPUTING THE REORGANIZATIONS THROUGH SIMULATION

We have also implemented simulation algorithms that compute the number of reorganizations and the amount of data moved within the core due to the reorganizations. Liu has investigated the reorganizations for left-looking factorization [34] but he limited his study to the number of reorganizations, without looking the the amount of data moved within the core. We explore this issue further and we look at both left-looking and right-looking algorithms.

In order to count the number of core reorganizations we use the *reorganizecnt* variable, which is initialized as zero and then incremented by one each time the core is reorganized.

For the amount of data moved within the core we use two arrays of pointers *nextptr* and *crtptr*, both of size n. In both arrays, each pointer corresponds to a column. For convenience, we use a column-based implementation in this section, for a space complexity of $\Theta(|L|)$. In practice we use the more efficient supernode-based implementation.

There is thus a *colptr* array of size $n + 1$ that is part of the implementation of the nonzero structure of $|L|$. For column L_j, *nextptr*$[j]$ points to the first entry in L_j that is still needed in core, while *crtptr*$[j]$ points to the first entry in L_j that is still in core. For each column j, both *nextptr*$[j]$ and *crtptr*$[j]$ are initialized as *colptr*$[j]$.

When the data in core needs to be reorganized, the entries between *crtptr*$[j]$ and *nextptr*$[j]$ must be discarded and the entries between *nextptr*$[j]$ and *colptr*$[j + 1]$ must be moved. It is then easy to compute the amount of data moved within the core. We use the *shiftcnt* variable, which is initialized as zero. During each core reorganization this variable is incremented by *colptr*$[j + 1] -$ *nextptr*$[j]$, for each column L_j that is in core.

Algorithm 16 represents the simulation of the REORGANIZE task, which is based on the scheme presented above. Note the *corecnt* variable and the *incore* array, also present in the algorithms that compute the minimum core. We need to keep *corecnt* updated in order to determine when to reorganize. This is done by decrementing it by *nextptr*$[j] -$ *crtptr*$[j]$, for each column that is stored in core.

Algorithm 16 is called from Algorithms 17 and Algorithms 18, which simulate the whole core

```
1.    reorganizecnt := reorganizecnt + 1;
2.    for j := 1 to n
3.        if incore[j] = true begin
4.            corecnt := corecnt − (nextptr[j] − crtptr[j]);
5.            crtptr[j] := nextptr[j];
6.            if crtptr[j] < colptr[j + 1]
7.                shiftcnt := shiftcnt + (colptr[j + 1] − crtptr[j]);
8.            else
9.                incore[j] := false;
10.       end
```

Algorithm 16: Simulation algorithm for the REORGANIZE task.

```
1.    corecnt := 0;
2.    reorganizecnt := 0;
3.    shiftcnt := 0;
4.    for k := 1 to n begin
5.        incore[k] := false;
6.        crtptr[k] := colptr[k];
7.        nextptr[k] := colptr[k];
8.    end
9.    for k := 1 to n begin
10.       if |L_k| > coresize − corecnt
11.           simulate the REORGANIZE task;
12.       if |L_k| > coresize − corecnt
13.           ABORT();
14.       corecnt := corecnt + |L_k|;
15.       for j in row[k] \ {k}
16.           nextptr[j] := nextptr[j] + 1;
17.       nextptr[k] := nextptr[k] + 1;
18.   end
```

Algorithm 17: Simulation algorithm for the reorganizations in R1/W1 left-looking factorization.

```
1.    corecnt := 0;
2.    reorganizecnt := 0;
3.    shiftcnt := 0;
4.    for j := 1 to n begin
5.        incore[j] := false;
6.        crtptr[j] := colptr[j];
7.        nextptr[j] := colptr[j];
8.    end
9.    for j := 1 to n begin
10.       if incore[j] = false begin
11.           if |L_j| > coresize − corecnt
12.               simulate the REORGANIZE task;
13.           if |L_j| > coresize − corecnt
14.               ABORT();
15.           corecnt := corecnt + |L_j|;
16.       end
17.       nextptr[j] := nextptr[j] + 1;
18.       for k in col[j] \ {j} begin
19.           if incore[k] = false begin
20.               if |L_k| > coresize − corecnt
21.                   simulate the REORGANIZE task;
22.               if |L_k| > coresize − corecnt
23.                   ABORT();
24.               corecnt := corecnt + |L_k|;
25.           end
26.           nextptr[j] := nextptr[j] + 1;
27.       end
28.   end
```

Algorithm 18: Simulation algorithm for the reorganizations in R1/W1 right-looking factorization.

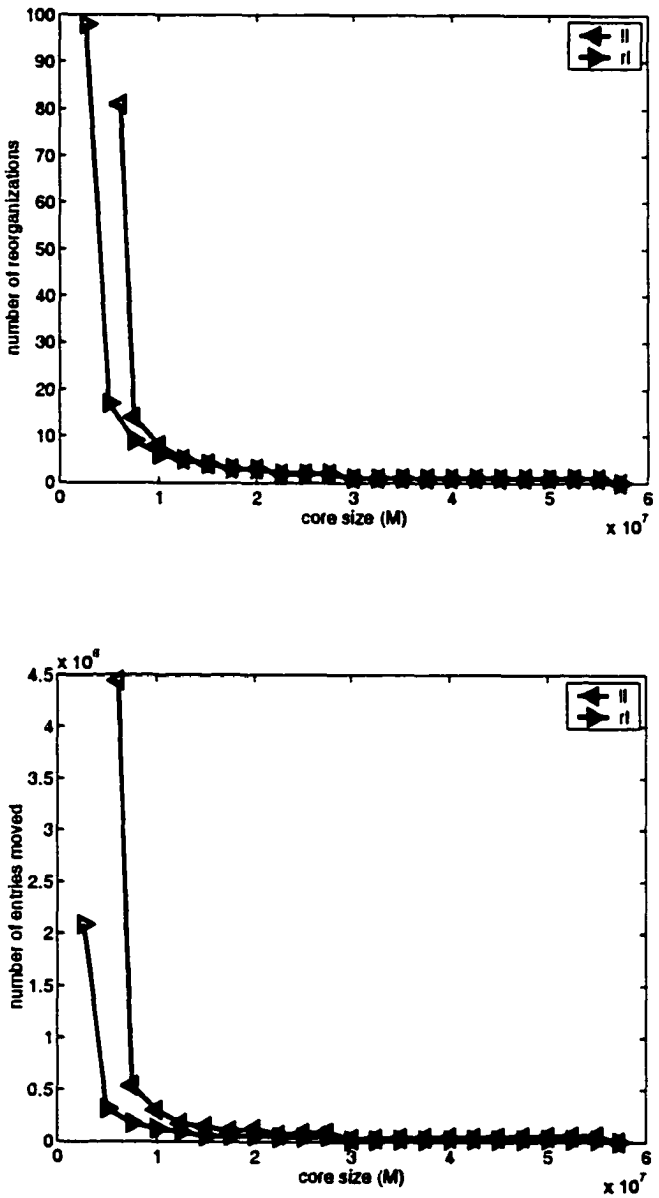


Figure 24: Number of reorganizations and amount of data moved within the core for a 1023 x 1023 2-d model ordered with nested dissection.

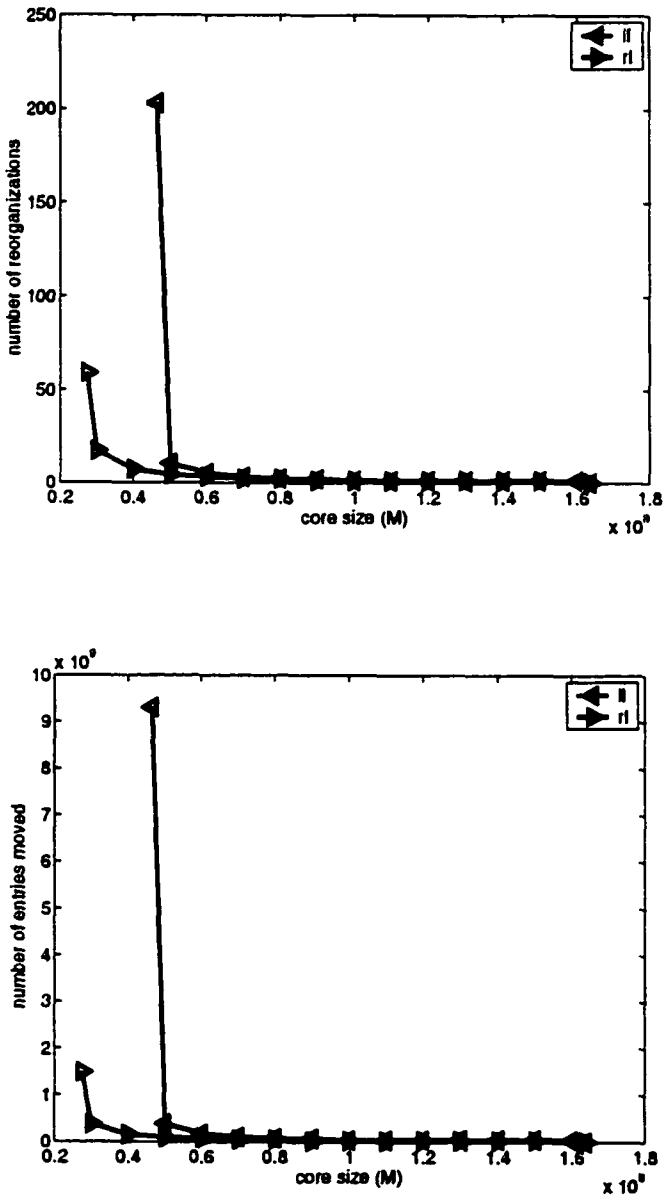


Figure 25: Number of reorganizations and amount of data moved within the core for a 63 x 63 x 63 3-d model ordered with nested dissection.

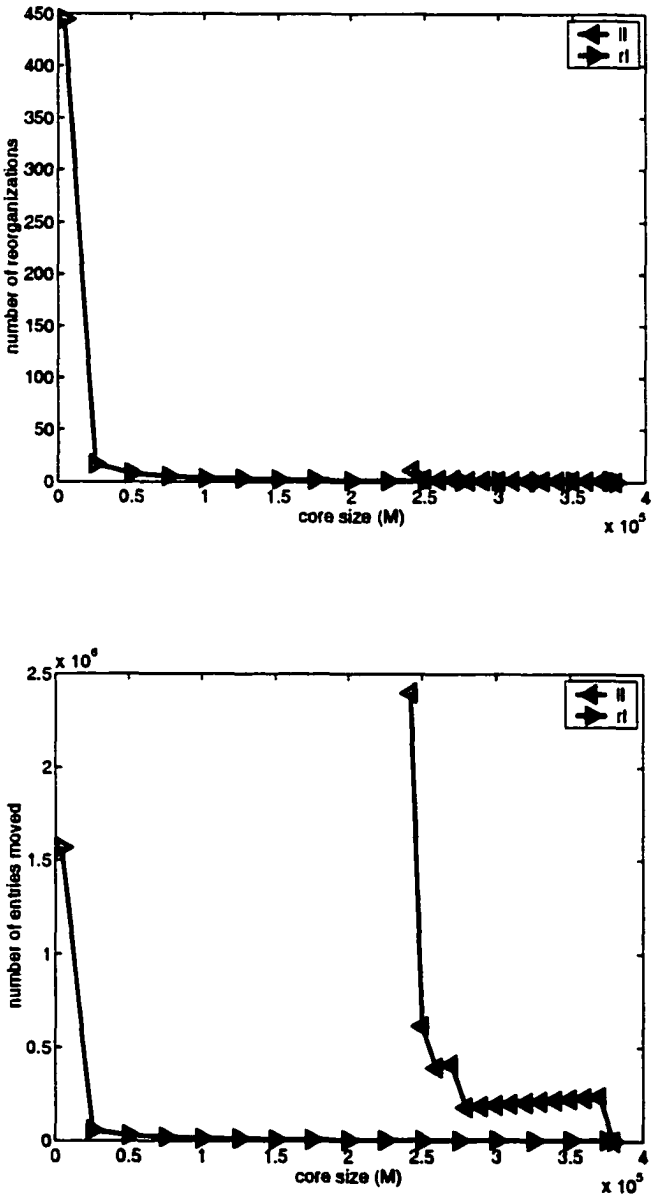


Figure 26: Number of reorganizations and amount of data moved within the core for ken13 ordered nested dissection.

reorganization process for R1/W1 left-looking and right-looking factorization, respectively.

We show results obtained for a 2-d model and a 3-d model, both ordered with the geometrically perfect nested dissection. We selected a 1023 × 1023 2-d model and a 63 × 63 × 63 3-d model. We also show results obtained for ken13, ordered with the nested dissection algorithm from MeTiS.

The results for the models are shown in Figures 24 and 25, respectively. The plot of the top represents the number of core reorganizations while the plot on the bottom represents the amount of data moved within the core during the reorganizations. The core size M is tuned between M_2^L and M_3 for left-looking factorization and between M_2^R and M_3 for right-looking factorization. Note the quick drop in both the number of reorganizations and the amount of data moved. Liu observed the same trend, but only for left-looking factorization and only for the number of reorganizations. We show that this is the trend for both algorithms and for both properties we measure. Note, in addition, that, immediately after the drop, the two algorithms are comparable.

For ken13 the behavior is slightly different. The results are shown in Figure 26. The range of M is very different between the two algorithms because M_2^R is much smaller than M_2^L for this problem. Also, the number of reorganizations is not large for left-looking factorization, for $M = M_2^L$, although the amount of data moved is large for the same value of M. This is due to the particular shape of the elimination tree for this problem, for which we can have a small number of reorganizations but we can still move a significant amount of data. Considering a generalized star elimination tree, one reorganization can move data that corresponds to a large number of leaves.

CHAPTER IV

READ-MANY/WRITE-MANY FACTORIZATION

In the RM/WM scenario the core is too small to allow just reading the input and writing the output.

The issue of interest is therefore minimizing the traffic. A related study belongs to Hong and Kung [26]. Using the red-blue pebble game they proved that the standard multiplication of dense n-by-n matrices on a computing system with a core of size M requires $\Omega(n^3/\sqrt{M})$ traffic. Toledo [54] gave a new proof of the same result. His approach can be adapted in order to show that the left-looking or right-looking factorization of a dense n-by-n matrix also requires $\Omega(n^3/\sqrt{M})$ traffic. Rothberg and Schreiber [47] studied out-of-core factorization algorithms using an experimental approach. They focused on left-looking and multifrontal algorithms and they considered some left-looking/multifrontal hybrids as well.

In this chapter we describe RM/WM factorization algorithms and we determine the complexity of the traffic for model problems. We also compute the exact amount of traffic through simulation.

4.1 READ-MANY/WRITE-MANY FACTORIZATION ALGORITHMS

As in Chapter III, we exemplify with left-looking and right-looking algorithms. We rewrite Algorithms 8 through 11 using explicit data movement.

Algorithms 19 and 20 represent the RM/WM left-looking and right-looking factorizations with 1-d blocks, respectively, and with explicit data movement. Similarly, Algorithms 21 and 22 represent the RM/WM left-looking and right-looking factorization with 2-d blocks, respectively, and with explicit data movement. The DISCARD task removes a block of data from the core. The latter two algorithms are slightly improved versions of Algorithms 10 and 11. The UPDATE tasks that involve just two block entries are treated differently than those that involve three block entries. Also, the FACTOR tasks that involve just one block entry are treated differently than those that involve two block entries.

Note that in Algorithm 19 we don't need to read the whole block column J_b for the UPDATE(L_{J_b}, L_{K_b}) task (line 6). Consider Figure 13. Block column 1 updates block column 17 but

```
1.    for K_b := 1 to N_b begin
2.        READ(L_{K_b});
3.        for J_b in Row[K_b] \ {K_b} begin
4.            READ(L_{J_b});
5.            UPDATE(L_{J_b}, L_{K_b});
6.            DISCARD(L_{J_b});
7.        end
8.        FACTOR(L_{K_b});
9.        WRITE(L_{K_b});
10.       DISCARD(L_{K_b});
11.   end
```

Algorithm 19: RM/WM left-looking factorization with 1-d blocks.

```
1.    for J_b := 1 to N_b
2.        READ(L_{J_b});
3.        FACTOR(L_{J_b});
4.        WRITE(L_{J_b});
5.        for K_b in Col[J_b] \ {J_b} begin
6.            READ(L_{K_b});
7.            UPDATE(L_{J_b}, L_{K_b});
8.            WRITE(L_{K_b});
9.            DISCARD(L_{K_b});
10.       end
11.       DISCARD(L_{J_b});
12.   end
```

Algorithm 20: RM/WM right-looking factorization with 1-d blocks.

```
1.   for K_b := 1 to N_b begin
2.     for J_b in Row[K_b] \ {K_b} begin
3.       READ(L_{K_b,J_b});
4.       READ(L_{K_b,K_b});
5.       UPDATE(L_{K_b,J_b}, L_{K_b,K_b});
6.       WRITE(L_{K_b,K_b});
7.       DISCARD(L_{K_b,K_b});
8.       for I_b in (Col[J_b] \ {J_b}) ∩ (Col[K_b] \ {K_b}) begin
9.         READ(L_{I_b,J_b});
10.        READ(L_{I_b,K_b});
11.        UPDATE(L_{I_b,J_b}, L_{I_b,K_b});
12.        WRITE(L_{I_b,K_b});
13.        DISCARD(L_{I_b,J_b});
14.        DISCARD(L_{I_b,K_b});
15.      end
16.      DISCARD(L_{K_b,J_b});
17.    end
18.    READ(L_{K_b,K_b});
19.    FACTOR(L_{K_b,K_b});
20.    WRITE(L_{K_b,K_b});
21.    for I_b in Col[K_b] \ {K_b} begin
22.      READ(L_{I_b,K_b});
23.      FACTOR(L_{I_b,K_b});
24.      WRITE(L_{I_b,K_b});
25.      DISCARD(L_{I_b,K_b});
26.    DISCARD(L_{K_b,K_b});
27.  end
```

Algorithm 21: RM/WM left-looking factorization with 2-d blocks.

```
1.   for J_b := 1 to N_b begin
2.       READ(L_{J_b,J_b});
3.       FACTOR(L_{J_b,J_b});
4.       WRITE(L_{J_b,J_b});
5.       for I_b in Col[J_b] \ {J_b} begin
6.           READ(L_{I_b,J_b});
7.           FACTOR(L_{I_b,J_b});
8.           WRITE(L_{I_b,J_b});
9.           DISCARD(L_{I_b,J_b});
10.      DISCARD(L_{J_b,J_b});
11.      for K_b in Col[J_b] \ {J_b} begin
12.          READ(L_{K_b,J_b});
13.          READ(L_{K_b,K_b});
14.          UPDATE(L_{K_b,J_b}, L_{K_b,K_b});
15.          WRITE(L_{K_b,K_b});
16.          DISCARD(L_{K_b,K_b});
17.          for I_b in (Col[J_b] \ {J_b}) ∩ (Col[K_b] \ {K_b}) begin
18.              READ(L_{I_b,J_b});
19.              READ(L_{I_b,K_b});
20.              UPDATE(L_{I_b,J_b}, L_{I_b,K_b});
21.              WRITE(L_{I_b,K_b});
22.              DISCARD(L_{I_b,J_b});
23.              DISCARD(L_{I_b,K_b});
24.          end
25.          DISCARD(L_{K_b,J_b});
26.      end
27.  end
```

Algorithm 22: RM/WM right-looking factorization with 2-d blocks.

only rows 19, 20, 22 and 23 are required by the update task. Therefore a partial read is sufficient.

On the other hand, this is no longer possible in Algorithm 20, where we need to move the whole block column L_{K_b} for the UPDATE(L_{J_b}, L_{K_b}) task (line 7). Using Figure 13 as well as block columns 1 and 17 again, even if only rows 19, 20, 22 and 23 are required by the update task, the only efficient solution is to move the whole block column 17. As a consequence, a right-looking UPDATE task performs unnecessary data movement. This is a major reason for avoiding sparse right-looking factorization with 1-d blocking and with explicit data movement, as it will be clear in Section 4.4. In fact, even the dense right-looking factorization should be avoided because it requires twice as much traffic as its left-looking counterpart.

The same happens in the 2-d case. Figure 14 can be used to work out an example. Unfortunately, with 2-d block this happens for left-looking factorization as well. A closer look at Algorithms 21 and 22 should be enough to realize that the traffic is the same in both cases. Therefore, as we will show in Section 4.4, both sparse left-looking and right-looking factorization with 2-d blocking and with explicit data movement should be avoided.

4.2 THE DENSE FACTORIZATION TRAFFIC COMPLEXITY

Since data reuse depends on how dense computations are performed, it is important to understand the complexity of the traffic for dense factorization first. In this section we discuss the complexity of the traffic for dense left-looking and right-looking factorization. First we consider nonblocked algorithms, then we continue with three blocked algorithms: 2-d, 1-d constant width, and 1-d constant area. In each case we determine the complexity of the data traffic based on asymptotically tight lower and upper bounds. We only count out-of-core data accesses here, implicitly making the assumption that the cost of moving data from an arbitrary block is proportional to the area of the block. In the end we discuss the optimality of the factorization algorithms considered here, with respect to the data traffic. We also explain what happens when the factorization is recursive.

Nonblocked Factorization

Lemma 7 *Without blocking, the left-looking and right-looking factorization of a matrix of order n requires $\Theta(n^3)$ traffic.*

Proof

Upper bounds can be determined by counting the number of entry operations and, for each entry operation, the number of entries that may need to be moved. In the k-th outer iteration of the left-looking algorithm there is exactly one operation for each entry in the rectangle at the left of and including column k. The total number of entry operations is thus

$$\begin{aligned}\sum_{k=1}^{n}(n-k+1)k &= \sum_{k=1}^{n}[-k^2+(n+1)k] \\ &= -\sum_{k=1}^{n}k^2+(n+1)\sum_{k=1}^{n}k \\ &= -\frac{n(n+1)(2n+1)}{6}+(n+1)\frac{n(n+1)}{2} \\ &= \frac{n^3+3n^2+2n}{6}.\end{aligned}$$

Similarly, in the j-th outer iteration of the right-looking algorithm there is exactly one operation for each entry in the triangle at the right of and including column j. The total number of entry operations is thus

$$\begin{aligned}\sum_{j=1}^{n}\frac{(n-j+1)(n-j+2)}{2} &= \sum_{j=1}^{n}\left(\frac{1}{2}j^2-\frac{2n+3}{2}j+\frac{n^2+3n+2}{2}\right) \\ &= \frac{1}{2}\sum_{j=1}^{n}j^2-\frac{2n+3}{2}\sum_{j=1}^{n}j+\frac{n^2+3n+2}{2}\sum_{j=1}^{n}1 \\ &= \frac{1}{2}\cdot\frac{n(n+1)(2n+1)}{6}-\frac{2n+3}{2}\cdot\frac{n(n+1)}{2} \\ &\quad +\frac{n^2+3n+2}{2}n \\ &= \frac{n^3+3n^2+2n}{6}.\end{aligned}$$

Obviously, we should expect the same number of entry operations, no matter what factorization algorithm is used.

Now, the maximum number of entries that may need to be moved corresponds to an entry update operation. In this case three entries are involved. Two of them may need to be read only while

the third one may need to be both read and written. An upper bound on the data traffic is thus $\frac{2n^3+6n^2+4n}{3}$ for both algorithms.

For the lower bounds, as long as the core can store at least $\frac{n(n+1)}{2}$ there is minimum traffic (twice the number of entries in the factor, once for reading and once for writing). However, this changes if we assume a smaller core.

For the left-looking algorithm assume that the core can store at most $\frac{n^2}{16}$ entries (consider n divisible by 4) and consider the left side of Figure 27. Then, starting at the outer iteration that corresponds to $k = \frac{n}{4}$ and ending at the outer iteration that corresponds to $k = \frac{3n}{4} + 1$, the core can continuously store the factor entries in the lower left square of L (the area that is marked as *in*). However, in the k-th outer iteration $(n-k+1)k$ entries are required (those in the rectangle at the left of and including column k). Since at most $\frac{n^2}{16}$ of them can be kept in core, at least $(n-k+1)k - \frac{n^2}{16}$ must be moved (the area that is marked as *out*). Between $k = \frac{n}{4}$ and $k = \frac{3n}{4} + 1$ the total number of moved entries is thus

$$
\begin{aligned}
& \sum_{k=n/4}^{3n/4+1} \left[(n-k+1)k - \frac{n^2}{16}\right] \\
= & \sum_{k=n/4}^{3n/4+1} (n-k+1)k - \sum_{k=n/4}^{3n/4+1} \frac{n^2}{16} \\
= & \sum_{k=n/4}^{3n/4+1} [-k^2 + (n+1)k] - \sum_{k=n/4}^{3n/4+1} \frac{n^2}{16} \\
= & -\sum_{k=n/4}^{3n/4+1} k^2 + (n+1) \sum_{k=n/4}^{3n/4+1} k - \frac{n^2}{16} \sum_{k=n/4}^{3n/4+1} 1 \\
= & -\sum_{k=1}^{3n/4+1} k^2 + \sum_{k=1}^{n/4-1} k^2 + (n+1) \sum_{k=1}^{3n/4+1} k - (n+1) \sum_{k=1}^{n/4-1} k \\
& -\frac{n^2}{16} \sum_{k=n/4}^{3n/4+1} 1 \\
= & -\frac{\left(\frac{3n}{4}+1\right)\left(\frac{3n}{4}+2\right)\left(\frac{3n}{2}+3\right)}{6} + \frac{\left(\frac{n}{4}-1\right)\frac{n}{4}\left(\frac{n}{2}-1\right)}{6} \\
& +(n+1)\frac{\left(\frac{3n}{4}+1\right)\left(\frac{3n}{4}+2\right)}{2} - (n+1)\frac{\left(\frac{n}{4}-1\right)\frac{n}{4}}{2} - \frac{n^2}{16} \cdot \frac{n+4}{2} \\
= & \frac{1}{12}n^3 + \frac{37}{48}n^2 + \frac{7}{12}n.
\end{aligned}
$$

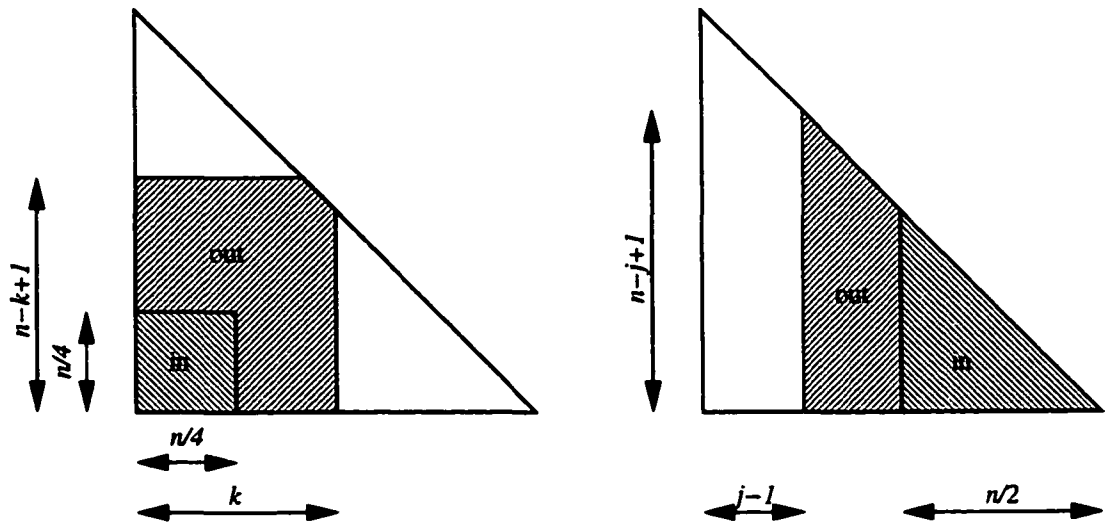


Figure 27: Computing lower bounds on traffic for the nonblocked left-looking and right-looking algorithms.

For the right-looking algorithm assume that the core can store at most $\frac{n(n+2)}{8}$ entries (consider n even) and consider the right size of Figure 27. Then, starting at the outer iteration that corresponds to $j = 1$ and ending at the outer iteration that corresponds to $j = \frac{n}{2}$, the core can continuously store the factor entries in the lower right triangle of L (the area that is marked as *in*). However, in the j-th outer iteration $\frac{(n-j+1)(n-j+2)}{2}$ entries are required (those in the triangle at the right of and including column j). Since at most $\frac{n(n+2)}{8}$ of them can be kept in core, at least $\frac{(n-j+1)(n-j+2)}{2} - \frac{n(n+2)}{8}$ must be moved (the area marked as *out*). Between $j = 1$ and $j = \frac{n}{2}$ the total number of moved entries is thus

$$
\begin{aligned}
& \sum_{j=1}^{n/2} \left[\frac{(n-j+1)(n-j+2)}{2} - \frac{n(n+2)}{8} \right] \\
= & \sum_{j=1}^{n/2} \frac{(n-j+1)(n-j+2)}{2} - \sum_{j=1}^{n/2} \frac{n(n+2)}{8} \\
= & \sum_{j=1}^{n/2} \left(\frac{1}{2} j^2 - \frac{2n+3}{2} j + \frac{n^2+3n+2}{2} \right) - \sum_{j=1}^{n/2} \frac{n(n+2)}{8} \\
= & \frac{1}{2} \sum_{j=1}^{n/2} j^2 - \frac{2n+3}{2} \sum_{j=1}^{n/2} j + \frac{n^2+3n+2}{2} \sum_{j=1}^{n/2} 1 - \frac{n(n+2)}{8} \sum_{j=1}^{n/2} 1 \\
= & \frac{1}{2} \cdot \frac{\frac{n}{2}\left(\frac{n}{2}+1\right)(n+1)}{6} - \frac{2n+3}{2} \cdot \frac{\frac{n}{2}\left(\frac{n}{2}+1\right)}{2}
\end{aligned}
$$

$$+\frac{n^2+3n+2}{2}\cdot\frac{n}{2}-\frac{n(n+2)}{8}\cdot\frac{n}{2}$$
$$= \frac{1}{12}n^3+\frac{1}{4}n^2+\frac{1}{6}n. \quad \square$$

2-d Blocked Factorization

2-d blocks are defined with a 2-d grid of step b. Assume n divisible by b. Then most of the entries end up in b-by-b square blocks (some end up in triangular blocks, along the diagonal). Remember from Section 2.7 that we refer to these blocks as block entries. There are $\frac{N_b(N_b+1)}{2}$ block entries, where $N_b = \frac{n}{b}$.

The entry operations from the nonblocked algorithms are replaced by block entry operations and at most three block entries can simultaneously be in core. Therefore, $3b^2 \leq M$, which means that $b \leq \sqrt{\frac{M}{3}}$.

Lemma 8 *With 2-d blocking, the left-looking and right-looking factorization of a matrix of order n requires $\Theta(n^3/b)$ traffic (b is the block width/height).*

Proof

Using the same approach as before, upper bounds can be determined by counting the number of block entry operations and, for each block entry operation, the number of entries that need to be moved. The number of block entry operations is $\frac{N_b^3+3N_b^2+2N_b}{6}$ for both algorithms. At most three block entries can be moved per block entry operation (two just read, one both read and written) and there are at most b^2 entries in a block entry. An upper bound on the traffic for both algorithms is thus

$$\frac{N_b^3+3N_b^2+2N_b}{6}\cdot 4b^2 = \frac{2n^3}{3b}+2n^2+\frac{4nb}{3}.$$

This time the same approach works for lower bounds as well. At least one block entry is moved per block entry operation (both read and written) and there are at least $\frac{b(b+1)}{2}$ entries in a block entry. A lower bound on the traffic for both algorithms is thus

$$\frac{N_b^3+3N_b^2+2N_b}{6}\cdot b(b+1) = \frac{n^3}{6b}+\frac{n^3}{6b^2}+\frac{n^2}{2}+\frac{n^2}{2b}+\frac{nb}{3}+\frac{n}{3}. \quad \square$$

1-d Constant Width Blocked Algorithms

1-d blocks can be defined with a 1-d grid of step b. Assume n still divisible by b. Then all the entries end up as trapezoidal blocks. Recall from Section 2.7 that we refer to these blocks as block columns. There are N_b block columns, where $N_b = \frac{n}{b}$.

This time we have block column tasks and at most two block columns can be simultaneously in core. The block columns are not as regular as the block entries from the previous case (the number of entries per block column decreases from left to right, with $nb - \frac{b(b-1)}{2}$ entries in the leftmost block column and $\frac{b(b+1)}{2}$ entries in the rightmost block column) but we can bound the number of entries per block column by nb (from above). Therefore, $2nb < M$, which means that $b < \frac{M}{2n}$.

Ignoring the data movement and core management tasks, the algorithms are identical with Algorithms 3 and 4, just that j, k and n are replaced by J_b, K_b and N_b, respectively.

Lemma 9 *With 1-d constant width blocking, the left-looking and right-looking factorization of a matrix of order n requires $\Theta(n^3/b)$ traffic (b is the block width).*

Proof

For upper bounds, using the fact that each block column has less than nb entries, less than $K_b nb$ entries are moved in the K_b-th outer iteration of the left-looking algorithm and less than $(N_b - J_b + 1)nb$ entries are moved in the J_b-th outer iteration of the right-looking algorithm. This determines a common upper bound of

$$\begin{aligned}\sum_{K_b=1}^{N_b} K_b nb &= \sum_{J_b=1}^{N_b} (N_b - J_b + 1)nb \\ &= \frac{N_b(N_b+1)}{2} nb \\ &= \frac{n^3}{2b} + \frac{n^2}{2}.\end{aligned}$$

For lower bounds, focus on the first $\frac{N_b}{2}$ blocks (assume N_b even), each one with more than $\frac{nb}{2}$ entries (N_b even implies n even as well). No more than two of these blocks can fit in core simultaneously. Thus, in the K_b-th outer iteration of the left-looking algorithm, with $K_b \leq \frac{N_b}{2}$, more than $K_b \frac{nb}{2}$ entries are moved, and in the J_b-th outer iteration of the right-looking algorithm,

with $J_b \leq \frac{N_b}{2}$, more than $\left(\frac{N_b}{2} - J_b + 1\right)\frac{nb}{2}$ entries are moved. This determines a common lower bound of

$$\begin{aligned}\sum_{K_b=1}^{N_b/2} K_b \frac{nb}{2} &= \sum_{J_b=1}^{N_b/2} \left(\frac{N_b}{2} - J_b + 1\right)\frac{nb}{2} \\ &= \frac{\frac{N_b}{2}\left(\frac{N_b}{2}+1\right)}{2}\frac{nb}{2} \\ &= \frac{n^3}{16b} + \frac{n^2}{8}. \quad \square\end{aligned}$$

1-d Constant Area Blocked Factorization

The problem with constant area block columns is that the number of entries per block column decreases from left to right, due to the triangular shape of L. What we really want is to fill the core with the entries of two block columns. This is can be done by replacing the requirement for a constant width with a requirement for a constant area.

Let b be the number of columns in the leftmost block column. Then the number of entries per block column can be bounded from above by nb and, since at most two block columns can simultaneously be in core, we have $b = \frac{M}{2n}$. Now, there are $\frac{n(n+1)}{2}$ entries in L, which implies that $N_b > \frac{n+1}{2b}$ (the number of block columns is equal to the total number of entries divided by the number of entries per block column). Thus, balancing the number of entries per block column can halve the number of block columns, compared to the previous case.

Since columns should be added to a block column as long as the number of entries in the block column is less than nb, the number of entries per block column is also larger than $\frac{nb}{2}$ (assume n even), except perhaps for the rightmost block. To see why, consider a block column of c columns, other than the rightmost one, and having at most $\frac{nb}{2}$ entries. Then there are c or less columns at its right which clearly have less than $\frac{nb}{2}$ entries. Thus, the original assumption is not correct.

Lemma 10 *With 1-d constant area blocking, the left-looking and right-looking factorization of a matrix of order n requires $\Theta(n^3/b)$ traffic (b is the maximum block width).*

Proof

Upper bounds can be determined by using the fact that each block column has less than nb entries. This means that in the K_b-th outer iteration of the left-looking algorithm less than $K_b nb$ entries are moved and in the J_b-th outer iteration of the right-looking algorithm less than $(N_b - J_b + 1)nb$ entries are moved. A common upper bound for both algorithms is thus

$$\begin{aligned}\sum_{K_b=1}^{N_b} K_b nb &= \sum_{J_b=1}^{N_b} (N_b - J_b + 1)nb \\ &= \frac{N_b(N_b+1)}{2} nb \\ &= \frac{n^3}{2b} + \frac{n^2}{2}.\end{aligned}$$

Lower bounds can be determined by using the fact that each block column, except perhaps for the rightmost one, has at least $\frac{nb}{2}$ entries. In addition, just to keep the proof simple enough, assume that even in the left-looking algorithm each block column is fully moved (see Section 4.1). Asymptotically, this does not change anything. Then a lower bound for the left-looking algorithm is

$$\begin{aligned}\sum_{K_b=1}^{N_b-1} K_b \frac{nb}{2} &= \frac{N_b(N_b-1)}{2} \cdot \frac{nb}{2} \\ &= \frac{n^3}{4b} - \frac{n^2}{4}\end{aligned}$$

while a lower bound for the right-looking algorithm is

$$\begin{aligned}\sum_{J_b=1}^{N_b-1} (N_b - J_b + 1)\frac{nb}{2} &= \frac{(N_b-1)(N_b+2)}{2} \cdot \frac{nb}{2} \\ &= \frac{n^3}{4b} + \frac{n^2}{4} - nb. \quad \square\end{aligned}$$

Optimal Traffic

It should be obvious now that the nonblocked algorithms are suboptimal from the traffic perspective. Their traffic is $\Theta(n^3)$, which is a factor of $\sqrt{M}$ away from optimal. On the other hand, with $b = \sqrt{\frac{M}{3}}$, the traffic of the 2-d blocked algorithms is $\Theta(n^3/\sqrt{M})$, which makes these algorithms optimal. As for the 1-d blocked algorithms, with $b = \frac{M}{2n}$, their traffic is $\Theta(n^4/M)$. The optimality of these algorithms depends on the relationship between n and M.

If it only makes sense to assume that M is constant then the 1-d blocked algorithms are suboptimal, with a traffic that is a factor of $\sqrt{M}/n$ away from optimal. Note that, asymptotically, this means that the 1-d blocked algorithms are actually worse than the nonblocked ones (although for practical values of n and M they may represent an improvement).

However, we can generally assume several relationships between M and n. Table 6 shows few possible cases, including the constant M above. For M proportional with n^2 1-d blocked algorithms are optimal as well.

M	nonblocked	2-d blocked	1-d blocked
$\Theta(1)$	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^4)$
$\Theta(n)$	$\Theta(n^3)$	$\Theta(n^{5/2})$	$\Theta(n^3)$
$\Theta(n^{3/2})$	$\Theta(n^3)$	$\Theta(n^{9/4})$	$\Theta(n^{5/2})$
$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n^2)$

Table 6: The complexity of the dense factorization traffic for various core sizes.

Recursive Factorization

At this point it is worth mentioning the recursive factorization. Toledo [53] showed that the traffic complexity of the recursive dense factorization is $\Theta(n^3/\sqrt{M})$, which is optimal. This is actually determined by the fact that the recursive factorization is based on 2-d blocked matrix multiplication. Note that he also assumed implicit data movement in his analysis.

For a dense matrix, recursive factorization is thus not better than 2-d blocked left-looking or right-looking factorization from the traffic perspective. The advantage of recursive factorization is adaptivity. We do not need to compute block sizes. Yet, this works well for caches, where the data movement is implicit. With explicit data movement we need to partition the data; therefore, the recursive technique is not suited for out-of-core factorization.

4.3 THE SPARSE FACTORIZATION TRAFFIC COMPLEXITY

We determine the complexity of the sparse factorization traffic for the r-d models (ordered with nested dissection). We focus only on these models this time because the other models we used in Chapter III are not suited for data reuse. We denote the data traffic as D_*, where the asterisk is a placeholder for NB (nonblocked), $1B$ (1-d blocked) and $2B$ (2-d blocked).

As in Section 4.2, we only count out-of-core data accesses in order to determine the traffic complexity, which corresponds to an idealized implicit data movement model. As we will see in Section 4.4, the actual data traffic is quite different with explicit data movement. The results from this section apply to all three sparse factorization algorithms: left-looking, right-looking and multifrontal.

Lemma 11 *For the idealized r-d supernodal elimination tree, using either of the three sparse factorization algorithms, we have the following:*

- $D_{NB} = \Theta(n^{3(r-1)/r})$;
- $D_{2B} = \Theta(n^{3(r-1)/r}/\sqrt{M})$;
- $D_{1B} = \Theta(n^{4(r-1)/r}/M)$.

Proof

The proof covers all three algorithms. Conceptually, it is easier to visualize it from the multifrontal perspective, because we can think in terms of frontal matrices. For left-looking and right-looking factorization we would work with the same entries, even if they are not located within the frontal matrices.

Lower bounds can be determined by focusing on the root of the supernodal tree, where we need to factor a frontal matrix of order k^{r-1}. Assuming that this matrix is sufficiently large with respect to the core, we can write

$$\begin{aligned} D_{NB} &= \Omega(k^{3(r-1)}) \\ &= \Omega(n^{3(r-1)/r}), \end{aligned}$$

$$
\begin{aligned}
D_{2B} &= \Omega\left(\frac{k^{3(r-1)}}{\sqrt{M}}\right) \\
&= \Omega\left(\frac{n^{3(r-1)/r}}{\sqrt{M}}\right), \\
D_{1B} &= \Omega\left(\frac{k^{4(r-1)}}{M}\right) \\
&= \Omega\left(\frac{n^{4(r-1)/r}}{M}\right).
\end{aligned}
$$

For upper bounds note that the order of a frontal matrix at depth D is at most $3(k/2^D)^{r-1}$. The partial factorization of such a matrix requires $\Theta(k^{3(r-1)})$, $\Theta(k^{3(r-1)}/\sqrt{M})$ and $\Theta(k^{4(r-1)}/M)$ traffic, respectively, assuming all frontal matrices are sufficiently large with respect to M. We can plug these expressions into the recursive equations:

$$
\begin{aligned}
D_{NB}(k) &= 2^r D_{NB}\left(\frac{k}{2}\right) + \Theta(k^{3(r-1)}), \\
D_{2B}(k) &= 2^r D_{2B}\left(\frac{k}{2}\right) + \Theta\left(\frac{k^{3(r-1)}}{\sqrt{M}}\right), \\
D_{1B}(k) &= 2^r D_{1B}\left(\frac{k}{2}\right) + \Theta\left(\frac{k^{4(r-1)}}{M}\right).
\end{aligned}
$$

Keep in mind though that these equations describe the upper bounds to the data traffic. Also, note that we did not include the traffic that corresponds to the assembly tasks from the multifrontal factorization. This accounts only for lower order terms. After solving the recursive equations we can thus write

$$
\begin{aligned}
D_{NB} &= O(k^{3(r-1)}) \\
&= O(n^{3(r-1)/r}), \\
D_{2B} &= O\left(\frac{k^{3(r-1)}}{\sqrt{M}}\right) \\
&= O\left(\frac{n^{3(r-1)/r}}{\sqrt{M}}\right), \\
D_{1B} &= O\left(\frac{k^{4(r-1)}}{M}\right) \\
&= O\left(\frac{n^{4(r-1)/r}}{M}\right). \quad \square
\end{aligned}
$$

Note that the 2-d blocking traffic is optimal again, as indicated by the lower bounds.

4.4 COMPUTING THE TRAFFIC THROUGH SIMULATION

We have implemented simulation algorithms that compute the exact amount of traffic for the blocked factorizations. These algorithms are quite trivial and we do not show them here. Basically, we initialize a traffic counter as zero and we keep incrementing it each time we move data. The results correspond to explicit data movement. As we will see shortly, there are significant differences between the three factorization algorithm because of this.

We show results obtained for two model problems ordered with the geometrically perfect nested dissection (a 2-d 1023×1023 grid and a 3-d 63×63×63 grid) as well as for the **ken13** problem order with the nested dissection algorithm from MeTiS.

In each figure we plot the scaled traffic (the actual traffic divided by $2|L|$, which represents the minimum traffic). We tune the core size M between M_1 and $256 \times M_1$.

Figure 28 shows the traffic for the 2-d model. Note that for 1-d blocking the left-looking traffic and the multifrontal traffic are comparable and the trend is that the traffic decreases with the increase of the core size. On the other hand, the right-looking traffic is much larger, due to the unnecessary data movement. The same happens with both the left-looking and the right-looking traffic for 2-d blocking. Note also the particular shape of the plot for right-looking factorization as well as for 2-d blocked left-looking factorization. When the core is small, the large traffic is caused by the lack of data reuse. When the core is large, the large traffic is caused by the unnecessary data movement, as explained in Section 4.1.

Figure 29 shows the traffic for the 3-d model. The same observations apply.

Finally, Figure 30 shows the traffic for the **ken13** problem. The trends are similar, with the right-looking traffic increasing with the core size. However, the range of M includes M_2^R this time, therefore we can switch to R1/W1 factorization when M becomes larger than M_2^R, as indicated by the drop in the right-looking traffic. Note the slight increase in the 1-d blocked multifrontal traffic. This is also due to unnecessary data movement, which is always present in the multifrontal traffic during the assembly tasks. Generally though, this is not visible, as the assemble tasks do not account for a significant amount of traffic.

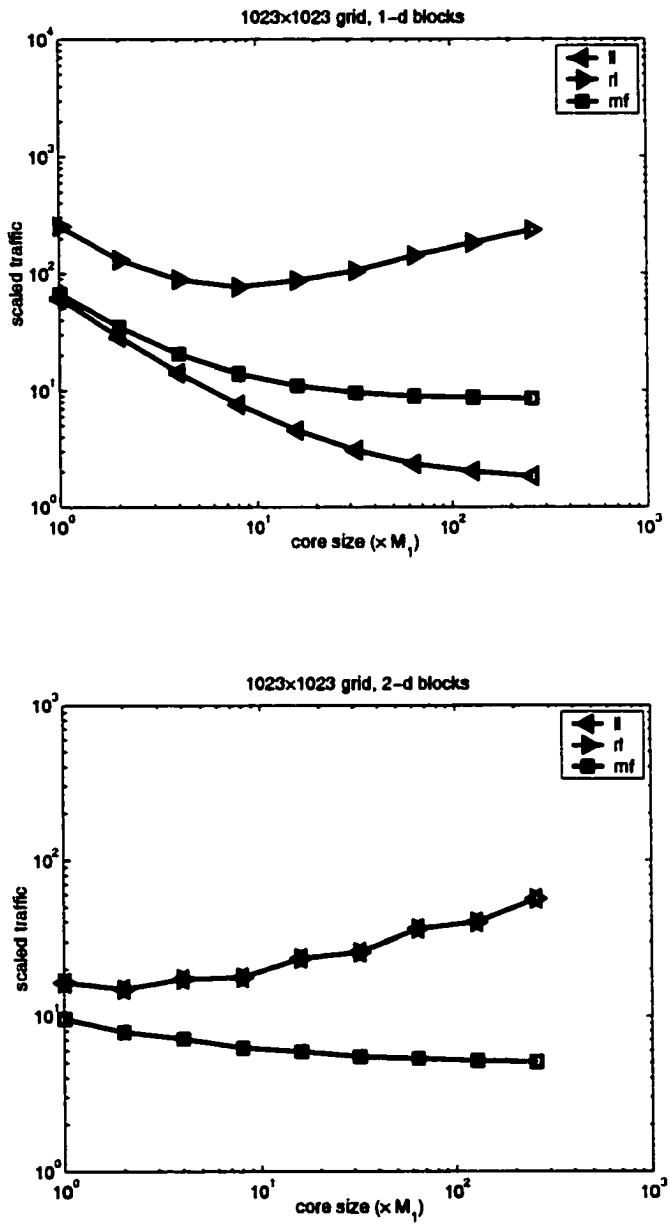


Figure 28: The traffic for a 2-d 1023×1023 model, ordered with nested dissection.

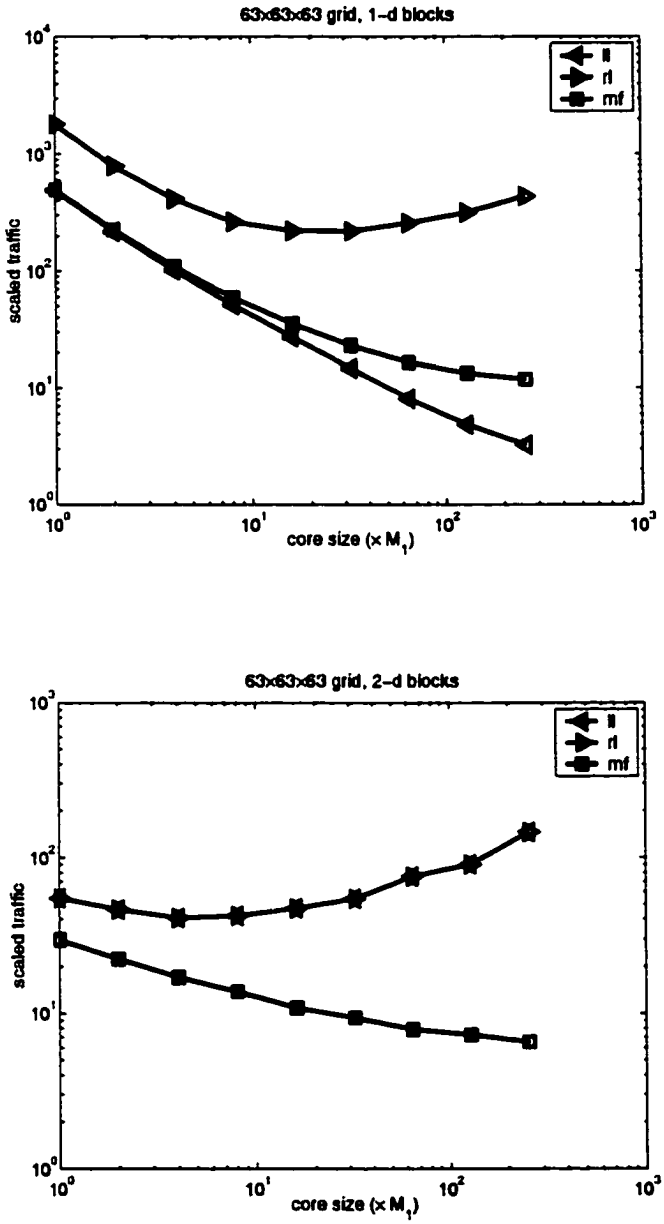


Figure 29: The traffic for a 3-d 63×63×63 model, ordered with nested dissection.

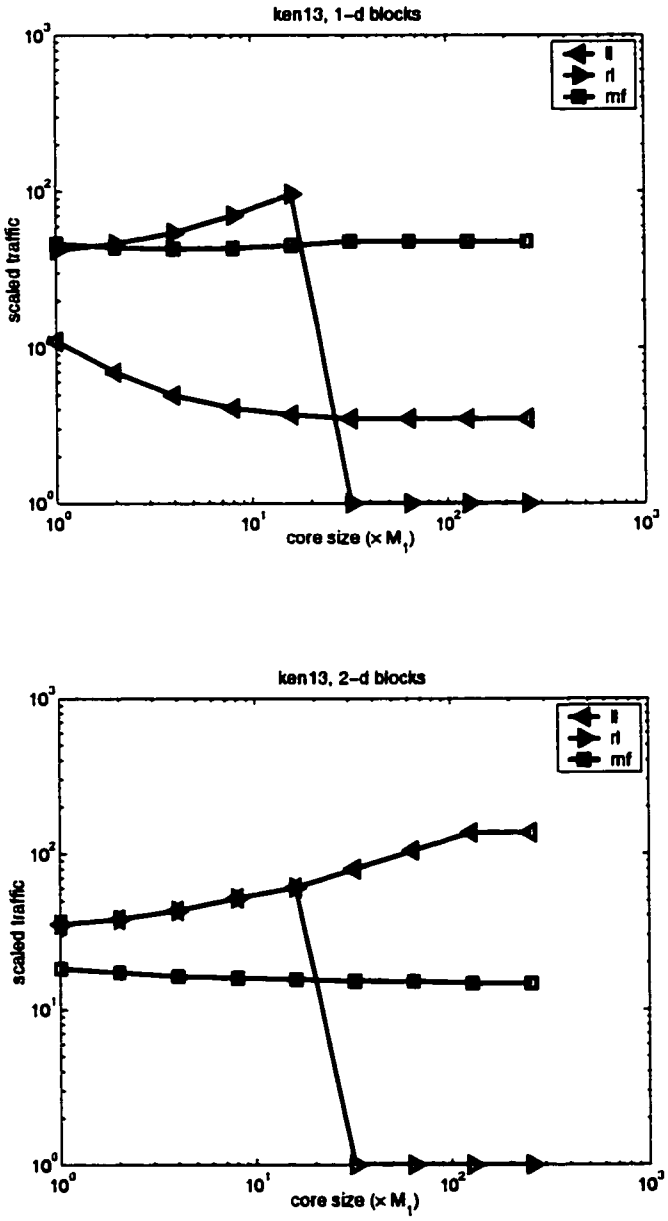


Figure 30: The traffic for ken13, ordered with nested dissection.

To conclude, right-looking factorization should be avoided all the time. Left-looking factorization is a good choice as long as 1-d blocks are used. Otherwise, it should be avoided as well. Multifrontal performs well with both types of blocks. However, it is not a good choice when too much temporary storage is required. In addition, although 2-d blocks are optimal, the difference between 1-d and 2-d blocks may not be significant for practical problem sizes.

CHAPTER V

OBLIO

Oblio is a software package that we wrote as an experimentation tool [15, 16]. The code performs only the factorization and triangular solve steps. We use the nested dissection algorithm from MeTiS [27] or Liu's multiple minimum degree algorithm [22, 31] for ordering.

Most of Oblio is written in C++ [19, 20, 49, 50], in order to take advantage of some object-oriented constructs [10] without compromising efficiency [39, 40]. Only the numerical kernels are partially written in Fortran 77, also for efficiency reasons.

Using C++ and object-orientedness makes Oblio part of the current efforts toward modernizing scientific computing [7, 13, 28, 29].

Oblio solves general indefinite symmetric systems, both real and complex valued. Pivoting is thus available in Oblio but it can be disabled for positive definite systems. In order to accommodate pivoting, the factorization equation used in Oblio is $PAP^T = \tilde{L}D\tilde{L}^T$, where $\tilde{L}$ is block unit lower triangular and D is block diagonal [9]. The pivots are either 1-by-1 or 2-by-2. As for P, it combines both sparsity preservation and numerical stability. The factorization is multifrontal and it is currently blocked only for the registers and for the primary and secondary cache. Due to the nature of these storage layers, blocking is based on implicit data movement. Currently, blocking is enabled only when pivoting is disabled and it is 2-d only.

We organized the presentation of Oblio along two sections. In the first section we address some design issues and in the second one we discuss the performance of Oblio on a particular computing platform.

5.1 SOFTWARE DESIGN

We considered two major issues in Oblio: good software design and efficiency. Since many times these lead to different design decisions, tradeoffs were necessary. We wrote most of the code in C++ because the language offers full support for object-orientedness without enforcing object-oriented programming. This helped us make tradeoffs, the use of object-oriented constructs being

limited in Oblio. In addition, in order to make sure that the numerical computations are efficiently implemented, we partially implemented the numerical kernels in Fortran 77.

The design aspects that we find the most important and we discuss in this section are the object-oriented constructs used, the major classes, the implementation of the numerical kernels and the error handling mechanism.

Object-Oriented Constructs

Two object-oriented constructs are used in Oblio: classes and templates.

Classes are related to the concepts of abstraction and encapsulation. According to Booch [10], an abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer, while encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior, serving to separate the contractual interface of an abstraction and its implementation.

In Oblio classes are used as support for abstraction, in order to help manage the complexity of the code. Our attempt is to express the computation in terms of the problem rather than in terms of the implementation.

On the other hand, encapsulation is weak in Oblio, for efficiency reasons. Instead of using only abstract interfaces, implementation details are provided outside the classes, with the drawback that any modification of an implementation is no longer local to the corresponding class.

As for templates, they help reducing the size of the code when many classes are very similar and they also make the code less error-prone. A template class is just a generic class that can be instantiated in several ways.

In Oblio template classes are used for the the array-like objects discussed next as well as for real/complex valued numerical objects.

Arrays

Most Oblio objects allocate storage dynamically in the form of arrays. Such arrays are described by the **Array** template class, partially shown in Figure 31. Since it is a template class, **Array** can be used to instantiate arrays of integers, double precision numbers, and so on, using the same code.

From the abstraction perspective, note first that **Array** groups the two array attributes, the size (**size_**) and the beginning address of the data (**item_**). This solves the problem of raw C++ arrays, whose sizes can be easily lost or forgotten, which creates a potential for errors.

Special support for abstraction comes from the constructor and destructor methods, which provide the mechanism for initialization and cleanup. The storage used by an **Array** object must be dynamically allocated when the object begins to exist and freed when the object ends to exist. This tasks, however, should not be performed explicitly. All that the user of an **Array** object should be required to do is to declare it and to specify its size at the time of the declaration. Storage allocation should be performed implicitly at the same time, and the same storage should also be implicitly freed when the object is thrown away.

In C++ storage is allocated and freed dynamically by calling the **new** and **delete** operators, respectively. However, these operators are not directly called by the constructor and destructor methods. The reason is that the tasks of allocating and freeing storage are not equivalent with the tasks of initialization and cleanup. In order to understand this consider Figure 33, which shows the constructor and the destructor of the **Array** template class, as well as the **resize** method, used for modifying the size of **Array** objects. Storage is allocated by both the constructor and the **resize** method. Similarly, storage is freed by both the destructor and the **resize** method.

As a consequence, the task of allocating and freeing storage are performed by the **allocate** and **free** methods, respectively, as shown in Figure 32. Note that the code of **allocate** includes handling potential exceptions generated by calling the **new** operator and the code of **free** includes setting **item_** to zero, which is not performed by the **delete** operator. Going back to Figure 31, note also that **allocate** and **free** are **private** methods, which forbids any access to them from outside the **Array** class.

```
#undef __CLASS__
#define __CLASS__ "Array"
template<class T>
class Array
{
  private:
    int size_;
    T* item_;

    Array(const Array&);

    void allocate(void);
    void free(void);

  public:
    Array(int size);
    virtual ~Array();
    Array& operator=(const Array&);

    int getSize(void) const {return size_;}
    T* getItem(void) {return item_;}
    const T* getItem(void) const {return item_;}

    void resize(int size);
    int getHeapSize(void) const;
    void print(void) const;

    /* ... */
};
```

Figure 31: The `Array` template class.

```
#undef __FUNC__
#define __FUNC__ "allocate"
template<class T>
void Array<T>::allocate(void)
{
  BEGIN_FUNCTION();

  try
  {
    item_ = new T[size_];
  }
  catch (...)
  {
    SET_ERROR1(MemoryAllocation);
  }

  END_FUNCTION();
}

#undef __FUNC__
#define __FUNC__ "free"
template<class T>
void Array<T>::free(void)
{
  BEGIN_FUNCTION();

  delete [] item_;
  item_ = 0;

  END_FUNCTION();
}
```

Figure 32: The **allocate** and **free** methods from the **Array** template class.

```
#undef __FUNC__
#define __FUNC__ "Array"
template<class T>
Array<T>::Array(int size):
  size_(size),
  item_(0)
{
  BEGIN_FUNCTION();

  allocate();

  END_FUNCTION();
}

#undef __FUNC__
#define __FUNC__ "~Array"
template<class T>
Array<T>::~Array()
{
  BEGIN_FUNCTION();

  free();

  END_FUNCTION();
}

#undef __FUNC__
#define __FUNC__ "resize"
template<class T>
void Array<T>::resize(int size)
{
  BEGIN_FUNCTION();

  free();
  size_ = size;
  allocate();

  END_FUNCTION();
}
```

Figure 33: The constructor, destructor and the `resize` methods from the `Array` template class.

From the encapsulation perspective, this is clearly weakened by the `getItem` method, which provides direct access to `item_`. One place where this access becomes necessary is within the numerical kernels, which, as we mentioned, are partially implemented in Fortran 77. Note the two versions of the `getItem` method, one that can be called for `Array` objects that are not `const` and one that can be called only for `Array` objects that are `const`. This provides protection against modifying data within `const Array` objects.

Few other features of the `Array` class are worth mentioning because they are shared by other classes in Oblio.

Two methods that are closely related are the copy constructor and the assignment operator. Both are implicitly included, with bitwise copy as the default behavior. For classes that have pointers as attributes this determines a shallow copy and several objects may end up pointing to the same data. This scheme is not correct. The correct alternatives are either deep copy or shared data with reference counting.

In Oblio we disable the copy constructor and we choose deep copy for the assignment operator. We prefer deep copy because it is simpler to implement and because we don't find any strong reason for sharing objects. Yet, since deep copy is time consuming, there is no reason to use it for passing and returning objects by value. Oblio objects are passed and returned only by reference.

In order to disable the copy constructor we make it `private`. This way it cannot be called from outside the class. Also, no explicit definition is given for the copy constructor, which means that the default one (shallow copy) is in use. This does no harm because the copy constructor cannot be called anyway.

On the other hand, the assignment operator is `public` and a definition that performs deep copy is provided.

The `print` method is provided for debugging while the `getHeapSize` method helps keeping track of the heap allocation.

Some error handling code is also present in Figures 31, 32 and 33. The definition of the `Array` class is preceded by the definition of the `_CLASS_` tag. Similarly, the definition of each method of

the `Array` class (except those defined in the body of the class) is preceded by the definition of the `__FUNC__` tag and, in addition, each method begins by calling the `BEGIN_FUNCTION` macro and ends by calling the `END_FUNCTION` macro. Another macro, `SET_ERROR1` is called by the `allocate` method, inside the `try/catch` block. We discuss Oblio's error handling mechanism later in this section.

Major Classes

Oblio's major classes describe two types of objects: data objects and algorithm objects. Sparse matrices, permutations and factors are examples of data objects. Factorization and triangular solve algorithms are examples of algorithm objects.

The `SparseMatrix` template class for example, partially shown in Figure 34, describes sparse symmetric coefficient matrices. Like all Oblio's numerical classes, `SparseMatrix` is a template class. This helps instantiating both real and complex valued objects using the same code.

`SparseMatrix` groups attributes such as `order_`, `numberOfOffDiagonals_`, `columnPointer_`, `rowIndex_` and `entry_`. The last three are implemented as pointers to `Array` objects.

The design of `SparseMatrix`, shared by the design of all data classes, follows the design of `Array`. The `allocate` and `free` methods perform the tasks of allocating and freeing storage dynamically. These are called by the constructor and destructor methods, as well as by the `resize` method. Access to the implementation is provided through methods such as `getColumnPointer`, which returns `columnPointer_`. The copy constructor is disabled while the assignment operator performs deep copy. The `print` method is provided for debugging and the `getHeapSize` method helps keeping track of the heap allocation.

Other data classes are `Permutation`, `EliminationForest`, `SparseFactors`, `FrontalMatrix`, `UpdateMatrix`, `UpdateStack`.

The `MultifrontalFactor` template class, partially shown in Figure 35, describes multifrontal factorization algorithm objects. Again, this is a template class in order to deal with both real and complex valued arithmetic.

A `MultifrontalFactor` object stores pointers the several data objects its accesses. The task of

```
#undef __CLASS__
#define __CLASS__ "SparseMatrix"
template<class T>
class SparseMatrix
{
  private:
    int order_;
    int numberOfOffDiagonals_;
    Array<int>* columnPointer_;
    Array<int>* rowIndex_;
    Array<T>* entry_;

    /* ... */

    SparseMatrix(const SparseMatrix<T>&);

    void allocate(void);
    void free(void);

    /* ... */

  public:
    SparseMatrix();
    SparseMatrix(int order, int numberOfOffDiagonals);
    virtual ~SparseMatrix();
    SparseMatrix<T>& operator=(const SparseMatrix<T>&);

    int getOrder(void) const
      {return order_;}
    int getNumberOfOffDiagonals(void) const
      {return numberOfOffDiagonals_;}
    Array<int>* getColumnPointer(void)
      {return columnPointer_;}
    const Array<int>* getColumnPointer(void) const
      {return columnPointer_;}

    /* ... */

    void resize(int order, int numberOfOffDiagonals);
    int getHeapSize(void) const;
    void print(void) const;

    /* ... */
};
```

Figure 34: The SparseMatrix template class.

```
#undef __CLASS__
#define __CLASS__ "MultifrontalFactor"
template<class T>
class MultifrontalFactor
{
  private:
    const SparseMatrix<T>* a_;
    Permutation* p_;
    const EliminationForest* f_;
    SparseFactors* l_;
    UpdateStack<T>* u_;
    FrontalMatrix<T>* fm_;
    UpdateMatrix<T>* um_;

    /* ... */

    MultifrontalFactor(const MultifrontalFactor<T>&);
    MultifrontalFactor<T>& operator=(const MultifrontalFactor<T>&);

  public:
    MultifrontalFactor();
    virtual ~MultifrontalFactor();

    void setA(const SparseMatrix<T>& a)
      {a_ = &a;}
    void setP(Permutation& p)
      {p_ = &p;}
    void setF(const EliminationForest& f)
      {f_ = &f;}
    void setL(const SparseFactors& l)
      {l_ = &l;}
    void setU(UpdateStack<T>& u)
      {u_ = &u;}

    /* ... */

    void run(void);
};
```

Figure 35: The `MultifrontalFactor` template class.

```
enum Error {None, /* ... */};

extern Error ErrorCode;
extern char* ErrorMessage[];

#define CLEAR_ERROR() \
{ \
  ErrorCode = None; \
}
```

Figure 36: Error handling related data types and variables and the macro used to reset the current error.

most methods is to set the values of these pointers. An exception is the run method, which performs the actual factorization.

Having classes that describe algorithm objects is a natural approach for algorithms such as the factorization. If the factorization would have been implemented as a method within a data class it is not clear which data class it was supposed to belong to.

Another algorithm class is `Solve`, which describes triangular solve algorithm objects. There are also classes that describe ordering algorithms but these are only wrappers around library calls.

Error Handling

Error handling code is supposed to execute very rarely. Yet, it usually accounts for a large number of lines of code. In addition, it can make any code messy. A simple and effective error handling mechanism is thus desirable.

While C++ provides exceptions as an error handling mechanism, with their advantages, we chose a simple macro-based scheme in Oblio. There is a global `ErrorCode` variable of type `Error` that records the last error code. `Error` is an enumeration and `ErrorCode` is initially set to `None`. Its value never changes unless an error occurs. In order to reset it to `None`, the `CLEAR_ERROR` macro needs to be called. Figure 36 shows the definitions of the `Error` type, the `ErrorCode` variable and the `CLEAR_ERROR` macro.

A number of Oblio methods can set `ErrorCode` under some circumstances. They do it by calling one of two macros, `SET_ERROR1` and `SET_ERROR2`, shown in Figure 37. The first one is used by those

```
#define SET_ERROR1(errorCode) \
{ \
  ErrorCode = (errorCode); \
  END_FUNCTION(); \
  return; \
}

#define SET_ERROR2(errorCode, returnValue) \
{ \
  ErrorCode = (errorCode); \
  END_FUNCTION(); \
  return (returnValue); \
}
```

Figure 37: Macros used to set the current error.

methods that do not return anything while the second one is used by those methods that return something.

Any method call that can set ErrorCode should be followed by a call of a macro that checks ErrorCode. There are also two error checking macros. CHECK_ERROR1 is supposed to be used when there is nothing to return from the current scope while CHECK_ERROR2 is supposed to be used otherwise. The error checking macros are shown in Figure 38.

We have previously mentioned the definitions of the __CLASS__ and __FUNC__ tags throughout the code. These are also part of the error handling mechanism. Their purpose is to help tracing method calls. The definition of each class is preceded by the definition of the __CLASS__ tag and the definition of each method (except those defined in the body of the class to which they belong) is preceded by the definition of the __FUNC__ tag. Also, each such method begins by calling the BEGIN_FUNCTION macro and ends by calling the END_FUNCTION macro. These two macros can be customized to perform several tasks at the beginning and end of a method call. For a normal execution they are simply empty. However, a method call tracing mode can be enabled in Oblio, in which the BEGIN_FUNCTION macro writes the values of the __CLASS__ and __FUNC__ tags to the standard error. There is also an error tracing mode, in which the error setting and checking macros themselves write the values of the __CLASS__ and __FUNC__ tags to the standard error. Note that END_FUNCTION is also called by the error setting and checking macros. This way the END_FUNCTION call does not need to be typed each

```
#define CHECK_ERROR1() \
{ \
  if (ErrorCode != None) \
  { \
    END_FUNCTION(); \
    return; \
  } \
}

#define CHECK_ERROR2(returnValue) \
{ \
  if (ErrorCode != None) \
  { \
    END_FUNCTION(); \
    return (returnValue); \
  } \
}
```

Figure 38: Macros used to check the current error.

time a method returns due to an error.

There is a single situation in which an exception can be thrown, when calling **new**. Such an exception is immediately caught in Oblio and translated into a memory allocation error. This can be seen inside the definition of the **allocate** method of the **Array** class. The call of **new** is placed within a **try** block. If the memory allocation fails, the exception is caught and the corresponding error is set within the **catch** block.

The main disadvantage of this error handling mechanism is that calls to the error setting/checking macros can be easily forgotten and thus errors can be missed. Some programming discipline is required in order to avoid this. We prefer this mechanism though because it is simple enough for an experimental code.

Numerical Kernels

Oblio's numerical kernels are blocked for the secondary and primary cache, and for the registers. The blocks are 2-d and the data movement is implicit. Explicit data movement is possible only at the interface between the disk and the core and Oblio does not perform out-of-core computations yet.

```
      ai1 = 1
      ci1 = 1
      do 103 i = 1, m3
        b1j = 1
        cij = ci1
        do 102 j = 1, n3
          aik = ai1
          bkj = b1j
          do 101 k = 1, l3
            c(cij) = c(cij) - a(aik) * b(bkj)
            aik = aik + lda
            bkj = bkj + 1
101       continue
          b1j = b1j + ldb
          cij = cij + ldc
102     continue
        ai1 = ai1 + 1
        ci1 = ci1 + 1
103   continue
```

Figure 39: Fortran 77 code for dense matrix multiplication without register blocking.

The numerical kernels are organized as a chain of function calls: the secondary cache block functions call the primary cache block functions, and the primary cache block functions call the register block functions. The actual numerical computations are performed by the register block functions, which are written in Fortran 77. The secondary/primary cache block functions are written in C++.

There is an important difference between the secondary/primary cache block functions and the register block functions. For the former, the body of each function is a nested loop with the block size as a parameter and the inner loop calls the next function in the chain. At register level this scheme is no more possible. A different function must be written for each register block size. In order to illustrate this we use matrix multiplication, which is the most frequent block operation. Figure 39 shows the main body of the matrix multiplication function at register level without blocking (or 1-by-1 blocking). Similarly, Figure 40 represents the main body of the matrix multiplication function at register level for 2-by-2 blocks. The idea is to force the processor to reuse data by keeping it in registers. This is done by declaring local variables that store the block entries.

The perspective of writing a different function for each block size is not very appealing, especially

```
      ai1 = 1
      ci1 = 1
      do 203 i = 1, m3a, 2
        b1j = 1
        cij = ci1
        do 202 j = 1, n3a, 2
          c11 = c(cij)
          c12 = c(cij + ldc)
          c21 = c(cij + 1)
          c22 = c(cij + ldc + 1)
          aik = ai1
          bkj = b1j
          do 201 k = 1, l3a, 2
            a11 = a(aik)
            a12 = a(aik + lda)
            a21 = a(aik + 1)
            a22 = a(aik + lda + 1)
            b11 = b(bkj)
            b12 = b(bkj + ldb)
            b21 = b(bkj + 1)
            b22 = b(bkj + ldb + 1)
            c11 = c11 - a11 * b11 - a12 * b21
            c12 = c12 - a11 * b12 - a12 * b22
            c21 = c21 - a21 * b11 - a22 * b21
            c22 = c22 - a21 * b12 - a22 * b22
            aik = aik + lda + lda
            bkj = bkj + 2
201       continue
          c(cij) = c11
          c(cij + ldc) = c12
          c(cij + 1) = c21
          c(cij + ldc + 1) = c22
          b1j = b1j + ldb + ldb
          cij = cij + ldc + ldc
202     continue
        ai1 = ai1 + 2
        ci1 = ci1 + 2
203   continue
```

Figure 40: Fortran 77 code for dense matrix multiplication with 2-by-2 register blocks.

because for block sizes larger than one there are particular cases to deal with (when the primary cache block size is not a multiple of the register block size). However, the number of registers in a processor is limited, therefore there should be only few block sizes to consider. In Oblio, the register block sizes are one, two, three and four.

5.2 CASE STUDY: THE SGI ORIGIN 200

We have experimented with Oblio on an SGI Origin 200 machine built with two 270 MHz MIPS R12000 processors and running the IRIX Release 6.5 IP27 operating system. Due to the sequential nature of the code, we used a single processor in each experiment.

MIPS R12000 is a four-way superscalar processor with five functional units [24, 43]: two integer, two floating point, and a load-store unit. Integer addition, floating point addition and multiplication, and load and store, all have a repeat rate of one cycle. A maximum of two floating point operations, one addition and one multiplication, can be executed every cycle, which determines a theoretical peak rate of 540 Mflop/s. There can be a maximum of one data movement instruction (load or store) per cycle though.

The register file on the processor contains 64 integer registers and 64 floating point registers (double precision). The number of logical registers is actually half of this (32 integer and 32 floating point). Each processor has an on chip two-way set associative primary data cache with a capacity of 32 KB and a cache line size of 32 B. There is also an external secondary data cache for each processor that has a capacity of 4 MB and a cache line size of 64 B/128 B (programmable). The main memory is shared by the two processors and has 256 MB. The virtual memory is set to 512 MB.

Since one load or store can graduate every cycle, the peak bandwidth between the primary data cache and the register file is 8 B per cycle or 2160 MB/s. The bus between the secondary and the primary data cache runs at the processor's internal frequency and can transfer 16 B every cycle, for a peak bandwidth of 4,320 MB/s.

We performed three sets of experiments. In the first one we measured the performance of Oblio's dense matrix multiplication kernel in isolation. In the second one we measured the performance of

Oblio's sparse factorization. In the third one we measured the effect of the data movement between the primary and the secondary storage on performance, using the BLAS dense matrix multiplication kernel from SCSL (SGI's scientific library). We used the MIPSpro Compilers Version 7.3.1.1m C++ and Fortran 77 Compilers with level 3 optimization (`CC -O3`, `f77 -O3`). All the experiments were based on double precision real valued data and arithmetic.

Oblio's In-Core Dense Matrix Multiplication

Either rate or time would be appropriate metrics to report in order present the performance of the numerical kernels. We prefer to report rate results because this way we can compare achieved rate to peak rate and we can determine how well the processor is used. However, in order to measure rate we had to measure time first, using the UNIX `times` function [48]. All the results reported in this section are based on the wall time returned by the `times` function (as opposed to the CPU time, available through the same function).

We focused on register blocking first. Consider the code from Figure 39. The maximum rate it can achieve is determined by the ratio between the number of floating point operations and the number of data movement instructions times the maximum number of data movement instructions that can be executed by the processor per cycle. Since the value of the latter is one, the code's maximum rate is numerically equal to the floating point operation/data movement instruction ratio.

Look at the inner loop, where most of the arithmetic operations and data accesses occur. Three floating point values are accessed in each iteration: `c(cij)`, `a(aik)` and `b(bkj)`. Since `c(cij)` is an invariant with respect to the inner loop, it can be kept in a register. The other two floating point values need to be loaded only. That means two load instructions for each iteration of the inner loop. Also, two floating point operations are performed in each iteration of the inner loop, one multiplication and one addition.

If the matrices to be multiplied are of order n then the total number of floating point operations is $2n^3$, the total number of load instructions is about $2n^3$ and the total number of store instructions is not significant. The ratio between floating point operations and data movement instructions is

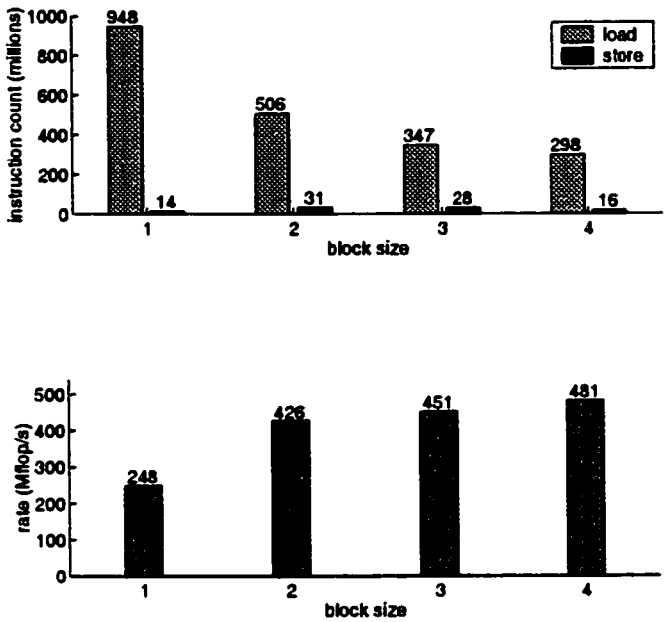


Figure 41: Dense matrix multiplication blocked for registers.

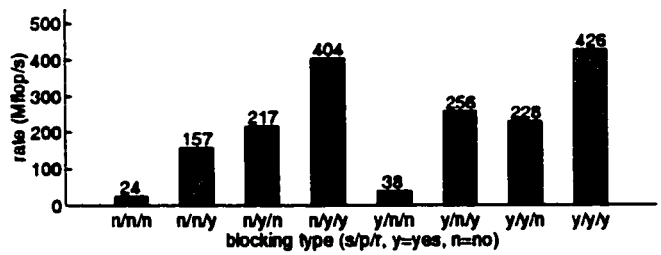


Figure 42: Dense matrix multiplication blocked for secondary cache, primary cache, and registers.

thus 1, which means a maximum rate of one floating point operation per cycle or 270 Mflop/s. Unfortunately, this is only half of the theoretical peak rate.

Consider now register blocking, such as in Figure 40. For a block size b the total number of floating point operations is still $2n^3$, the total number of load instructions changes to about $(n/b)^3 \cdot 2b^2 = 2n^3/b$, and the total number of store instructions remains not significant. In this case the ratio between floating point operations and load instructions becomes b. Thus, we can control the maximum rate through the block size. Of course, the processor cannot perform more than two floating point operations per cycle and it cannot even store large blocks in the register file. Theoretically thus, the optimum register block size is 2, corresponding to a maximum rate equal to the theoretical peak rate.

Figure 41 shows measurements for 1-by-1, 2-by-2, 3-by-3, and 4-by-4 register blocks. The matrices were of order 36, chosen such that all three fit in the primary cache, and the computation was repeated 10,000 times. For each experiment we report the number of graduated load and store instructions, measured with `perfex`, SGI's performance monitor, and the achieved rate. Note how the number of loads decreases with the register block size, as expected. On the other hand, the number of store instructions is not significant. Note also how rate increases with the register block size. As expected, without blocking the rate is lower that half of the theoretical peak.

We also looked at blocking for primary and secondary cache. Figure 42 shows measurements for the eight possible combinations (including register blocking). This time the matrices were of order 1,000 (all three fit in the main memory but not in the secondary cache) and the computation was performed only once. The block size was 4 for registers, 36 for the primary cache and 400 for the secondary cache. Note the impact of register and primary cache blocking on rate. On the other hand, secondary cache blocking has little effect (there are much more primary cache misses than secondary cache misses, and primary and secondary cache miss penalties are comparable).

Oblio's In-Core Sparse Factorization

We report both rate and time for the sparse factorization. As before, reporting rate helps comparing achieved rate to peak rate, but it can also be misleading. A high rate can always be obtained with a bad ordering at the price of a dramatic increase in the time.

We show results obtained for two model problems ordered with the geometrically perfect nested dissection (a 2-d 512×512 grid and a 3-d 31×31×31 grid). We blocked only for registers and for the primary cache. As we have already shown, secondary cache blocking does not determine a significant gain and, in addition, most front clusters in the test problems are not that large.

Figures 43 and 44 show the results. Note that a better rate is obtained for the 3-d model with both primary cache and register blocking and remember that a 3-d model has a higher potential for data reuse than a 2-d model.

Out-of-Core Dense Matrix Multiplication Using SCSL's BLAS

At this time Oblio provides a starting point for an out-of-core solver and the question is if the out-of-core data movement should be implicit or explicit. This last group of experiments suggests that the latter is the right approach.

This time we used the **dgemm** BLAS kernel from SCSL. The matrix orders were 1,000, 2,000 and 4,000. The corresponding storage requirements are 23 MB, 92 MB and 367 MB, respectively (the storage required for the the multiplication of order n matrices is $3 \cdot n^2 \cdot 8$ MB). At 256 MB of main memory only the first two multiplications can be performed in-core.

We performed three experiments. As before, we measured the CPU time and the wall time and we determined the rate based on the number of floating point operations and on the wall time. We also measured the number of page faults using the UNIX **getrusage** function [48] (this is not part of the POSIX standard but it is implemented by many UNIX-like operating systems, including IRIX).

In the first experiment we performed the multiplication with a single **dgemm** call for each matrix order.

In the second experiment we blocked the matrix multiplication of the order 4,000 matrices along

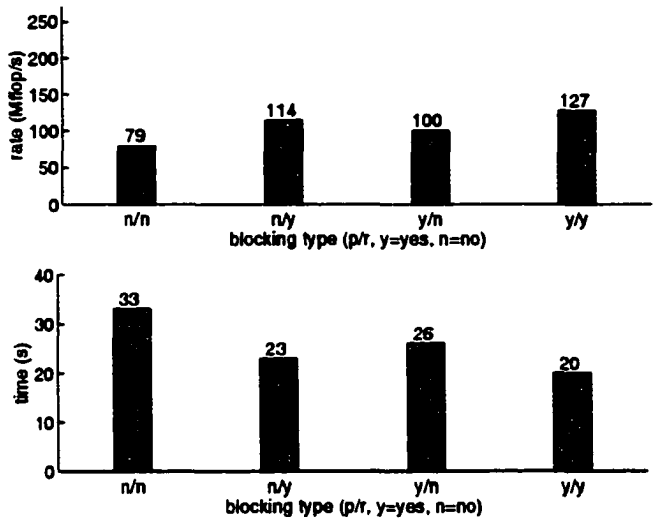


Figure 43: Factorization rate and time for a 2-d 511×511 model, ordered with nested dissection.

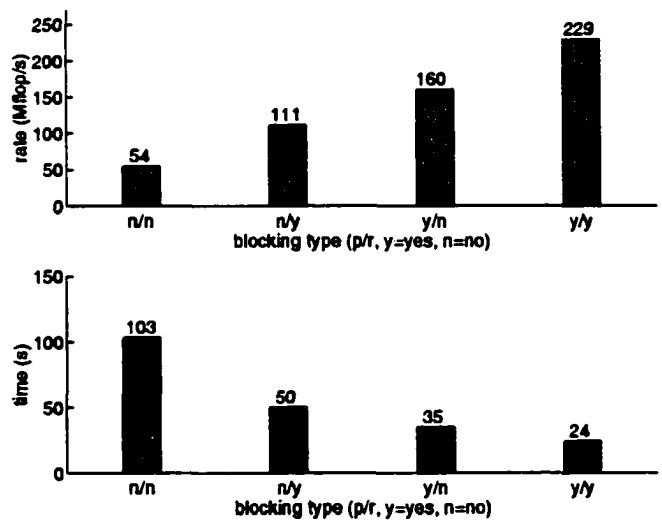


Figure 44: Factorization rate and time for a 3-d 31×31×31 model, ordered with nested dissection.

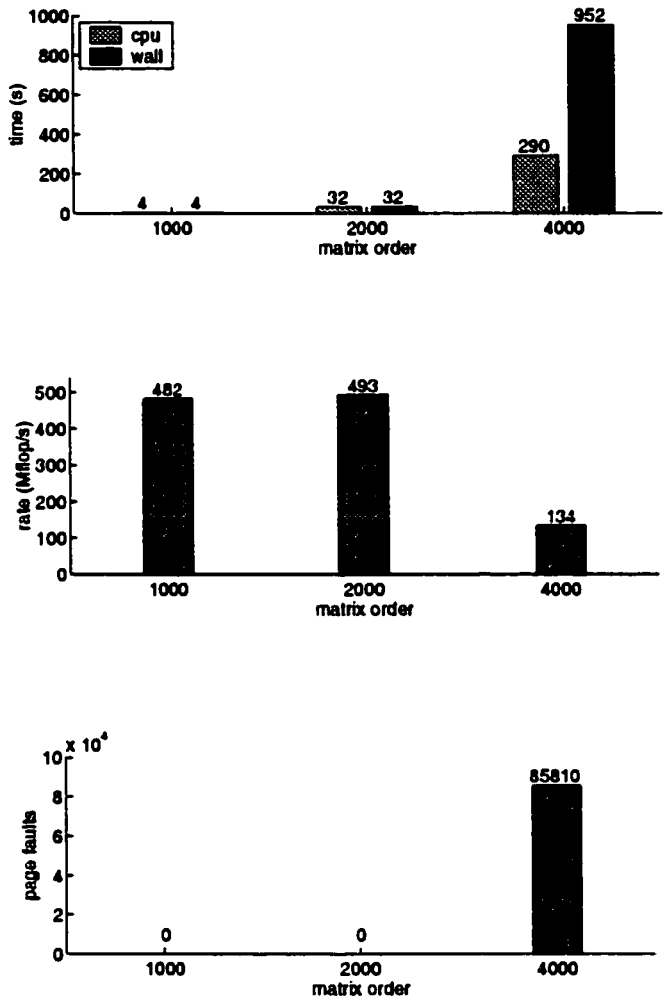


Figure 45: Execution time, rate, and page faults for nonblocked dense matrix multiplication.

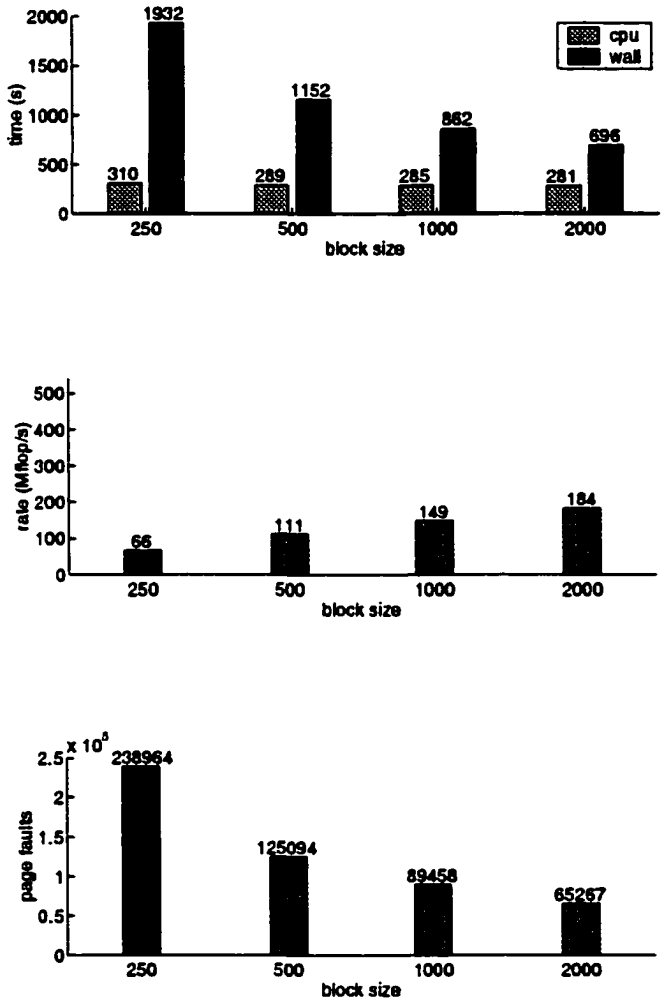


Figure 46: Execution time and rate, and page faults, for blocked dense matrix multiplication with implicit data movement.

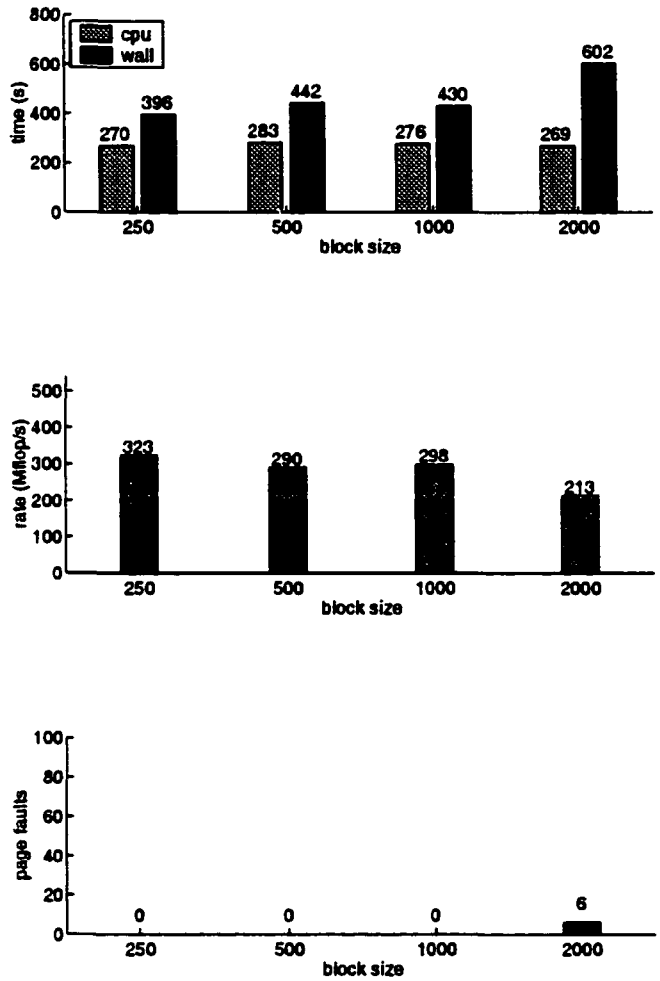


Figure 47: Execution time and rate, and page faults, for blocked dense matrix multiplication with explicit data movement.

two dimensions and we used implicit data movement. There were b^2 dgemm calls for a particular block size b. We used several values for b, chosen such that the multiplication of two blocks does not require more storage than the core size.

In the third and final experiment we blocked the matrix multiplication of the order 4,000 matrices just like in the previous experiment but this time we used explicit data movement. We used the same block sizes as in the previous experiment, thus the number of BLAS dgemm calls was the same.

The results are shown in Figures 45, 46 and 47. Note how the coarse granularity of the implicit data movement determines a lower rate. Moving data explicitly seems thus to be a much better solution. In addition, it may be better to call BLAS as well as LAPACK [5] kernels because these are well optimized for particular architectures.

CHAPTER VI
CONCLUSION

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. For both scenarios, we provided analytical results for model problems, and experimental results from simulation for irregular problems from computational partial differential equations and linear programming.

For the R1/W1 scenario the results show that for problems with good separators, such as those coming from the discretization of partial differential equations, ordered with nested dissection, right-looking and multifrontal factorization perform slightly better than left-looking factorization. However, we identify cases in which multifrontal is a bad choice since it requires too much temporary storage. 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.

For the RM/WM scenario, the most common case in external memory factorizations for large-scale problems, the 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. 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 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 also written a direct solver called Oblio 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 order to achieve good performance on a large range of problems an out-of-core solver should provide various algorithmic options, such as left-looking, right-looking, multifrontal, as well as hybrids. Also, one should be given the possibility to choose between 1-d and 2-d blocks. The code becomes thus very sophisticated. Further complications are determined by the data movement. 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) 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] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, Hammarling S., A. McKenney, and D. Sorensen. *LAPACK User's Guide.* SIAM, 1999.

[6] Anonymous. U.S. has sparse-matrix gap. *HPCC Week*, pages 9–10, December 1998.

[7] E. Arge, A. M. Bruaset, and H. P. Langtangen. *Modern Software Tools for Scientific Computing.* Birkhäuser, 1997.

[8] C. Ashcraft. Personal comunication.

[9] C. Ashcraft, R. G. Grimes, and J. G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM Journal on Matrix Analysis and Applications*, 20(2):513–561, October 1998.

[10] G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, 1994.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[12] T. H. Cormen and D. M. Nicol. Performing out-of-core FFTs on parallel disk systems. Technical report, ICASE, December 1996.

[13] M. Dæhlen and A. Tveito, editors. *Numerical Methods and Software Tools in Industrial Mathematics.* Birkhäuser, 1997.

[14] J. W. Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997.

[15] F. Dobrian, G. K. Kumfert, and A. Pothen. Object-oriented design for sparse direct solvers. In D. Caromel, R. R Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 207–214. Springer-Verlag, 1998.

[16] 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.

[17] J. Dongarra, S. Hammarling, and D. W. Walker. Key concepts for parallel out-of-core LU factorization. Technical report, University of Tennessee, April 1996.

[18] I. Duff and J. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9(3):302–325, September 1983.

[19] B. Eckel. *Thinking in C++.* Prentice Hall, 1995.

[20] M. A Elis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[21] A. George, J. R. Gilbert, and J. W. H. Liu, editors. *Graph Theory and Sparse Matrix Computation.* Springer-Verlag, 1993.

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

[23] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 3rd edition, 1996.

[24] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[25] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, 1996.

[26] 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.

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

[28] G. K. Kumfert. *An Object-Oriented Algorithmic Laboratory for Ordering Sparse Matrices.* PhD thesis, Old Dominion University, 2000.

[29] H. P. Langtangen, A. M. Bruaset, and E. Quak. *Advances in Software Tools for Scientific Computing.* Springer-Verlag, 2000.

[30] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, April 1979.

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

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

[33] 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.

[34] 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.

[35] 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.

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

[37] J. W. H. Liu, E. G. Ng, and B. W. Peyton. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 14(1):242–252, January 1993.

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

[39] S. Meyers. *More Effective C++: 35 New Ways to Improve your Programs and Designs.* Addison-Wesley, 1996.

[40] S. Meyers. *Effective C++: 50 Specific Ways to Improve your Programs and Designs.* Addison-Wesley, 2nd edition, 1998.

[41] 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.

[42] M. L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic.* SIAM, 2001.

[43] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann, 1998.

[44] A. Pothen and C. Sun. A distributed multifrontal algorithm using clique trees. Technical report, Pennsylvania State University, August 1991.

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

[46] E. Rothberg and A. Gupta. An evaluation of left-looking, right-looking and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines. Technical report, Stanford University, August 1991.

[47] 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.

[48] W. R. Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1993.

[49] B. Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

[50] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1997.

[51] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation.* Prentice Hall, 1997.

[52] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, 1983.

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

[54] 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.

[55] L. N. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, 1997.

[56] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence*, (3):331–360, 1991.

[57] J. S. Vitter and A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2):110–147, August 1994.

[58] J. S. Vitter and A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2):148–169, August 1994.

VITA

Florin Dobrian was born and raised in Bucharest, Romania. After graduating from high school in 1983, he enrolled in the Bucharest Polytechnic Institute. In 1989, he earned a Bachelor's degree in Science with a major in Computer Science. After receiving his undergraduate degree he worked in the software industry for five years. In 1994 he was accepted as a graduate student in the Computer Science Department at Old Dominion University, in Norfolk, Virginia. He completed all course requirements for a doctorate degree in 1997, then he spent four years doing research.

The Department of Computer Science can be contacted by mail, telephone, or e-mail.

Department of Computer Science
Old Dominion University
Norfolk, VA 23529
(757)683-3915