Parallel Sparse Cholesky Factorization

Hjalmtyr Hafsteinsson
Ph. D. Thesis

TR 88-940
September 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
PARALLEL SPARSE CHOLESKY FACTORIZATION

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Hjálmtýr Hafsteinsson

January 1989
PARALLEL SPARSE CHOLESKY FACTORIZATION

Hjálmtýr Hafsteinsson, Ph.D.
Cornell University 1989

As sequential computers seem to be approaching their limits in CPU speed there is increasing interest in parallel computers. This development calls for parallel algorithms for problems that may already have efficient sequential algorithms.

The problem of solving a linear system of equations arises in many areas of science and engineering. Quite often each equation only involves a small number of the variables. In that case the linear system is sparse. If we can take advantage of the sparsity we can solve much larger systems.

Consider the linear system $Ax = b$, where $A$ is a sparse symmetric positive definite matrix. A common approach to solving this system is to use Cholesky factorization. Algorithms that solve sparse linear systems using the Cholesky factorization usually consist of the following steps.

First the matrix $A$ is permuted in order to get a sparse Cholesky factor $L$. Then there is the symbolic factorization of $A$ to determine the nonzero structure of $L$. Finally the value of $L$ is calculated in the numeric factorization phase and $x$ is computed by solving the triangular systems $Ly = b$ and $L^T x = y$. 
In this thesis we present parallel algorithms for all the above steps except for the permutation step.

Before the symbolic factorization we compute the elimination tree of $A$. Elimination trees have many applications in sparse matrix computations. Therefore our parallel algorithm to find elimination trees is important in its own right. The algorithm we present for symbolic factorization then uses the elimination tree to compute the nonzero structure of $L$.

We next present a parallel algorithm to compute the numeric factorization of $A$. It runs in time proportional to the height of $A$'s elimination tree times a log factor. We also show how that algorithm can be converted into an NC algorithm (i.e., an algorithm that runs in polylogarithmic time) by the use of fast algorithms for dense matrices.

Finally we demonstrate a parallel algorithm to solve sparse triangular systems of equations. There again we show a version that runs in time related to the height of the elimination tree and a version that is an NC algorithm.
Biographical Sketch

Hjálmtýr Hafsteinsson was born on September 13th 1959 in Reykjavík, Iceland. He was a rather normal child, showing a healthy interest in sports and a healthy disinterest in too much studying. However as he grew older his priorities changed and he realized that as long as he was in school he would not have to go out and get a job.

Hjálmtýr graduated from the Commercial College of Iceland in 1980, which, despite its grand name, is a business high school. He then entered the University of Iceland to study Business Administration, but changed his mind and transferred to Computer Science, where he was awarded a B.S. degree in the spring of 1984. In the fall of that same year Hjálmtýr crossed the Atlantic Ocean to enroll in the Computer Science graduate program at Cornell University. The Graduate School saw fit to award him a Master of Science degree in 1987. In the fall of 1988 he will finally return to his native country to take a faculty position at the University of Iceland.
To my parents, Ágústa Hjalmtýsdóttir and Hafstein Sigurdsson.
Acknowledgements

First of all I would like to thank my advisor John Gilbert. His influence on me has been great. He taught me a great deal about how to write clearly, how to present my work, and how to do research. John was always very generous with his time and he never discouraged me, but gently turned me the right way when I was off on some wild-goose chase in my research.

Earl Zmijewski proofread the thesis and was untiring in correcting my sometimes awkward and ungrammatical sentences. He practically became my second advisor this last year, when John was away. I would have had a tough time without him. Thanks, Earl.

I would like to thank Gianfranco Bilardi and Michael Todd for serving on my committee.

I would like to thank the teachers in the Computer Science department of the University of Iceland. In particular Oddur Benediktsson and Sven Th. Sigurdsson. Their dedication to the department and high standard of teaching inspired me and encouraged me to always do my best. It will be an honor to join them on the faculty of the University of Iceland.
I lived in the International Living Center my entire time here at Cornell. It became my home away from home and I will take with me many fond memories from my stay there.

Many thanks to my office mates, Alex Aiken, Bruce Hendrickson, Ken Kane, Andy Meltzer, Michael Schwartzbach, and Carolyn Turbyfill, for keeping my company these four years. Also, to my weight lifting comrades Alex, Bruce, Andy, David Basin, and Pat Xavier: thanks for all the spots. Hey guys, whenever you are going over to Europe, stop by in Iceland...

My friends and family back in Iceland have my gratitude for keeping in touch. They quite often managed to relieve my attacks of homesickness.

Finally I would like to thank Stephanie Smith for her love and support. She was always there when I needed someone to talk to.
# Table of Contents

1 Introduction ............................................. 1
  1.1 Model of Computation .................................. 5
  1.2 Notation ............................................. 7
  1.3 Sequential Cholesky Factorization ................. 11
  1.4 Ordering ............................................ 13

2 Elimination Trees ....................................... 16
  2.1 Finding Elimination Trees ......................... 20
  2.2 Adding Fill Edges to $H_R$ ......................... 23
  2.3 Merging Elimination Forests .................... 25
  2.4 An Example ........................................ 26
  2.5 Timing Analysis .................................... 28
  2.6 Restricting the Memory Size .................... 29
  2.7 Proof of Correctness ............................. 30

3 Symbolic Factorization .................................. 37
  3.1 Finding the Fill .................................. 38
  3.2 Restricting the Memory Size .................... 44
  3.3 An Example ........................................ 45
  3.4 Proof of Correctness ............................. 47

4 Numeric Factorization ................................... 52
  4.1 Parallel Cholesky Factorization .................. 53
    4.1.1 An Example of Parallel Cholesky Factorization 59
  4.2 Saving Processors ................................. 59
  4.3 Pipelining ......................................... 64
  4.4 An NC Algorithm .................................. 68
    4.4.1 An Sample Execution of the NC Algorithm ... 74
  4.5 Questions about Elimination Trees ............... 77
5 Triangular System Solving 83
  5.1 Forward and Back Solve ............................................. 84
    5.1.1 An Example ............................................................ 89
  5.2 Reducing the number of Processors .................................... 91
  5.3 NC Algorithms ............................................................. 93
    5.3.1 An Example ............................................................ 95

6 Conclusion 98
  6.1 Summary ................................................................. 98
  6.2 Future Work ............................................................. 100

Bibliography 103
List of Figures

1.1 A matrix and its corresponding graph .................................. 8
1.2 Sequential Cholesky factorization ...................................... 11
1.3 Routines cmod and cdiv .................................................. 12

2.1 A matrix and its corresponding graph and elimination tree .... 18
2.2 The edge $(i, j)$ causes fill edges $(k_1, j)$, $(k_2, j)$, and $(k_3, j)$ . 19
2.3 Finding elimination trees ............................................... 21
2.4 Situation that causes a fill edge between $j$ and $q$ .............. 22
2.5 An example of elimination tree computation ...................... 27
2.6 The edge $(x, v_{min})$ will not be forgotten ....................... 30
2.7 Vertex $j$ will become $i$'s parent in the elimination tree ... 32
2.8 Vertex $k$ caused the fill edge between $i$ and $j$ ............... 33
2.9 A path of original edges between $i$ and $j$ ....................... 34

3.1 Two nontree edges with the same ancestor vertex ............... 40
3.2 Stage 1 of the symbolic factorization algorithm ................ 41
3.3 Stage 2 of the symbolic factorization algorithm ................ 43
3.4 The situation after Stage One ......................................... 46

4.1 Parallel Cholesky factorization ...................................... 55
4.2 Column $j$ has to be modified by a multiple of column $i$ ....... 56
4.3 Row $i$ is summed and the result subtracted from $(i, j)$ ....... 58
4.4 An example of parallel Cholesky factorization .................... 60
4.5 An elimination tree for a two-dimensional finite element graph .. 62
4.6 The submatrix $H$ corresponds to the leaf path $P$ ............ 70
4.7 The matrix $R$ is at most $h \times h$ ................................ 71
4.8 The $u$-leaf paths of the elimination tree ....................... 75
4.9 The $R$ matrices that come up in the example ................... 76
4.10 Making a chain more cliquish ...................................... 81

5.1 Parallel forward solve ............................................. 85
5.2 Routines $f_{sol}$ and $f_{sub}$ ....................................... 86

ix
5.3 Parallel back solve ........................................ 88
5.4 A sample execution of parallel back solve ............ 90
5.5 The NC forward solve algorithm ......................... 94
5.6 A sample execution of the NC back solve algorithm 96
Chapter 1

Introduction

Lately there has been growing interest in parallel computers. Such computers have more potential for solving the increasingly large instances of problems that occur in science and engineering. For instance, there are geodesic problems that involve several million equations [Kol78]. Sequential computers are approaching their limits in terms of CPU speed. The speed of light is becoming a constraining factor. Therefore, there is a need for parallel algorithms for problems that already may have efficient sequential algorithms. Quite often, radically new methods have to be devised to solve these problems in parallel.

The solving of a linear system of equations arises in a wide variety of important problems. If the problem has some sort of locality, which often is the case, then the resulting linear system is sparse. For example, the matrices that arise in the finite element method are sparse, because each point is only related to the ones around it. Taking advantage of the sparsity allows us to solve larger systems than
we would otherwise be able to. Sometimes a sparse system has structure that can be exploited to get very efficient algorithms (e.g., banded systems), but frequently the sparsity has no obvious structure. We therefore need methods that work well on general sparse systems.

Consider the problem of solving the linear system $Ax = b$, where $A$ is an $n \times n$ sparse symmetric positive definite matrix. The fastest direct method on a single processor is to use Cholesky factorization to find a lower triangular matrix $L$, such that $A = LL^T$. Then the solution to the original system can be found by solving $Ly = b$ and $L^Tx = y$. Assuming no exact numeric cancellation, the lower triangular $L$ will have nonzeros in the same positions as the lower triangular part of $A$ plus some extra nonzeros. These new nonzeros, which are introduced in the factorization, are called fill. The Cholesky factorization of a sparse matrix usually involves two steps [GL81]. First is the symbolic factorization, which finds the locations that will become nonzero during the factorization. This enables us to efficiently allocate processors and space for the second step, numeric factorization. To produce a sparse $L$ and thereby save time and space, the rows and columns of $A$ are commonly permuted before the factorization.

Many linear systems that arise in practice are symmetric positive definite. The fact that the system is positive definite ensures that the factorization is numerically stable without the need for pivoting [GVL83]. This allows us to use symmetric permutation on the matrix to change its nonzero structure, which in turn affects the efficiency of the algorithm.

Here we present a highly parallel and efficient algorithm for sparse Cholesky
factorization using a shared memory CRCW PRAM model of computation. The number of processors needed in the algorithm is proportional to the amount of memory used. This is important for several reasons. First, this is the case in current highly parallel computers such as the Connection Machine and the AMT DAP. In addition, memory and processors are now both made of the same VLSI technology, which means that this will probably continue to apply in future highly parallel computers. Second, even for dense linear systems, the fastest known algorithms that use more processors than space are either iterative algorithms that do not compute an exact solution [PR87], or are numerically unstable and inefficient in processor utilization [PS78]. These fast algorithms can, however, be used to obtain theoretical speedups of our numeric Cholesky factorization algorithm. Finally, it does not seem likely that algorithms that use more processors than space will ever be practical to use. For instance, we will probably never use a million processors to solve a $100 \times 100$ linear system.

Our main result is an algorithm that performs sparse Cholesky factorization with processors linear in space and in $O(h \log n + \log^2 n)$ time, where $h$ is the height of the elimination tree of the matrix $A$. We describe this tree in detail in Chapter 2. The height of the tree can range from 0 to $n - 1$ depending on the nonzero structure of $A$. If $A$ is full then $h = n - 1$, so the algorithm takes $O(n \log n)$ time with $n^2$ processors, which is only a log factor slower than a systolic algorithm specifically designed for dense matrices [OS85]. However, we also present a pipelined algorithm that eliminates this log factor in the time bound for dense matrices. If the graph of $A$ and its subgraphs have $O(n^\alpha)$-separators, then $A$ can be permuted so that $h =$
$O(n^\alpha)$ and the algorithm takes $O(n^\alpha \log n)$ time with $\max(n \log n, n^{2\alpha})$ processors. In practice such graphs are common. For instance, planar graphs have $O(\sqrt{n})$-separators. The chief practical significance of this algorithm is that it is applicable to a matrix of arbitrary sparsity structure. In this case the factorization will be efficient if a good preordering of $A$ can be found, and effective heuristics are well-known to produce good preorderings of arbitrary matrices [GL81].

In the remainder of this chapter we first describe our parallel model of computation. We then cover our notation and some basic definitions that will be useful later in the thesis. Next we review sequential sparse Cholesky factorization. Finally we look at how a matrix can be permuted before the start of our factorization algorithm. In our case it is important that a permutation can reduce not only the amount of fill, but also the height of the elimination tree.

In Chapter 2 we describe the elimination tree and present a parallel algorithm to find it. The elimination tree has important applications in many areas of sparse matrix factorization, so it is useful to have an efficient algorithm to compute it. We will then use the elimination tree in Chapter 3 to find the fill that will occur in the factorization. That will give us the sparsity structure of $L$.

In Chapter 4 we describe a numeric Cholesky factorization algorithm that runs in time proportional to the height of the elimination tree. We then consider faster versions of the algorithm and show how the algorithm can be optimized for special classes of matrices (graphs).

The last step in solving a linear system using Cholesky factorization is to solve two triangular systems. We show how that can be done in Chapter 5 using an
algorithm related to the one used to do the numeric factorization, which has the same time bounds. We also present a faster version of this algorithm.

Finally in Chapter 6 we present an overview, discuss applications, and look at open problems related to this work.

1.1 Model of Computation

In this thesis we will be using the shared memory PRAM model as our parallel model of computation (See [Vis83]). It consists of many independent processors, each of which has its own internal memory, along with a shared memory through which the processors communicate. All the processors run the same program. However the program can reference a processor's unique identification number, so each processor does not have to execute all the statements of the program. Each access to the shared memory takes only one unit of time.

PRAMs are classified according to whether or not several processors can access the same location in shared memory at the same time. The EREW PRAM does not allow simultaneous access to the memory at all. The CREW PRAM allows two or more processors to read the value of a location at the same time, but not to simultaneously write into a location. The least restrictive class is the CRCW PRAM, which allows both concurrent read and concurrent write. This is the class of PRAMs that we will use. It is the least realistic one, but an algorithm for CRCW PRAM that runs in $O(t)$ time using $p$ processors can be simulated on a EREW PRAM in $O(t \log p)$ time using $p$ processors [BH85].
The CRCW model is strictly more powerful than the CREW model. For instance, $n$ binary values can easily be OR-ed together in constant time using the CRCW PRAM, but the CREW PRAM needs $\Omega(\log n)$ time [CD82]. For many other important problems the fastest known algorithm for CREW requires a factor of $\log n$ more time than the fastest CRCW algorithm. One of these problems is graph connectivity. The fastest CREW algorithm takes $O(\log^2 n)$ time [HCS79], while on the CRCW the problem can be solved in $O(\log n)$ time [CV86].

In the case of CRCW PRAMs we have to decide what happens when two or more processors attempt to simultaneously write into the same memory location. We decree that an arbitrary processor succeeds in writing into the location. Thus, the algorithms have to be written so that it does not matter which processor succeeds if many are trying to write at the same time. Note that we can make sure that the other processors realize that they did not succeed. Let $l$ be a location that several processors are trying to write into at the same time. We first have the processors write their identification number. Then all the processors read location $l$ to see which one succeeded. The one processor that reads its own identification number can then write its value into location $l$.

We will also require that a processor in our CRCW PRAM model can spawn other processors. This will only be used in Chapter 3, when we do symbolic factorization. In that algorithm the number of processors depends on the output of the algorithm, so we can not possibly allocate the processors beforehand.

Let $S$ be a problem that can be solved in $O(t)$ time on a sequential RAM. Then a PRAM using $p$ processors cannot solve $S$ faster than $O(t/p)$ without improving
on the sequential algorithm. If a parallel algorithm solves $S$ in $O(t/p)$ time using $p$ processors, where $O(t)$ is the time for the best known sequential algorithm, then we say that the parallel algorithm has \textit{optimal time-processor product} with respect to the sequential algorithm.

\subsection{1.2 Notation}

In this section we cover the notation we will use in the thesis. Sparse matrix problems are commonly translated into problems in graphs theory. Thus, most of the notation we use is standard graph theoretic notation, but sparse matrix computation does have some definitions that are specific to that area. George and Liu’s book [GL81] is the standard reference for sparse symmetric positive definite linear system solving. See Harary [Har69] for an introduction to graph theory.

A \textit{graph} $G = (V, E)$ consists of a \textit{vertex set} $V$, and an \textit{edge set} $E$. We denote the cardinality of the vertex set by $n$ and that of the edge set by $m$. An edge is an unordered pair of vertices $(v, w)$. Vertices $v$ and $w$ are \textit{adjacent} if $(v, w) \in E$. The \textit{degree} of a vertex $v$ is the number of adjacent vertices. An \textit{ordering (labelling) $\alpha$} of $G$ is a mapping of $\{1, 2, \ldots, n\}$ onto $V$. The vertices of $G$ will be denoted by $v_i$ or $i$, where $i \in \{1, \ldots, n\}$, or by $v$ or $w$ when the labelling is irrelevant.

In this thesis we will be looking at algorithms for solving the linear system $Ax = b$, where $A$ is an $n \times n$ sparse symmetric positive definite matrix. We will associate with $A$ the \textit{graph of $A$}, $G(A) = (V(A), E(A))$. The graph $G(A)$ is undirected, since $A$ is symmetric. The vertices of $G(A)$ are numbered from 1 to $n$
Figure 1.1: A matrix and its corresponding graph

and \((v_i, v_j) \in E(A)\) if and only if \(a_{ij} = a_{ji} \neq 0\) when \(i \neq j\). The correspondence between \(A\) and \(G(A)\) can be seen in Figure 1.1. The graph \(G(A)\) represents the nonzero structure of \(A\).

Permuting the rows and columns of \(A\) by an \(n \times n\) permutation matrix \(P\) corresponds to relabelling the vertices of \(G(A)\).

A graph \(G' = (V', E')\) is a subgraph of \(G\) if \(V' \subseteq V\) and \(E' \subseteq E\). The subgraph \(G'\) is induced by \(V'\) if \(E'\) contains all the edges of \(E\) that have both endpoints in \(V'\). In matrix terms, if \(G'\) is the subgraph of \(G(A)\) induced by \(V'\) then \(G'\) is the graph of a matrix in which all the rows and columns not represented in \(V'\) are deleted from the matrix \(A\).

A subgraph \(G'\) is a clique if all its vertices are adjacent. A clique in \(G(A)\) corresponds to a full submatrix of \(A\).

A path in a graph \(G\) between \(v_1\) and \(v_k\) is a sequence of vertices \(v_1, v_2, \ldots, v_k\), such that \((v_i, v_{i+1}) \in E\) for all \(i\) from 1 to \(k - 1\) and these edges are distinct. The length of a path is the number of edges in it. A cycle is a path with at least 3 vertices all distinct, in which \(v_1 = v_k\). A graph is acyclic if it has no cycles. The
graph $G$ is *connected* if there is a path from every vertex to every other vertex. The maximal connected subgraphs of $G$ are called its *connected components*.

The vertices can be renumbered so that those in each connected component of $G(A)$ are consecutive. Then the corresponding permutation of $A$ is block diagonal and each block corresponds to a connected component of $G(A)$. Thus $A$ is irreducible if and only if $G(A)$ is connected.

A *separator* of $G$ is a subset of $V$ whose removal divides $G$ into two or more connected components.

In sparse Cholesky factorization the first step is often to find the nonzero structure of the triangular factor $L$, assuming that no coincidental numerical cancellation takes place during the factorization. We can always pick values for the matrix so that there is no numerical cancellation in the Cholesky factorization [Gil86]. Let $G^* = G(L + L^T)$ be the graph representing the nonzero structure of $L$. We can transform $G(A)$ into $G^*$ with the following one-person game, which is originally due to Parter [Par61].

Begin with all vertices unmarked. For each $i$ from 1 to $n$, add edges as necessary to make all unmarked neighbors of $i$ adjacent, and then mark $i$. The edges added to $G(A)$ are called *fill edges* and $G^*$ is called the *filled graph* of $G(A)$.

The number of fill edges that will have to be added depends on the order in which the vertices are marked (eliminated). We will consider the problem of finding a good ordering in Section 1.4. The following lemma characterizes the fill.
Lemma 1.1 (Path lemma) The edge \((u, w) \in G^*\) if and only if there is a path \(u = v_1, v_2, \ldots, v_k = w\) in \(G(A)\) with \(v_i < \min\{u, w\}\) for \(1 < i < k\).

That is, there is an edge \((u, w)\) in \(G^*\) if and only if there is a path in \(G(A)\) from \(u\) to \(w\) through vertices numbered lower than both \(u\) and \(w\).

In a directed graph \(G = (V, E)\) each edge is a directed pair \((v, w)\). We say that the edge \((v, w)\) is from \(v\) and to \(w\). The in-degree of a vertex \(v\) is the number of edges coming into \(v\). The out-degree of \(v\) is the number of edges coming from \(v\).

A directed tree \(T\) is an acyclic directed graph, in which one vertex \(r\) has out-degree 0 and all the other vertices have out-degree 1. The vertex \(r\) is called the root of \(T\). If the edge \((v, w)\) is in \(T\) then \(w\) is the parent of \(v\) and \(v\) is a child of \(w\). The root has no parent and the vertices with no children are called leaves. The height of a vertex \(v\) in \(T\) is the length of the longest path from a leaf to \(v\). The height of the tree is the height of its root. There is a unique path in \(T\) from any vertex \(v\) to \(r\). If the path from \(v\) to \(r\) goes through \(w\) then \(w\) is said to be an ancestor of \(v\) and \(v\) is a descendant of \(w\). A directed graph that consists of more than one vertex-disjoint directed tree is a directed forest.

We will use these definitions of directed trees in Chapter 2 when we talk about elimination trees.

All logarithms in this thesis are base 2; hence, \(\log n\) stands for \(\log_2 n\).
algorithm sequential-Cholesky
begin
    for $j := 1$ to $n$ do
        for $k < j$ with $(j, k)$ an edge of $G^*$ do
            $\text{cmod}(j, k)$
        od;
        $\text{cdiv}(j)$
    od
end

Figure 1.2: Sequential Cholesky factorization

1.3 Sequential Cholesky Factorization

We now review the sparse sequential column Cholesky factorization algorithm (See [GL81]). It transforms a sparse matrix $A$ into its Cholesky factor one column at a time, starting with column 1. Each column is computed using some of the previously computed columns. Figure 1.2 provides the algorithm. We assume that symbolic factorization has already taken place, so that the edges of $G^*$ are known.

The routines $\text{cmod}$ and $\text{cdiv}$ are in Figure 1.3. The procedure $\text{cmod}(j, k)$ modifies column $j$ by column $k$, where $k$ is assumed to be less than $j$. The procedure $\text{cdiv}(j)$ divides column $j$ by the square root of its diagonal element and $\text{cmod}(j, k)$ subtracts a multiple of column $k$ from column $j$, where the multiplier is $l_{jk}$. Each operation $\text{cdiv}(j)$ corresponds to a vertex $j$ in the filled graph $G^*$ and each $\text{cmod}(j, k)$ corresponds to an edge $(j, k)$ in $G^*$, with $j > k$. Thus, the $\text{cdiv}$ routine
algorithm $cmod(j, k)$
begin
  for each edge $(i, k)$ of $G^*$ with $i \geq j$ do
    $a_{ij} := a_{ij} - a_{jk} \cdot a_{ik}$
  od
end

algorithm $cdiv(j)$
begin
  $a_{jj} := \sqrt{a_{jj}}$
  for each edge $(i, j)$ of $G^*$ with $i \geq j$ do
    $a_{ij} := a_{ij}/a_{jj}$
  od
end

Figure 1.3: Routines $cmod$ and $cdiv$

is executed $n$ times and $cmod$ is executed $m^*$ times.

The time bound for this algorithm is dominated by the column modifications. Using the proper data structures, the time can be shown to be on the order of

$$\sum_{j=1}^{n} d_j^2,$$

where $d_j$ is the number of nonzeros in column $j$ of $L$. This is the same as the number of operations required to multiply $L$ and $L^T$ [GL81].
1.4 Ordering

The performance of the algorithm presented here to solve the sparse system $Ax = b$ depends critically on the height of $A$'s elimination tree $T$ and the amount of fill in the Cholesky factor $L$. The time bound for the whole algorithm is $O(h \log n + \log^2 n)$, where $h$ is the height of $T$, and the number of processors is $m^*$, where $m^*$ is the number of nonzeros in $L$. The height of $T$ determines the time needed by the algorithm and the amount of fill determines the number of processors required. Reordering the columns and rows of $A$ (or equivalently, renumbering the vertices of $G(A)$) affects both the height of the elimination tree and the amount of fill that occurs during the factorization. Thus, we would like to find a permutation of the matrix $A$ that gives both minimum fill and minimum elimination tree height. However, both of these problems are NP-complete [Yan81, Pot88a]. Most of the work so far in finding better orderings has concentrated on minimizing the fill. Since both problems are NP-complete, all the algorithms are either heuristics or only work well on certain classes of matrices.

The most commonly used ordering algorithm for minimizing fill is the minimum degree algorithm [GL87]. It has low overhead and often outperforms more sophisticated ordering strategies. If $G(A)$ has good separators then the nested dissection algorithm gives good results [GL81, LRT79]. A good separator is one that is small and divides the graph into components of roughly equal size. Nested dissection orderings also result in elimination trees of modest height. However, the minimum degree algorithm usually produces less fill than nested dissection.
If a graph $G(A)$ does not have good separators then the amount of fill will be large for any ordering. Most random graphs do not have good separators [LRT79]. Only graphs with some sort of structure do. Luckily many graphs that occur in practice have good separators, for instance two-dimensional finite element graphs, planar graphs, and graphs of bounded genus [LT79,GHT84].

Jess and Kees [JK82] devised an algorithm that finds a permutation giving the minimum height elimination tree over all permutations with the same fill structure. The time bounds for that algorithm have been improved by the algorithms of Liu and Mirzaian [LM87], Pothen [Pot88b], and Lewis and Peyton [LP88].

One possible strategy to reduce both the amount of fill and the height of the elimination tree is to use a two step approach. First a heuristic algorithm, like minimum degree or nested dissection, is used to reduce the fill. Then the algorithm of Jess and Kees can be used to find the minimum elimination tree while preserving the fill structure.

Recently Leiserson and Lewis [LL88] have designed a heuristic algorithm that finds an ordering resulting in a shallow elimination tree. Their algorithm, which is a generalization of the nested dissection algorithm, seems in practice to produce a lower elimination tree than the method of Jess and Kees.

All of the above ordering algorithms are sequential and there are few algorithms for finding good orderings in parallel. Nested dissection can be parallelized [GZ87] and we suspect that Liu and Mirzaian's algorithm to reduce the height of the elimination tree can also be parallelized.

The problem of finding orderings that effectively reduce both the height of the
elimination tree and the amount of fill remains an interesting challenge. Doing it efficiently in parallel looks even more challenging.

We conjecture that for any graph there exists an ordering that simultaneously achieves minimum elimination tree height and a number of nonzeros in $L$ that is within a factor of 2 of the minimum $m^*$. Nested dissection does this for graphs that are just a single chain of vertices, but finding an ordering like that for general graphs is very likely NP-complete.
Chapter 2

Elimination Trees

In this chapter we describe a parallel algorithm for finding elimination trees. In Chapter 3 we then use the elimination tree of a matrix $A$ to help us find the symbolic factorization of $A$. Elimination trees were first defined by Schreiber [Sch82] and have emerged as important data structures in many areas of sparse matrix factorization. They have applications in storing a sparse matrix [Liu86a] and in computing its Cholesky factorization both sequentially and in parallel [Zmi87, ZG88]. Liu's paper [Liu87b] contains a survey of the uses of the elimination tree in sparse factorization.

**Definition 2.1** Let $A$ be an $n \times n$ symmetric positive definite matrix and let $L = (l_{ij})$ be its Cholesky factor. The elimination tree of $A$ is a directed tree $T$, with vertices $1, 2, \ldots, n$ and an edge $(i, j)$ if and only if

$$j = \min \{k \mid l_{ki} \neq 0 \text{ and } k > i\}.$$
The vertex \( j \) is the parent of \( i \) in the tree if the first nonzero below the diagonal in column \( i \) is in row \( j \). The graph \( T \) is a tree because \( j \) is uniquely defined for each \( i \), and since \( j > i \) there are no cycles. Thus \( i \) is a child of \( j \) in \( T \). In general \( T \) is a forest, but then the matrix \( A \) is reducible. In that case \( A \) can be permuted so that it is block diagonal and each block factored independently. Therefore we assume that \( A \) is irreducible, which means that \( G(A) \) is connected and \( T \) is a tree.

In Figure 2.1 we show matrix \( A \), its filled graph \( G^* \), and its elimination tree \( T \). The lower triangular part of the matrix also contains the fill that will occur during the factorization. The corresponding fill edges of \( G^* \) are shown as dashed lines.

The tree \( T \) has \( n - 1 \) edges some of which are also in \( G \) (some might correspond to fill edges), but \( G \) has \( m \) edges, so there are some edges of \( G \) that do not correspond to edges of \( T \). We call them nontree edges. They will be useful in computing the fill in Chapter 3. The four nontree edges in Figure 2.1 are represented by dotted lines.

Note that the elimination tree \( T \) is defined in terms of nonzeros in the filled matrix \( L \). However, since we use \( T \) to compute \( L \), we need to find \( T \) from the original matrix \( A \) (or its graph \( G(A) \)). In fact, in the next chapter we are going to use the elimination tree to find \( L \)'s nonzero structure.

We now present a few properties of elimination trees. These results are mostly taken from Liu's survey [Liu87b] and are stated here without proof.

**Lemma 2.2** The elimination tree \( T \) is heap ordered; i.e., each vertex has a higher number than its children in the tree.
Figure 2.1: A matrix and its corresponding graph and elimination tree
Figure 2.2: The edge \((i, j)\) causes fill edges \((k_1, j), (k_2, j),\) and \((k_3, j)\)

**Lemma 2.3** If \((i, j) \in G\) and \(i < j\) then \(j\) is an ancestor of \(i\) in \(T\), and for all vertices \(k\) on the path from \(i\) through \(j\) in the tree, \((k, j) \in G^*\). All fill edges occur this way.

For the tree in Figure 2.2, which has a nontree edge between \(i\) and \(j\), Lemma 2.3 says that we will get the fill edges \((k_1, j), (k_2, j),\) and \((k_3, j)\). Note that even though the vertices \(k_3\) and \(j\) are adjacent in the elimination tree they need not be adjacent in \(G\). However, they are adjacent in \(G^*\). In Figure 2.1 we see that \((3, 5)\) is an edge of the elimination tree but it is not an edge of \(G\). Furthermore we see in Section 2.4 that it is not added as a fill edge during the computation of the tree.

**Lemma 2.4** The set of nodes in the subtree of \(T\) rooted at vertex \(i\) form a connected component in the subgraph of \(G(A)\) induced by all the vertices except for the ancestors of \(i\) in \(T\).
Consider the example in Figure 2.1. The subtree rooted at vertex 5 consists of vertices \{1, 2, 3, 4, 5\}. The ancestors of vertex 5 are 7 and 8. The vertices of the subtree form a connected component in the subgraph of \(G(A)\) consisting of vertices \{1, 2, 3, 4, 5, 6\}.

The best sequential algorithm to find elimination trees is by Liu [Liu86a]. It requires \(O(m \alpha(m, n))\) time, where \(\alpha\) is the functional inverse of Ackerman's function from Tarjan's disjoint set union analysis [Tar83]. Zmijewski and Gilbert [ZG88] give a parallel algorithm for a message-passing multiprocessor that takes

\[
O((m/p) \alpha(m/n, n) + n \alpha(n, n) \log p)
\]

time with \(p\) processors. That algorithm is efficient when \(p\) is small compared to \(n\).

Our algorithm, which is rather different from either of those, takes \(O(\log^2 n)\) time on \(m\) processors, and is thus asymptotically superior to that of [ZG88].

2.1 Finding Elimination Trees

Our algorithm to find the elimination tree \(T\) uses a divide-and-conquer approach. Because the algorithm will split the vertices of the graph into two groups at each level, the resulting subgraphs will not necessarily be connected. For that reason the algorithm will be working with elimination forests, but the final result will be an elimination tree.

The algorithm will be presented in terms of the graph \(G(A)\). It will take as input a copy of the graph \(G(A)\) and start by splitting the vertices into two groups. One group will contain the vertices 1 through \(n/2\) and the other will contain vertices
Figure 2.3: Finding elimination trees

\( n/2 + 1 \) through \( n \). The algorithm then recursively finds the elimination forests for the two groups and merges them. However, before recurring on the second group we need to add to it the fill edges that are caused by the first group. The algorithm is given in Figure 2.3.

At some level in the algorithm we have a subgraph \( H \) that consists of all the vertices from \( h \) through \( k \) and the edges from \( G(A) \) that have both their endpoints in that set plus some fill edges that have been added at an earlier level. If \( H \) contains more than one vertex we split it up so that \( H_L \) is the subgraph consisting of the vertices from \( h \) through \( l \) and the edges within that set and \( H_R \) is the subgraph of vertices from \( l + 1 \) through \( k \) and associated edges, where \( l \) is the
Figure 2.4: Situation that causes a fill edge between $j$ and $q$

vertex that is halfway between the vertices $h$ and $k$.

Before we recur on $H_L$ and $H_R$ we need to add to $H_R$ enough fill edges resulting from the elimination of the vertices of $H_L$, so that the resulting graph $H'_R$ will have an elimination forest that is an induced subgraph of the correct elimination forest for $H$. For instance if $(i, j)$ and $(p, q)$ are edges between $H_L$ and $H_R$ (i.e., $i, p \in H_L$ and $j, q \in H_R$) and there is a path in $H_L$ between $i$ and $p$ then, by the Path lemma (Lemma 1.1), $(j, q)$ is a fill edge in $H'_R$ (see Figure 2.4). We will not add all these edges, only the ones between the smallest vertex in $H_R$ adjacent to each connected component of $H_L$ and all other vertices of $H_R$ adjacent to this component. We will show that this is adequate to ensure that $H'_R$ has an elimination forest that is an induced subgraph of the correct elimination tree for $G(A)$. When these fill edges have been added to $H_R$ to make $H'_R$ we can recur on $H'_R$. We recur on $H_L$ and $H'_R$ in parallel, computing the elimination forests for each of them. These two forests are then merged, giving us an elimination forest for $H$.

There are now two things we have to show how to do: adding the fill edges caused by $H_L$ to $H_R$ and merging the elimination forests for $H_L$ and $H'_R$. 
2.2 Adding Fill Edges to $H_R$

The key idea used in adding the fill edges to $H_R$ is the fact that the vertices of $H_R$ that are adjacent to the same connected component of $H_L$ will eventually form a clique in the filled graph $G^*$. This is a consequence of the Pathlemma from Chapter 1. However, we do not have to add all those edges to $H_R$ to get $H'_R$, only enough so that $H'_R$ will have the right elimination forest.

To implement this idea we first form a new graph $\tilde{H}_L$. Its vertex set $V(\tilde{H}_L)$ is $V(H_L) \cup \{ v_u \mid \text{there is an edge } (u,v) \in H \text{ with } u \in V(H_L) \text{ and } v \in V(H_R) \}$. The edge set $E(\tilde{H}_L)$ is $E(H_L) \cup \{ (u,v_u) \mid u \in V(H_L) \}$. Therefore $\tilde{H}_L$ consists of all the vertices and edges of $H_L$, along with an additional vertex and an edge for each edge between $H_L$ and $H_R$. These additional vertices will have degree 1 in $\tilde{H}_L$. In other words, we form $\tilde{H}_L$ from $G$ by first removing all vertices and edges not incident on vertices of $H_L$. Then, for each vertex $v \in H_R$ of degree $d(v)$ we replace it with $d(v)$ vertices of degree 1.

We now use Cole and Vishkin's connected components algorithm [CV86] on $\tilde{H}_L$. It runs in $O(\log(|V(\tilde{H}_L)|))$ time using

$$(|E(\tilde{H}_L)| + |V(\tilde{H}_L)|)\alpha(|E(\tilde{H}_L)|,|V(\tilde{H}_L)|)/\log(|V(\tilde{H}_L)|)$$

processors, where $\alpha$ is the inverse of the Ackerman function. Note that whenever some $v_u$-vertices of $\tilde{H}_L$ are in the same connected component, their associated $v$-vertices of $H'_R$ will be a clique of fill edges in $G^*$.

Next we want to find the lowest numbered vertex in $H_R$ adjacent to each connected component of $H_L$. To do that we allocate an array MIN that has the same
size as the number of components in $\hat{H}_L$. Each element of the array consists of $\log(|V(H_R)|)$ bits. All the elements of MIN are initialized to contain the bit sequence $11\cdots1$. We also assign a processor $P_{vu}$ to each $v_u$-vertex. Each component is assigned a unique integer. Initially each processor $P_{vu}$ looks at the first (most significant) bit in the binary representation of the number of the associated $v$-vertex of $H_R$. If it is 0 then $P_{vu}$ writes 0 in the first bit position of MIN[$k$], where $k$ is the component number of a $v_u$-vertex. Each processor then reads the first bit of MIN[$k$] and if they read a 0, then those processors that did not try to write 0 will stop. The rest of the processors then do the same for the second bit of MIN[$k$] and so on.

After $\log(|V(H_R)|)$ iterations of this, the processors still running for a particular $k$ all have the same associated $v$-vertex in $H_R$ and the bit sequence in MIN[$k$] is the binary representation of that $v$. Thus that $v$ is the minimum vertex in $H_R$ adjacent to component $k$ of $\hat{H}_L$. Finally we form $H'_R$ from $H_R$ as follows. Each processor $P_{vu}$ that stopped during the $\log(|V(H_L)|)$-iteration looks at the number in MIN[$k$], for its respective $k$. It then adds an edge between the $v$-vertex represented in MIN[$k$] and $P_{vu}$'s associated $v$-vertex in $H_R$.

Now for each connected component $k$ of $H_L$, the smallest vertex in $H'_R$ that is adjacent to $k$ has an edge to all the other vertices in $H'_R$ that are also adjacent to $k$. We will show in Section 2.7 that this will ensure that the elimination forest of $H'_R$ will be an induced subgraph of the elimination tree of $G$.

The total time for adding these fill edges to $H_R$ is $O(\log(|V(\hat{H}_L)|))$, which is equivalent to $O(\log(|V(H_L)|))$, and the maximum number of processors used is
\[ |E(\mathcal{H}_L)|, \text{ which is no more than } |E(H)|.\]

### 2.3 Merging Elimination Forests

We now have the elimination forests \( F_L \) and \( F_R \) for \( H_L \) and \( H'_R \) respectively and we want to merge them to get an elimination forest for \( H \).

We do this by adding at most one edge from the root of each tree in \( F_L \) to a vertex of a tree in \( F_R \). For each tree \( S \) in \( F_L \) we add an edge from the root of \( S \) to vertex \( v \) of \( F_R \), where \( v \) is the smallest node in \( H'_R \) that is adjacent to a node in \( S \). Note that all the vertices in \( H'_R \) that are adjacent to \( S \) are in the same tree of \( H'_R \) and in fact they form a chain in that tree, with the lowest numbered vertex at the bottom of that chain.

In the previous section we showed how to find the minimum node of \( H_R \) connected to each connected component of \( H_L \). Each tree in \( F_L \) corresponds to a connected component of \( H_L \). Thus we want to hook the root of each tree in \( F_L \) onto the node of \( H_R \) we found for that component when we were adding the fill edges to \( H_R \). We found those nodes before going down the recursion, so we need not compute them again.

The trees can be merged in constant time, since no computation is required. We already know which trees in \( H_L \) should be hooked onto which nodes in \( H'_R \).
2.4 An Example

Let us now examine how the algorithm computes the elimination tree in Figure 2.1. In what follows refer to Figure 2.5. The first step is to split the vertices of $G$ up into $H_L = \{1, 2, 3, 4\}$ and $H_R = \{5, 6, 7, 8\}$. The next step is to add fill edges to $H_R$. That entails finding the connected components of $H_L$. In our example they are $\{1, 3\}$ and $\{2, 4\}$. For each component, the lowest numbered vertex in $H_R$ adjacent to it is 5 (5 is adjacent to 1 and 4). Since 7 and 8 are the other vertices in $H_R$ that are adjacent to the two components of $H_L$, we add fill edges between vertices 5 and 7 and vertices 5 and 8. (See Figure 2.5 (a))

The next step of the algorithm is to work recursively on $H_L$ and $H'_R$ (see Figure 2.5 (b). Applying the algorithm recursively to $H_L$ produces a new $H_L$ that consists of the one-vertex components $\{1\}$ and $\{2\}$. Each of them only has one neighbor on the right hand side. The recursion stops at the next level and coming back up the merging phase will hook vertex 1 onto vertex 3 and vertex 2 onto vertex 4 (see Figure 2.5 (c)).

Applying the algorithm recursively to the top-level $H'_R$ there are again two connected components in the new $H_L$, $\{5\}$ and $\{6\}$. The vertex 5 has vertices 7 and 8 as its neighbors, so those vertices should get a fill edge, but they already have an edge. In the next round vertices 5 and 6 are not hooked up, but vertex 7 is hooked up to vertex 8. Then in the merge phase going up, since vertex 7 was the lowest numbered vertex on the right hand side that vertices 5 and 6 were adjacent to, 5 and 6 get hooked onto vertex 7.
Figure 2.5: An example of elimination tree computation
The elimination forest we have so far is in Figure 2.5 (c). Going up one more level in the recursion the elimination trees whose roots are vertices 3 and 4 get hooked onto vertex 5. This is because 5 was the lowest numbered vertex adjacent to 3 and 4's respective components. Now finally we have the elimination tree of Figure 2.1.

2.5 Timing Analysis

In the divide-and-conquer algorithm there are $\log n$ levels, each involving subgraphs with half as many vertices as the one before. On the way down we add fill edges, caused by $H_L$, to $H_R$. That takes time $O(\log(|V(H_L)|))$. In the beginning $|V(H_L)|$ is $n/2$ and it is halved at each level. Thus the total time on the way down the recursion is

$$
\sum_{i=1}^{\log n} \log \frac{n}{2^i} = O(\log^2 n).
$$

The number of processors used is $|E(H)|$ at each level. Initially $|E(H)| = |E(G)|$. At each level the sum of all the $|E(H)|$'s on that level is at most $|E(G)|$. Hence the total number of processors used going down the recursion is of the order $|E(G)|$.

In coming up the recursion we use $O(1)$ time at each level to merge the elimination trees with no more than $|E(G)|$ processors at each level.

Combining these numbers we find that the time for finding the elimination tree for $G$ is $O(\log^2 n)$, using $m$ processors.
2.6 Restricting the Memory Size

With $O(n^2)$ space, we can store the adjacency matrix of $G$, which allows us to add a new edge or test for an edge in constant time. To restrict the amount of space to $O(m)$ for finding the elimination tree, we keep a list of the edges of $G$ in memory, with a processor assigned to each edge. We use an additional $O(n)$ space and processors to store and manipulate the elimination tree.

The only tricky part is adding edges to the graph. Each time some edges are added, we will sort the list of old and new edges to remove duplicates and reassign processors to edges. The sort takes $O(\log n)$ time with $m$ processors using Cole's parallel merge sort [Col86].

To analyze the overall running time, we need to go into more detail on how the edges are added.

The algorithm in Figure 2.3 must add edges to the graph in the procedure $addfilltoH_R$. After the connected components of $H_L$ are found, the processors $P_{uv}$ (corresponding to edge $(u, v)$, with $u \in H_L$ and $v \in H_R$) for each connected component find the smallest vertex in $H_R$ adjacent to their component, call it $v_{min}$. Then the algorithm adds fill edges from $v_{min}$ to all the vertices on the right hand side that had edges into $u$'s component. Now, each $P_{uv}$ for that component with $v \neq v_{min}$ assigns itself the edge $(v_{min}, v)$, forgetting edge $(u, v)$. Then all the edges, old and new, are sorted to remove duplicates.

Notice that this procedure forgets some edges forever. The forgotten edges do not affect the elimination tree, however. For each edge $(u, v)$ that is forgotten,
there is an edge \((x, v_{\text{min}})\) with \(x \in H_L, v \in H_R, v_{\text{min}} < v\), and \(x\) in the same component of \(H_L\) as \(u\) (see Figure 2.6). After the recursive call, then, the root of the tree in \(F_L\) containing \(x\) and \(u\) will become a child of \(v_{\text{min}}\), whether or not edge \((u, v)\) exists.

The algorithm sorts the edges \(O(\log n)\) times. Therefore the total time spent sorting is \(O(\log^2 n)\), and the entire algorithm still takes \(O(\log^2 n)\) using \(m + n\) processors.

### 2.7 Proof of Correctness

To prove the correctness of the algorithm we first need the following lemma.

**Lemma 2.5** If the algorithm produces correct elimination forests \(F_L\) and \(F_R\) for \(H_L\) and \(H'_R\) respectively, then they are induced subgraphs of the correct elimination forest for \(G\).
Proof: We will first show that $F_R$ is an induced subgraph of the correct elimination forest for $G$. In other words, if $j$ is the parent of $i$ in the correct elimination forest and $i$ and $j$ are in $H_R$ then $j$ is also the parent of $i$ in $F_R$.

According to the definition $j$ is then the lowest numbered vertex greater than $i$ that is adjacent to it in $G^*$. Now, if $i$ and $j$ were adjacent in $G$ then they would also be adjacent in $H_R$. Then as we go down the recursion, we will eventually split the vertices up so that $i$ will be on the left side (in the current $H_L$) and $j$ will be on the right (the current $H_R$). Coming back up to that level, $i$ will be a root of an elimination tree on the left side and $j$ will be the smallest vertex adjacent to $i$'s component (see Figure 2.7). Since $j$ is the parent of $i$ in the correct elimination tree we know that there are no vertices between $i$ and $j$ adjacent to $i$'s component. Thus, $j$ must be the lowest numbered vertex in $H'_R$ adjacent to $i$'s component and $i$ must be the highest numbered vertex in its connected component, so it will become the root of its tree. Therefore $i$ will get hooked up to $j$ in the merge phase of that level of the algorithm.

If $i$ and $j$ were not adjacent in $G$, i.e., the edge between them is a fill edge, then there must be a vertex $k$ ($k < i < j$), such that the elimination of $k$ in the elimination game caused a fill edge to go between $i$ and $j$. If $(k,i)$ and $(k,j)$ are not fill edges then there are 3 cases: (i) $k$ is in $H_R$, (ii) $k$ is in $H_L$, and (iii) $k$ is smaller than the vertices in $H_L$.

If case (i) applies then as we continue going down the recursion we will eventually split the three vertices up. If $k$ is in the left half and $i$ and $j$ in the right, as in Figure 2.8, then we will find the smallest vertex $v$ in the right half that is adjacent
Figure 2.7: Vertex $j$ will become $i$'s parent in the elimination tree

to the connected component that $k$ is in and add edges between $v$ and all the other vertices on the right that are adjacent to $k$'s component. If $v = i$ then we are done because we just added an edge between $i$ and $j$ and as we showed above that will eventually result in the algorithm setting $j$ as $i$'s parent. If $v \neq i$ then we repeat this argument with $v$ instead of $k$. Now, if the split-up leaves $k$ and $i$ on the left side and $j$ on the right, then we see that $i$ will be a root of an elimination tree and $j$ will be the lowest numbered vertex on the right side adjacent to $i$'s tree, so $i$ will get hooked up to $j$ in the merge phase of this level of the algorithm. The reason is that there are no vertices between $i$ and $j$ adjacent to $i$'s tree, since $j$ is the parent of $i$ in the correct forest.

Case (ii) can be handled similarly to case (i) above. Here we have already split vertices up so that $k$ is on the left side and $i$ and $j$ are on the right, so we can apply the argument above.
Similarly in case (iii) the split-up happened at some earlier time with $k$ on the left and $i$ and $j$ on the right, so again we can proceed as in case (i) above.

Above we assumed that $(k, i)$ and $(k, j)$ were not fill edges. Now if either, or both, of them are fill edges, then we find the vertices that caused them. Say, if $(k, i)$ is a fill edge, then there must be a vertex $l$ ($l < i < j$), such that the elimination of $l$ caused $(k, i)$ to appear. Now if either (or both) $(l, k)$ or $(l, i)$ are fill edges we still go back and find the vertices that caused them. Eventually we will find a path of original graph edges between $i$ and $j$ and all of the vertices are lower numbered than $i$ and $j$.

Now we can argue similarly to above when the path only consisted of vertex $k$. If we split the path up so that some part of it is on the left and $i$ and $j$ are on the right, as shown in Figure 2.9, then we will put an edge from the lowest numbered vertex $x$ on the path that is in the right half to the other vertex $y$ on the path that had an edge across the split. Note that the path can only cross the
Figure 2.9: A path of original edges between $i$ and $j$

split twice, because of the way it was constructed. If $x = i$ then $y$ must be the same as $j$ and we have put an edge between $i$ and $j$ and the algorithm will go on to set $j$ as the parent of $i$. If $x \neq i$ then we have a new path, shorter than the one we had before and again we will split it, and so on. Eventually either the lowest numbered vertex on the left will be $i$ or we will split it up so that the path and the vertex $i$ will be on the left and $j$ on the right. In that case we have the same situation as above when the path consisted only of $k$. The vertex $i$ will become the root of an elimination tree on the left and it will get hooked onto $j$, which is the lowest numbered vertex on the right adjacent to $i$'s component.

We have thus managed to show that for every parent-child relationship between two vertices in $H_R$ in the correct elimination forest, that relationship also holds in $F_R$. Therefore $F_R$ is an induced subgraph of the correct elimination forest for $T$. The same thing can be shown for $F_L$ using the same argument as above. □

**Theorem 2.6** Let $T^{ALG}$ be the tree generated by the algorithm for the graph $G$ and let $T$ be the correct elimination tree for $G$. Then $(i, j) \in T^{ALG}$ if and only if
\[(i, j) \in T.\]

**Proof:** We use induction on the levels in the algorithm.

*Base case:* If there is only one vertex then the theorem holds trivially, since there is no edge in \(T\) and the algorithm will not produce any edge.

*Ind. step:* We assume that \(F_L\) and \(F_R\) are the correct elimination forests for \(H_L\) and \(H'_R\) respectively and we want to show that the algorithm merges them into the correct elimination forest \(T_H\) for \(H\). By Lemma 2.5 we know that \(F_L\) and \(F_R\) are induced subgraphs of \(T_H\). Hence, whenever vertex \(i\) has a parent in \(F_L\) or \(F_R\) then it has the same parent in \(T_H\). But if \(i\) does not have a parent in \(F_L\) or \(F_R\) and has one in \(T_H\) then \(i\) must be in \(H_L\) and its parent in \(H'_R\). Thus we need to consider the vertices in \(H_L\) that do not have any parents in \(F_L\).

Let \(i \in H_L\) have no parent in \(F_L\). Then the algorithm will set as the parent of \(i\) the lowest numbered vertex \(j\) in \(H'_R\) that has an edge to some vertex \(h\) in \(i\)'s elimination tree in \(F_L\). Edge \((j, h)\) causes \(l_{ji}\) to be nonzero. Furthermore this \(j \in H'_R\) that the algorithm picked is the smallest vertex greater than \(i\) with \(l_{ji} \neq 0\), because it is the smallest one in \(H'_R\) and there is no vertex in \(H_L\) that satisfies these conditions, or else \(i\) would have had a parent in \(H_L\). Thus the \(j\) picked by the algorithm is the one the definition of the elimination tree says is \(i\)'s parent. Thus, if \((i, j) \in T^{ALG}\) then \((i, j) \in T.\]

Let \((i, j) \in T_H\) with \(i \in H_L\) and \(j \in H_R\). Then by definition \(j\) is the smallest vertex in \(H_R\) with \(l_{ji} \neq 0\). This is the same as saying that \(j\) is the smallest vertex in \(H_R\) that has an edge to \(i\)'s elimination tree. But that is just the \(j\) that the algorithm selects.
Hence the elimination tree $T^{ALG}$ generated by the algorithm in Figure 2.3 is the same as the correct tree $T$. $\square$
Chapter 3

Symbolic Factorization

The purpose of symbolic factorization is to find the nonzero structure of the Cholesky factor, \( L \), under the assumption that no coincidental numeric cancellation takes place during the factorization. In the case of sequential Cholesky factorization it is important to know the nonzero structure to be able to allocate space for the numeric factorization phase. We are doing the numeric factorization in parallel and we allocate a processor to each nonzero of \( L \). Thus, we also use the nonzero structure to assign processors.

If we did not do the symbolic factorization, the numeric factorization phase would still have to determine where the fill occurs, so we would just be doing the symbolic factorization implicitly and the resulting numeric algorithm would be more complicated.

If we need to factor many matrices that have the same nonzero structure but have different values in their nonzero locations, we have to do a numeric factoriza-
tion for each matrix, but only one symbolic factorization is necessary.

In this chapter we present a parallel algorithm to compute the symbolic factorization of a sparse matrix that runs in \(O(\log^2 n)\) time using \(m^*\) processors, where \(m^*\) is the number of nonzeros in the Cholesky factor \(L\).

### 3.1 Finding the Fill

Given the elimination tree computed in Chapter 2, we want to use it to find the fill that occurs in the graph (i.e., the matrix). Some of the lemmas from Chapter 2 will be useful.

We can use Lemma 2.3 to find all the fill that will occur. For each nontree edge \((i, j)\), with \(i < j\), we add a fill edge \((k, j)\) for all vertices \(k\) between \(i\) and \(j\). However, there are some complications. First, we want to do this in parallel. In addition, since a single fill edge can result from many different nontree edges, we want to avoid redundant additions of fill edges.

Let us consider a single nontree edge \((i, j)\), with \(i < j\), and assume that the length of the path between \(i\) and \(j\) is an integer power of 2. Then we can use a path-halving method with \(\log n\) stages to add the fill edges along the path. We assume that we have a processor \(P_i\) associated with vertex \(i\). Processor \(P_i\) finds the vertex \(k\) that is halfway between \(i\) and \(j\) on the tree path and adds the fill edge \((k, j)\). Then \(P_i\) spawns another processor \(P_k\) and associates it with vertex \(k\). Processor \(P_k\) now goes on to recursively cover the path between \(k\) and \(j\), while \(P_i\) covers the path from \(i\) to \(k\).
If the path between $i$ and $j$ does not have length that is an integer power of 2 then we can split the path up into pieces whose lengths are the largest integer powers of 2 possible. For instance, if the path length is 11 then we split it up into pieces of lengths 8, 2, and 1 (i.e., $2^3$, $2^1$, and $2^0$). The longest piece will be between a vertex $p_1$ and $j$, the second longest between a vertex $p_2$ and $p_1$, and so on down to $i$. We add the fill edges $(p_1, j)$, $(p_2, j)$, \ldots, assign processors to each of the vertices $i$, $p_1$, $p_2$, \ldots, and then proceed as above.

What happens when we have more than one nontree edge? If they all have different ancestor vertices (i.e., the higher numbered vertex) then all the fill edges produced will be distinct. However, if two or more nontree edges have the same ancestor vertex then they might try to add the same fill edge. We do not mind two processors adding the same edge, except that it increases the number of processors needed. In an extreme case we could require $O(n^2)$ processors, when $m^* = O(n)$. Therefore, we need to make sure that processors are not duplicating each others work.

In Figure 3.1 we have two nontree edges with the same ancestor vertex. The processors associated with $i$ and $l$, $P_i$ and $P_l$, will both try to add the fill edge $(k_2, j)$. The one that succeeds will spawn a processor to cover the upper range, the other one will not. Regardless of which one succeeds, $P_i$ and $P_l$ will still recursively cover their lower ranges. Thus, $P_l$ will add the fill edge $(k_4, j)$ and $P_i$ will add $(k_1, j)$.

We will show later that whenever a processor $P_i$ tries to add a fill edge that has already been added, it need not spawn a processor to cover the upper range;
Figure 3.1: Two nontree edges with the same ancestor vertex

it will be covered. However, $P_1$ will still have to cover the lower range.

The algorithm consists of two stages. In the first stage we start by assigning a
processor $P_1$ to each nontree edge $(i,j)$. If the distance between $i$ and $j$ in the tree
is an integer power of 2 then we are done with the first stage for this particular
edge. If not, $P_1$ adds the fill edge $(p_1,j)$, spawns a processor and puts it on the
vertex $p_1$, where $p_1$ is the vertex on the path from $i$ to $j$, whose distance from $j$ is
the largest integer power of 2 less than the distance between $i$ and $j$. If the distance
between $i$ and $p_1$ is an integer power of 2 then this is the end of the first stage for
this edge. Otherwise, $P_1$ adds the fill edge $(p_2,j)$ and puts a new processor on the
vertex $p_2$ between $i$ and $p_1$, whose distance from $p_1$ is the largest integer power of
2 less than the distance between $i$ and $p_1$. We repeat this until we encounter a
path whose length is an integer power of 2. Thus, the distance between each of $j$,
proc StageOne(i, j : vertex)
    k := j
    d := lev(i) − lev(j)
    l := 2^{\lfloor \log d \rfloor}
    while d ≠ l do
        p := binsearch(K(lev(k) + l), preorder(i))
        spawnproc(j, p, k)
        k := p
        d := d − l
        l := 2^{\lfloor \log d \rfloor}
    od
end

Figure 3.2: Stage 1 of the symbolic factorization algorithm

p₁, p₂, \ldots, i is a strictly decreasing integer power of 2. Each of the processors on these vertices will, in the second stage, be responsible for covering the range from the vertex it is on through the vertex that the next processor is on. For example, the processor on p₂ will attempt to add a fill edge (k, j) for each vertex k on the path from p₂ to p₁ (excluding p₁).

Let us now look more carefully at the implementation of stage one. Before we start stage one we preprocess the elimination tree. We find a preorder numbering of the nodes. In addition we find the level number (i.e. depth) of each node and set up an array K of lists of vertices. Entry K(l) contains all the vertices on level l of the tree, ordered by preorder number. This preprocessing takes time O(log n) using n processors [TV84].

The algorithm in Figure 3.2 does the first stage for a nontree edge (i, j). Since
we know the level numbers of \( i \) and \( j \), we can calculate the level number of the vertex \( p_1 \). We then use binary search on the array \( K(\text{lev}(p_1)) \) to find the actual vertex \( p_1 \). It is the vertex in \( K(\text{lev}(p_1)) \) whose preorder number is closest to preorder\((i)\) from above. This will take at most \( O(\log n) \) time for each vertex \( p_k \).

The routine \( \text{spawnproc}(j, p, k) \) spawns a processor, associates it with vertex \( p \), and informs it that it has to add fill edges to \( j \) on the tree path from \( p \) to \( k \). The maximum number of processors that \( P_i \) spawns is \( \log n \). Thus the total time for stage one is \( O(\log^2 n) \).

Stage two starts with all the processors spawned in \( \text{StageOne} \) using \( \text{spawnproc} \) and all the processors \( P_i \), for each nontree edge \((i, j)\), simultaneously executing the statement

\[
\text{StageTwo}(j, p, k),
\]

where \( p \) is the lower vertex of the range that the processor has to cover and \( k \) is the upper one.

The algorithm for the second stage is in Figure 3.3. Invoking \( \text{StageTwo}(j, i, k) \) covers the path from the vertex \( i \) to \( k \), adding edges \((v, j)\) for all vertices \( v \) on the path between \( i \) and \( k \). The algorithm starts by finding the vertex \( p \) that is halfway between \( i \) and \( k \) and attempts to add a fill edge between it and \( j \). If the processor succeeds in adding the fill edge then another processor is spawned and the two of them recur in parallel on the paths from \( i \) to \( p \) and from \( p \) to \( k \) respectively. A processor fails in adding the fill edge if two processors try to enter the same fill edge at the same time and then an arbitrary one succeeds. That is a
\textbf{proc} StageTwo\((j, i, k: \text{vertex})\) \\
\textbf{if} \((i \neq k)\) \textbf{then} \\
\hspace{1em} l := \lfloor(\text{lev}(i) + \text{lev}(k))/2\rfloor \\
\hspace{1em} p := \text{binsearch}(K(l), \text{preorder}(i)) \\
\hspace{1em} \text{addfilledge}(p, j) \\
\hspace{1em} \textbf{if} \ (\text{success}) \ \textbf{then} \\
\hspace{2em} \textbf{pardo} \\
\hspace{3em} \text{StageTwo}(j, i, p) \\
\hspace{3em} \text{StageTwo}(j, p, k) \\
\hspace{1em} \textbf{od} \\
\textbf{else} \\
\hspace{1em} \text{StageTwo}(j, i, p) \\
\hspace{1em} \textbf{fi} \\
\textbf{fi} \\
\textbf{end}

Figure 3.3: Stage 2 of the symbolic factorization algorithm

property of our model of computation, the CRCW PRAM. Also, if there is already a fill edge there, added in \textit{StageOne} or earlier in \textit{StageTwo}, then the processor does not succeed. In these cases the processor will not spawn another processor, but will continue recurring on the path between \(i\) and \(p\). However, the processor does register success if there is already an original graph edge there or a fill edge added during the computation of the elimination tree. It then marks that edge as an added fill edge.

If two processors try to add the same fill edge at the same time then we will prove in Section 3.4 that they have the same upper range. However, their lower range might not be the same. The same thing applies if a processor tries to add a
fill edge that is already there.

The time that stage two takes is no more than $O(\log n)$ for each level of recursion and there are no more than $\log n$ levels. This gives us $O(\log^2 n)$ for the whole stage. A new processor is spawned only when an old processor successfully adds a fill edge, so the number of processors is the same as the number of fill edges plus the processors used for the preprocessing, a total of $m^* + n$. Since $m^* \geq n - 1$, we can use just $m^*$ processors and increase the time required by a constant factor.

### 3.2 Restricting the Memory Size

We can reduce the amount of memory used in the symbolic factorization as we did in finding the elimination tree. If we have $O(n^2)$ space then we can add a new edge or test for an edge in constant time. The amount of space can be restricted to $O(m^*)$ for finding the filled graph by keeping a list of the edges of the graph $G^*$ in memory, with a processor assigned to each edge.

The only tricky part is adding edges to the graph. Each time some edges are added, we will sort the list of old and new edges to remove duplicates and reassign processors to edges. The sort takes $O(\log n)$ time with $m$ processors using Cole’s parallel merge sort [Col86].

Edges are added to the graph in both StageOne and StageTwo during the symbolic factorization. In StageOne an edge might be added in each of its $O(\log n)$ rounds. This is also the case in StageTwo. After each round, we sort the edges (new and old) to remove duplicates. In addition StageTwo requires that each pro-
cessor that tries to add an edge must find out whether it was successful. This can be done by associating with each fill edge the name of the processor that added it. After the sort, but before eliminating duplicates, all but the lowest numbered of the processors that tried to add a given edge can be told that they failed.

In StageOne and StageTwo we sort edges $O(\log n)$ times. Therefore, the total time spent sorting is $O(\log^2 n)$, and the entire algorithm still takes $O(\log^2 n)$.

### 3.3 An Example

We now look at an example of how the above algorithm adds fill edges. We will be using the example from Figure 2.1.

The elimination tree has four nontree edges: $(1, 5), (1, 7), (2, 8)$, and $(3, 8)$. We allocate a processor to each of these edges and they start by executing StageOne. The distance between vertex 1 and vertex 5 in the tree is 2, so edge $(1, 5)$ does not result in any fill or new processors in the first stage. The same thing applies to edge $(2, 8)$, where the distance is 4 ($= 2^2$). For edges $(1, 7)$ and $(3, 8)$, however, the distance is 3. Thus, for these edges, in StageOne we will add the fill edges $(3, 7)$ and $(5, 8)$, and assign new processors to vertices 3 and 5. Actually edge $(5, 8)$ was added during the computation of the elimination tree, so we just mark it as a fill edge.

Now, we are ready to execute StageTwo. The current situation is shown in Figure 3.4. The original nontree edges are dashed lines and the fill edges added before StageTwo are dotted lines. The fill edge $(5, 7)$, which was added while
computing the elimination tree, is not shown. We now execute the following in parallel:

\[ \text{StageTwo}(5, 1, 5) \]
\[ \text{StageTwo}(8, 2, 8) \]
\[ \text{StageTwo}(7, 1, 3) \]
\[ \text{StageTwo}(7, 3, 7) \]
\[ \text{StageTwo}(8, 3, 5) \]
\[ \text{StageTwo}(8, 5, 8) \]

The parallel executions \text{StageTwo}(8, 2, 8) and \text{StageTwo}(8, 5, 8) are the only ones interfering with each other. Both of them attempt to cover the path from vertex 5 to vertex 8. The processor associated with (5, 8), \( P_{(5,8)} \), first tries to add the fill edge (7, 8). At the same time, \( P_{(2,8)} \) attempts to add the edge (5, 8), since the vertex 5 is halfway between 2 and 8 in the tree. The processor \( P_{(5,8)} \) registers success even though there is an original edge (7, 8). It marks the edge as added.
and recurs in parallel on the two halves, where it encounters the base case of the recursion and quits. The processor \(P_{2,8}\) does not succeed, since the fill edge \((5, 8)\) was added in \textit{StageOne}. It then goes on to cover the path from 2 to 5 and adds the fill edge \((4, 8)\).

The other parallel executions of \textit{StageTwo} follow a similar pattern, except that they never encounter a fill edge added by another processor also executing \textit{StageTwo}. The execution of \textit{StageTwo}(5, 1, 5) adds the fill edge \((3, 5)\) and \textit{StageTwo}(7, 3, 7) adds the fill edges \((3, 7)\) and \((5, 7)\).

### 3.4 Proof of Correctness

We will prove the correctness of the algorithms in Figures 3.2 and 3.3 in two parts.

**Theorem 3.1** If the two part symbolic factorization algorithm adds the edge \((v, j)\) then \((v, j)\) is a fill edge.

**Proof:** The edge \((v, j)\) is a fill edge if and only if \(v\) is a descendant of \(j\) in the elimination tree \(T\) and there is a descendant \(i\) of \(v\) with an edge to \(j\). If \textit{StageOne} adds the edge \((v, j)\) then the result is obvious.

Similarly if \((v, j)\) is added during some recursive call to \textit{StageTwo}, then there must have been an initial invocation of the form \textit{StageTwo}(\(j, i, k\)), where \(v\) is on the path between \(i\) and \(k\) in \(T\). Thus, vertices \(j\) and \(i\) have an edge between them and vertex \(i\) is a descendant of \(v\). \(\square\)

**Lemma 3.2** Let \textit{StageTwo}(\(j, i, k\)) and \textit{StageTwo}(\(j', i', k'\)) be two parallel executions and \(P\) and \(P'\) respectively be the processors executing them.
(i) If $P$ and $P'$ try to add the edge $(v, j)$ at the same time then $k = k'$ and $\text{level}(i) = \text{level}(i')$.

(ii) If $P$ tries to add the edge $(v, j)$ but does not succeed because $P'$ had already added $(v, j)$ then $k = k'$ and $\text{level}(i) = \text{level}(i')$.

(iii) If $P$ tries to add the edge $(v, j)$ but does not succeed because $(v, j)$ was added in StageOne then $v$ equals some $p_l$ and $k$ equals $p_{l-1}$, where $p_l$ and $p_{l-1}$ were defined in StageOne.

**Proof:** As a result of StageOne, whenever a processor starts StageTwo the size of its range is an integer power of 2. In StageTwo it splits its range into two equal halves, so their size will also be an integer power of 2. Thus, when a processor tries to add an edge $(v, j)$, $v$ is the midpoint of a range whose size is an integer power of 2. We want to prove that for all the processors trying to add $(v, j)$, this size is the same.

Let us consider an invocation of StageTwo that will eventually result in an attempt to add the edge $(v, j)$. That initial invocation will try to cover the path from vertex $p_k$ to vertex $p_{k-1}$, where $p_k$ and $p_{k-1}$ are from StageOne. Then $v$ is somewhere on the path between $p_k$ and $p_{k-1}$. The distance between $p_k$ and $j$ is an integer multiple of $2^l$ (for some integer $l$), but not of $2^{l+1}$ (i.e., $\text{dist}(p_k, j) = c2^l$, with $c$ an odd number). The distance from $p_k$ to $p_{k-1}$, as defined in StageOne, is the largest integer power of 2 less than the distance from $i$ to $p_{k-1}$, where $(i, j)$ is the original nontree edge. In StageOne we assigned the vertices $p_{k-1}, p_{k-2}, \ldots, p_1$, so that their distance from $p_{k-2}, p_{k-3}, \ldots, j$ respectively would be the largest integer
power of 2 less than the distance from $i$ to $p_{k-2}, p_{k-3}, \ldots, j$ respectively. Thus, the distance from $p_{k-1}$ to $j$ must be an integer multiple of $2^q$, where $q \geq l + 1$. However, the distance from $p_k$ to $j$ is an integer multiple of $2^l$ but not of $2^{l+1}$, so the distance between $p_k$ and $p_{k-1}$ must be $2^l$. Then the range between $p_k$ and $p_{k-1}$ is split up in two equal halves each of size $2^{l-1}$, so the distance between the midpoint of that range and $j$ has to be an integer multiple of $2^{l-1}$, but not of $2^l$.

This splitting up of ranges continues as we go down the recursion of StageTwo and the midpoint is always an integer multiple of some $2^h$, but not of $2^{h+1}$. The midpoint is divisible by lower and lower integer powers of 2 as we go down the recursion. At the same time the size of the range is $2^{h+1}$ and decreasing by a factor of 2 at each level.

Finally there will be a processor splitting up its range and the vertex $v$ will be the midpoint of that range. The distance from $v$ to $j$ will be an integer multiple of $2^{h'}$ (for some integer $h'$), but not of $2^{h'+1}$. The size of the range will be $2^{h'+1}$.

Thus, we have shown that whenever a processor tries to add the edge $(v, j)$, its range is of size $2^{h'+1}$, when $\text{dist}(v, j) = c2^{h'}$, with $c$ an odd number. Also, $v$ is the midpoint of the range.

If $P$ and $P'$ are processors executing StageTwo$(j, i, k)$ and StageTwo$(j, i', k')$ respectively and they try to add the edge $(v, j)$ at the same time then their ranges are the same size and $v$ in the midpoint of those ranges. Hence $k = k'$ and level$(i) = \text{level}(i')$. Similarly, if $P$ tries to add the edge $(v, j)$ that $P'$ has already added, their ranges are the same size and $k = k'$ and level$(i) = \text{level}(i')$. If $P$ tries to add $(v, j)$ but does not succeed because $(v, j)$ was added before stage two then $v = p_l$,
for some integer \( l \). Then the upper range of \( P \) is the same as the range from \( p_l \) to \( p_{l-1} \), so \( k = p_{l-1} \). \( \Box \)

**Theorem 3.3** If \((v, j)\) is a fill edge, then the algorithm will add \((v, j)\).

**Proof:** Since \((v, j)\) is a fill edge there must be some descendant \( i \) of \( v \) in the tree that has an edge to \( j \). Procedure \textit{StageOne} will therefore be invoked on that edge. Assume that in \textit{StageOne} \( v \) is found to be on the path between vertices \( p_k \) and \( p_{k-1} \). Thus, procedure \textit{StageTwo} gets called with parameters \( j, p_k \), and \( p_{k-1} \).

Each call \textit{StageTwo}(\( j, i, k \)) attempts to cover the range from the lower bound \( i \) to the upper bound \( k \). It splits the range into two equal parts, tries to add fill edge \((p, j)\), where \( p \) is the midpoint, and if successful makes a recursive call on the upper and lower half. If it does not succeed in adding the edge then it only goes on to cover the lower half of the range and we claim that the upper half will be covered by the processor that did add the fill edge \((p, j)\).

There are two reasons why a processor \( P \) may not succeed in adding the fill edge. The first is that another processor \( P' \) added the edge at the same time. By Lemma 3.2, whenever two processors try to add the same edge at the same time their upper ranges are the same. Since \( P' \) succeeded in adding the edge it will go on to cover both its upper and lower range, but its upper range is also from \( p \) through \( k \), so that range will be covered. The second reason for \( P \) not succeeding in adding the fill edge \((p, j)\) is if a processor \( P'' \) had already added the edge, either in a call to \textit{StageTwo} or in \textit{StageOne}. The upper range of \( P'' \) is from \( p \) through \( k \) by Lemma 3.2, so \( P'' \) will cover that range. In either case the range from \( p \) through
$k$ will be covered. If $P'$ or $P''$ ever get to a point where they do not succeed in adding a fill edge we can apply the above argument again on them.

Thus, since the procedure $StageTwo$ will cover the whole range from $p_k$ through $p_{k-1}$, it will add the fill edge $(v, j)$. □
Chapter 4

Numeric Factorization

The numeric factorization is the principal component of a sparse Cholesky factorization algorithm. In the previous stages we will have computed things that simplify or speed up the numeric part, i.e., a new ordering and the fill that will occur. However, the numeric part is where we actually compute the values of the Cholesky factor.

The best sequential algorithm computes the Cholesky factorization of a sparse matrix $A$ in time proportional to the number of operations required to multiply $L$ and $L^T$ [GL81].

Previous work on parallel sparse Cholesky factorization has concentrated on algorithms in which the number of processors is much smaller than the number of nonzeros in the matrix. The work of George et al. [GHLN88,GHLN86] is mostly of experimental nature, and there is no worst case analysis available. Duff [Duf86] presents a parallel multifrontal approach to factorization. Zmijewski’s
algorithm [Zmi87] is for message-passing multiprocessors and it is geared towards low message overhead.

In this chapter we will show how the numeric Cholesky factorization can be done in parallel using the elimination tree. The main version of the algorithm takes $O(h \log n)$ time using $m^*$ processors. We will then show how the algorithm can be made more efficient for certain types of matrices, and finally show how to use very fast but unstable algorithms for dense matrices to get a very fast algorithm for sparse matrices.

4.1 Parallel Cholesky Factorization

In the sequential Cholesky factorization algorithm in Figure 1.2 of Section 1.3 there are essentially two kinds of parallelism available. We can exploit the parallelism within the column operations, $cmod$ and $cdiv$, and we can also try to compute the columns simultaneously.

In Liu's medium grained task model [Liu86b] each task is either a $cmod$ or a $cdiv$ operation. A fine grained model [WH80] associates a task with a single multiplicative operation in the factorization process. Our algorithm is somewhere between these two models. We do the $cmod$ and $cdiv$ operations in parallel and we also exploit the parallelism within each of those operations.

In computing column $j$ of $L$, column $j$ of $A$ has to be modified by all the lower numbered columns of $L$ that have nonzeros in row $j$ of $L$. If $L$ is dense, this means that we cannot finish computing column $i$ of $L$ until we have computed
columns 1 through \( i - 1 \). However, if \( L \) is sparse there might be a number of columns, whose corresponding rows do not have any off-diagonal nonzeros. Thus, they do not have to be modified by any other columns and we can compute those columns simultaneously. The following lemma from Liu's survey of elimination trees [Liu87b] provides insight into the relationship between the elimination tree and column dependence.

**Lemma 4.1** Vertex \( j \) is a leaf in the elimination tree \( T \) if and only if there are no off-diagonal nonzeros in row \( j \) of the Cholesky factor \( L \).

To see why the lemma is true note that if there is a nonzero in location \((j,i)\) of \( L \), then, according to Theorem 2.3, vertex \( i \) is a descendant of \( j \) in \( T \). Thus \( j \) is not a leaf.

Also note that if there are no off-diagonal nonzeros in row \( j \) of \( L \) then vertex \( j \) can not be the parent of any other vertex in \( T \). For \( j \) to be the parent of a vertex \( i \), the first nonzero below the diagonal in column \( i \) of \( L \) has to be in row \( j \). But there are no nonzeros in row \( j \) (except for the diagonal), so vertex \( j \) must be a leaf.

The lemma tells us that all the columns that correspond to leaves in the elimination tree can be computed immediately. We also see that a column corresponding to vertex \( j \) in the tree only has to be modified by the columns corresponding to those of \( j \)'s descendants in the tree that have an edge in \( G^* \) to \( j \). Thus, if \( k \) is in another branch of the elimination tree than \( j \) then the computation of column \( j \) is completely independent of the computation of column \( k \).

Our algorithm for the numeric Cholesky factorization computes columns cor-
**Algorithm** parallel-Cholesky

**begin**

unmark every vertex of $T$

repeat

for each u-leaf $j$ of $T$ pardo

cdiv($j$)

mark $j$

od

for each u-leaf $j$ of $T$ pardo

for each edge $(j, k)$ of $G^*$ with $k < j$ pardo

cmod($j, k$)

od

od

until every vertex is marked

end

---

Figure 4.1: Parallel Cholesky factorization

responding to the leaves of the elimination tree upwards, marking vertices as their columns are computed. In the algorithm of Figure 4.1, a "u-leaf" is an unmarked vertex whose children are all marked; it is a leaf of the unmarked part of $T$. It is easy to see that Lemma 4.1 applies to the unmarked portions of $T$ and $L$.

In the algorithm one processor will be allocated to each nonzero element of $L$, so we need $m^*$ processors. All the processors have access to the elimination tree $T$, which is in global memory. Thus, each processor knows which vertices are u-leaves and can determine if it should be involved in some operation.

First we need to do $cdiv(j)$ for each u-leaf $j$. For each nonzero $(i, j)$ of $L$, the assigned processor $P_{ij}$ knows if $j$ is a u-leaf in $T$ and if so it divides the value at
Figure 4.2: Column \( j \) has to be modified by a multiple of column \( i \)
position \((i, j)\) by the square root of \( a_{ij} \). This can all be done in \( O(1) \) time. Then all the u-leaves of \( T \) are marked, giving \( T \) a new set of u-leaves.

Next, we need to do \( cmod \) for all the new u-leaves. First we have to find out, for each leaf, which columns it needs to be modified by. That can be done in constant time. Each processor \( P_{ji} \) determines if column \( j \) is a u-leaf in \( T \). If it is then column \( j \) has to be modified by a multiple of column \( i \) (see Figure 4.2). Sequentially, this can be done by subtracting a multiple of column \( i \) from column \( j \), one nonzero at a time. In parallel it is better to first sum up all the columns to be subtracted and then subtract the sum from column \( j \). Thus, the inner loop in the algorithm of Figure 4.1 is not done by performing each \( cmod(j, i) \) seperately in parallel, but rather by summing up all the modifications in parallel and subtracting the result. Hence, for each column \( i \), where \( L_{ji} \) is nonzero, we begin by multiplying each nonzero in the column by \( L_{ji} \). We then sum the resulting row vectors, using a \( O(\log n) \) time parallel vector sum algorithm, and subtract each result from its corresponding element in column \( j \).

However, if we only want to use a total of \( m^* \) processors in the algorithm, we
have to be careful. In general, the rows to be summed are sparse. Thus, when we are summing up a row we only have as many processors as there are nonzeros in that row. Therefore, to be able to use the parallel vector sum algorithm we have to make sure that each processor knows which other processors it is supposed to communicate with to compute the sum.

We assume that we have an array $AUX$ of size $m^*$ that stores column number and value for each nonzero in $L$, in row major order. We also assume we have an index into $AUX$ providing the starting location of each row. This data structure can be initialized in this way any time after we have computed the fill. We then assign the processor $P_{ji}$ to the position $(j, i)$, which will become nonzero in $L$. Each $P_{ji}$ records $j$ and a pointer to the position in $AUX$ that contains its nonzero value. This is enough information for $P_{ji}$ to determine $i$, and also for it to find out how many nonzeros are before $L_{ji}$ in row $j$.

Now, when $j$ becomes a u-leaf in $T$, we need to do $cmod(j, k_1), cmod(j, k_2), \ldots, cmod(j, k_q)$ for each of the nonzeros $L_{j_1}, L_{j_2}, \ldots, L_{j_q}$ in row $j$. Suppose row $i$ will participate in this $cmod$ and suppose that $L_{ik_l}$ is nonzero, for some $1 \leq l \leq q$ (see Figure 4.3). Then processor $P_{ik_l}$ gets the value of $L_{jk_l}$ from $AUX$, and also finds out that $L_{jk_l}$ is the second nonzero in row $j$. Therefore, $P_{ik_l}$ knows that its contribution to the update of $L_{ij}$ is the second value in the sum to be subtracted from $L_{ij}$. Now the processors involved in the update add their contributions using a parallel vector sum algorithm. This takes no more than $O(\log n)$ time and the only processors we use are the ones associated with nonzeros in row $i$.

Each column is only modified by the columns that are below it in the elimination
Figure 4.3: Row $i$ is summed and the result subtracted from $(i,j)$

tree. Hence, as the algorithm always modifies the columns at the current u-leaves of the elimination tree and u-leaves cannot be ancestors or descendants of one another, no two columns are modified by the same column at the same time. Thus, each nonzero is involved in at most one $cmod$ at a time.

The total amount of space used for the above computation is $O(m^*)$. The time required to do a $cmod$ is $O(\log n)$ and we only use the processors associated with the nonzero elements involved, so the total number of processors used is $m^*$. To determine the total time for the whole algorithm, we observe that we mark all the u-leaves each time through the repeat-until loop and thus we decrease the height of the unmarked elimination tree by one each time. We need $O(\log n)$ time for each time through the loop. Hence, the total time is $O(h \log n)$, where $h$ is the height of the original elimination tree. The number of processors is $m^*$, since we only use the processors associated with the nonzero elements of $L$. 
4.1.1 An Example of Parallel Cholesky Factorization

We will again take as an example the matrix from Chapter 2, here shown with its elimination tree in Figure 4.4 (a). The elimination tree has the three leaves 1, 2, and 6. Therefore, we start by doing \texttt{cdiv(1)}, \texttt{cdiv(2)}, and \texttt{cdiv(6)}, as well as marking vertices 1, 2, and 6. Figure 4.4 (b) shows that the unmarked part of \( T \) consists of vertices 3, 4, 5, 7, and 8. Those vertices correspond to the columns of \( L \) still to be computed.

We now have the u-leaves 3 and 4 and their corresponding columns need to be modified by columns 1 and 2 respectively. Thus we do \texttt{cmod(3, 1)} and \texttt{cmod(4, 2)}. Then we can do \texttt{cdiv(3)} and \texttt{cdiv(4)} after which we mark vertices 3 and 4, giving Figure 4.4 (c).

Now vertex 5 is the only u-leaf. Its column has to be modified by columns 1, 3, and 4. Then we do \texttt{cdiv(5)} and mark vertex 5, leaving us with a "u-tree" consisting of vertices 7 and 8. Column 7 gets modified by columns 1 and 5 and then we do \texttt{cdiv(7)} and mark vertex 7. The last steps involve modifying column 8 by columns 2, 3, and 7, and finally doing \texttt{cdiv(8)}.

4.2 Saving Processors

The algorithm presented above assumes that we have a processor associated with each nonzero of the Cholesky factor \( L \). During the course of the algorithm we might not need to have all of those processors active at all times. For some matrices and associated orderings we can take advantage of this fact and can make do with fewer
Figure 4.4: An example of parallel Cholesky factorization
processors. In this section we consider a particular class of matrices and we show
how the number of processors can be reduced from $m^* = n \log n$ to $n$.

Let $G(A)$ be a two-dimensional finite element graph. This bounded degree
graph can be obtained from a planar graph by inscribing a clique in each face. Applying a variant of nested dissection called “wide quotient tree ordering” [ZG85] we can get simultaneously a Cholesky factor with $O(n \log n)$ nonzeros and an elimination tree of height $O(\sqrt{n})$, both of which are best in general.

The shape of the elimination tree for the two-dimensional finite element graph
resulting from the wide quotient tree ordering is shown in Figure 4.5. It has
$O(\log n)$ levels of graph separators. The separator $a$ on the first level has $\approx \sqrt{n}$
vertices, the separators $b$ and $c$ on the second have $\approx \sqrt{\frac{n}{2}}$ vertices, the ones on the
third level have $\approx \sqrt{\frac{n}{4}}$ vertices, and so on. Another property of this elimination
tree is that the edges of $G^*$ are only between vertices within separators or between
vertices on adjacent levels of separators. For instance, there are no nontree edges
between a leaf and the root (unless the number of vertices in the tree is so small
that the leaf and the root are on adjacent separator levels).

We now divide the $h \log n$ steps of the original parallel factorization algorithm
into $O(\log n)$ phases corresponding to the levels of separators in the elimination
tree. Each phase uses only $n$ processors. By reallocating the processors at the
end of each phase we can compute the factorization in $O(\sqrt{n} \log n)$ time with $n$
processors, instead of $n \log n$ processors. We now describe this procedure in more
detail.

The last phase of the algorithm computes the $\approx \sqrt{n}$ columns that correspond to
Figure 4.5: An elimination tree for a two-dimensional finite element graph

the vertices of the first graph separator. In this last phase we allocate processors to the nonzeros that correspond to edges within the first graph separator. Since the size of the separator is $\approx \sqrt{n}$ there can be no more than $O(n)$ nonzeros. We also allocate processors to the nonzeros that correspond to the nontree edges between the first graph separator and the next ones. There are two such separators, each with $\approx \sqrt{n/2}$ vertices. Thus the maximum number of edges between the two levels is $2 \times \sqrt{n} \times \sqrt{n/2} = \sqrt{2} n$, which is $O(n)$. Then when we perform the cmods and the cdius in this last phase they only affect nonzeros that have associated processors.

The second to last phase of the algorithm deals with the two $\sqrt{n/2}$ separators. There can be no more than $2 \times (\sqrt{n/2})^2 = n$ edges within those separators. We also need processors on the edges both between this level and the previous one and between this level and the next one. We already calculated that there are at most $\sqrt{2} n$ edges between this level and the previous one. The next level has four separators each with $\sqrt{n/4}$ vertices. Thus the number of edges between that
level and the current one is no more than \(4 \times \sqrt{n/2} \times \sqrt{n/4} = \sqrt{2} n\). Therefore the total number of processors used in this phase is \(O(n)\). Note that all the nonzeros in the columns we are computing in this phase have processors associated with them. This ensures that the \(cmods\) and the \(cdivs\) only operate on nonzeros that have processors associated with them.

In a general phase of the algorithm we allocate processors to three types of edges: edges within the separators, edges to previous level, and edges to next level. If we are on level \(i\) in the elimination tree then there are \(2^i\) separators and each has \(\approx \sqrt{n/2^i}\) vertices. The number of edges within in the separators is no more than

\[
2^i (\sqrt{n/2^i})^2 = n.
\]

The number of edges to the previous level is at most

\[
2^i \times \sqrt{n/2^{i-1}} \times \sqrt{n/2^i} = \sqrt{2} n.
\]

The number of edges to the next level is also no more than

\[
2^{i+1} \times \sqrt{n/2^i} \times \sqrt{n/2^{i+1}} = \sqrt{2} n.
\]

Thus the total number of processors needed is \(O(n)\).

Hence, we can save a factor of \(\log n\) in the number of processors needed to factor a matrix, whose graph is the two-dimensional finite elimination graph ordered by the wide quotient tree order.

The above method can also be applied to bounded degree planar graphs ordered by wide quotient tree order [ZG85] or to any other bounded degree graph satisfying an \(n^\alpha\) separator theorem.
4.3 Pipelining

In the parallel factorization algorithm of Section 4.1 if \(j\) is a u-leaf then we do all the \(cmods\) on it at the same time using parallel vector addition to perform the modifications in logarithmic time. Column \(j\) is to be modified by all columns \(k\) such that \((j, k)\) is in \(G^*\) and \(k\) is a descendant of \(j\) in the elimination tree. Instead of doing all the modifications at the same time we can try to pipeline the process and modify \(j\) by \(k\) as soon as column \(k\) is computed. Thus, right after column \(k\) is computed (i.e., right after \(cdiv(k)\)) we would do all the \(cmods\) that column \(k\) is involved in with its ancestors in the tree in parallel. This would allow us to eliminate the \(\log n\) factor from the time bound for certain classes of graphs.

If we consider the example of parallel Cholesky factorization in Section 4.1.1 the pipelining version would first do \(cdiv(1)\), \(cdiv(2)\), and \(cdiv(6)\) just as the original version. But then the pipelining version would perform \(cmod(3, 1)\), \(cmod(5, 1)\), \(cmod(7, 1)\), \(cmod(4, 2)\), \(cmod(8, 2)\), and \(cmod(7, 6)\). Since we would attempt to modify column 7 by both column 1 and 6 that would take two time steps, but as no column attempts to modify column 7 in the next step this does not hold up any computation. In fact, on the example in Section 4.1.1, the pipelining version takes time proportional to the height of the elimination tree.

The pipelining version still has the processors associated with the nonzeros of \(L\), but now the processors might have to compute values at other nonzeros. For instance, if we are doing \(cmod(k_1, i)\), \(cmod(k_2, i)\), \ldots, \(cmod(k_q, i)\), then we use the processors associated with column \(i\) to do \(cmod(k_1, i)\), the processors associated
with column $k_1$ to compute $cmod(k_2, i)$, and so on. In general we use the processors associated with column $k_l$ to compute $cmod(k_{l+1}, i)$. Note that because of fill-in the nonzero structure of the lower part of column $k_l$ includes the nonzero structure of column $i$.

In addition, if more than one modification of the same column is being attempted at the same time then they will proceed one at a time. Hence the processor associated with each nonzero will only be involved in one $cmod$ at a time. Therefore we only need $m^*$ processors for the pipelining version of the numeric factorization.

For some graphs, however, the pipelining version of the algorithm takes longer. For instance, if we have an elimination tree that is a $t$-ary tree with no nontree edges then the original algorithm takes $O(h \log t)$ time, but the pipelined version takes $O(ht)$ time. This is because in $t$-ary trees no column is computed until just before it is needed, so we gain nothing by doing the $cmods$ as early as possible.

In the case of some special classes of graphs we can reduce the time bound for parallel Cholesky factorization by $\log n$ using pipelining. One of these graphs is the two-dimensional finite element graphs discussed in Section 4.2.

**Theorem 4.2** On two-dimensional finite element graphs ordered by wide quotient tree ordering the pipelining version of the numeric factorization algorithm runs in $O(\sqrt{n})$ time.

**Proof:** As described in the previous section all the edges of $G^*$ are between adjacent levels of separators in the elimination tree of the two-dimensional finite
element graph when the graph is ordered by wide quotient tree ordering. In the pipelining version of the factorization algorithm we only try to modify each vertex \( j \) by at most two descendants at the same time. That is, one from each of the two descendant separator chains. Thus the number of \textit{cmods} that we have to do sequentially is no more than \( 2h \), so the total time is \( O(h) \), or \( O(\sqrt{n}) \). \( \Box \)

We have already showed that if the graph is ordered by wide quotient tree ordering then we can reduce the number of processors from \( n \log n \) to \( n \). We have now showed that the time bounds can be reduced from \( \sqrt{n} \log n \) to \( \sqrt{n} \). This brings the time-processor product of the algorithm for two-dimensional finite element graphs down to \( n^{3/2} \), which is optimal with respect to the best sequential algorithm not using fast matrix multiplication.

**Theorem 4.3** On bounded degree graphs the pipelining version of the numeric factorization algorithm runs in \( O(h) \) time.

**Proof:** In bounded degree graphs no vertex has degree greater than a constant \( c_1 \). Then in the filled graph no vertex will have degree higher than \( c_1 h \). A vertex \( j \) might have a degree close to \( c_1 h \) if it is high in the elimination tree and has \( c_1 - 1 \) nontree edges to vertices far down in the tree, in which case at most \( h \) fill edges can appear for each original nontree edge. This means that if we use the pipelining version of the factorization algorithm there will only be at most \( c_1 \) \textit{cmods} being attempted at the same time on \( j \) and the time for the algorithm is \( O(c_1 h) \) or \( O(h) \). \( \Box \)

The original numeric factorization algorithm would need \( \Omega(h \log h) \) time in the
worst case on the bounded degree graph.

We would like to be able to characterize which elimination trees can be pipelined. In the following theorem we identify a class of such elimination trees. This class does not contain all elimination trees that can be pipelined but we believe that it represents an important subset.

**Theorem 4.4** If every vertex in the elimination tree $T(A)$ is only adjacent in $G(A)$ to a constant number of its unrelated descendants in $T(A)$ then we can use pipelining to perform the numeric factorization in $O(h)$ time.

**Proof:** Let $cmod(j, i)$ and $cmod(j, k)$ be two column modifications that need to be done. Vertex $j$ is an ancestor of both $i$ and $k$ in the elimination tree. If $i$ is an ancestor of $k$ in the tree (or vice versa) then the two column modifications cannot be attempted simultaneously in the pipelining version of the numeric factorization. However, if $i$ and $j$ are unrelated then the modifications might be tried at the same time.

Thus, if every vertex is only adjacent in $G(A)$ to a constant number of unrelated descendants then there will be no more than a constant number of conflicts in column modifications at each vertex. The numeric factorization can therefore be done in time proportional to the height of the elimination tree. □

Both wide quotient tree ordered two-dimensional finite element graphs and the more general bounded degree graphs are special cases of this theorem. In bounded degree graphs each vertex only has a constant number of neighbors, thus trivially satisfying the condition in the theorem. In the elimination trees for wide quotient
tree ordered two-dimensional finite element graphs each vertex is only adjacent in $G(A)$ to the vertices in the two separator chains in the separator level below.

Elimination trees for dense matrices are another special case of the above theorem. Thus, on dense systems the pipelining algorithm runs in time $O(n)$ with $n^2$ processors. These are the same resource bounds as the best systolic algorithm for dense systems has [OS85].

Theorem 4.4 can be extended by allowing a constant number of vertices to violate the condition in the theorem. If these vertices have no more than $t$ unrelated descendants each, then we can use pipelining to get an $O(h + t)$ time numeric factorization algorithm.

### 4.4 An NC Algorithm

The time bounds for the preceding algorithms all involve the height of the elimination tree. This quantity is only related to the nonzero structure of the matrix, not to its size. The height can be anywhere between 0 and $n - 1$ and very sparse matrices can have tall elimination trees. For instance, a tridiagonal matrix has an elimination tree of height $n - 1$, but only $3n - 2$ nonzeros. We can speed up our algorithms by permuting the matrix to reduce the height of the elimination tree. It would be better though, if we had an algorithm that had a low time bound that was independent of the matrix structure.

Algorithms are called NC algorithms if they only use polylogarithmic time (i.e., $O(\log^c n)$, where $c$ is a constant) and the number of processors is polynomial in $n$. 
The previous algorithms are not NC algorithms because, even by reordering the matrix, it is not possible to make the height of the elimination tree polylogarithmic except in certain special cases like graphs of bounded treewidth. However, we can develop an NC algorithm by using each step to process "leaf paths", instead of single leaves.

**Definition 4.5** A leaf path in a tree is a maximal path of single-child vertices ending in a leaf.

We can similarly define a u-leaf path to be a leaf path in the unmarked part of the elimination tree. In Figure 4.4 (a) the u-leaf paths are (1 3), (2 4), and (6). The previous factorization algorithms started working on all the u-leaves simultaneously. In this version we will start working on all the u-leaf paths simultaneously. Each u-leaf path consists of one or more columns of $A$. Even though those columns are not necessarily consecutive, we may assume that they are, since the columns in between do not affect the columns of the path. Thus, each u-leaf path $P$ forms a submatrix $H$ of $A$ (see Figure 4.6). The nonzeros in the columns of the submatrix $K$ correspond to edges between descendants of the path $P$ in the elimination tree and vertices on the path. Similarly, the nonzeros in $M$ correspond to edges between vertices on $P$ and vertices that are ancestors of $P$.

We compute the Cholesky factorization for the columns that correspond to the path in the elimination tree using an algorithm for dense matrices that takes $O(\log^3 k)$ time, where $k$ is the length of the path. Since a u-leaf path can be no longer than the height of the elimination tree, the time is bounded by $O(\log^3 h)$. 
We start by modifying the submatrix $H$ by the relevant columns of $K$ and the submatrix $M$ by the same columns of $N$. This can be done in polylogarithmic time. Then we compute the Cholesky factorization of a matrix composed of $H$ and the nonzero rows of $M$, using an NC algorithm for dense matrices. This gives us the correct Cholesky factorization of $H$ and $M$ within $A$.

When we start considering the path $P$, all the columns of $K$ and $N$ that are relevant to $H$ and $M$, i.e., all the descendants of $P$ in the elimination tree, have already been computed. Thus, we can modify the columns of $H$ and $M$ by the relevant columns of $K$ and $N$ immediately. Each column $j$ of $H$ and $M$ will be modified by those columns of $K$ and $N$ that have a nonzero in row $j$ of $K$. Those columns of $K$ and $N$, appropriately multiplied by the entry in row $j$ of $K$, will be summed up and then subtracted from columns $j$ of $H$ and $M$. There can be no more than $O(n)$ columns involved, each with no more than $h$ nonzeros. Thus, an upper bound on the number of nonzeros involved is $O(nh)$, but since $nh \geq m^*$ a better upper bound is $m^*$. Hence, the time to modify each column of $H$ and $M$
Figure 4.7: The matrix $R$ is at most $h \times h$

is $O(\log h)$ with no more than $m^*$ processors. We will modify all the columns of $H$ and $M$ simultaneously, so the total number of processors will be no more than $hm^*$.

Next, we will form a new matrix $R$ consisting of $H$, $M'$ and $I$, where $M'$ is $M$ with the zero rows omitted and $I$ is the identity matrix (see Figure 4.7). The matrix $R$ contains no more than $h$ rows and columns, since the nonzero rows of $M$ correspond to ancestors of the path $P$ in the elimination tree.

Now when we compute the Cholesky factorization of $R$, it will give us the correct factorization of the submatrices $H$ and $M$ within $A$. The lower right hand part will contain the modifications we need to do on the ancestors of the leaf path $P$. By storing this submatrix we can do the modifications more quickly, giving us a multifrontal-like method of factorization [DR83,Duf86]. However, this requires more space and it will not improve the overall time bound of the algorithm. Thus, we will not pursue this idea in any detail and just do the modifications independently. For that we only need the part of $R$'s Cholesky factor that corresponds to $H$ and $M$; the other parts are discarded. There are two different algorithms for finding the Cholesky factorization in polylogarithmic time. We can use Pan's iterative Cholesky factorization algorithm. It runs in $O(\log^3 n)$ time using $n^a$ pro-
cessors [Pan87]. It gives an approximate solution and requires the matrix to be well-conditioned. Then there is Preparata and Sarwate's direct inversion algorithm, which runs in $O(\log^2 n)$ time using $n^{\alpha+0.5}$ processors. It is numerically unstable, as are all known direct NC algorithms for inverting matrices. Datta [Dat85] shows how Preparata and Sarwate's inversion algorithm can be used to compute Cholesky factorization within the same resource bounds. Here $n$ is the size of the matrix and $n^\alpha$ is the best time bound for matrix multiplication (currently $\alpha \approx 2.376$ [CW87]). We can use either of the above algorithms to factor $R$, but in what follows we will assume that the iterative algorithm is used.

We now know how to compute the columns corresponding to leaf paths in the elimination tree in $O(\log^3 h)$ time with $h^\alpha$ processors (or $O(\log^2 n)$ time with $h^{\alpha+.5}$ processors). If we compute all the u-leaf paths simultaneously we will only have to do that $\log n$ times to mark the whole elimination tree. This follows from Miller and Reif's analysis of their parallel tree contraction scheme [MR85]. Thus, the total time for the algorithm is $O(\log n \log^3 h)$. To find the number of processors required we need to estimate the maximum number of processors ever used at the same time.

Since a leaf path can not be longer than $h$, we will need no more than $h^\alpha$ processors to factor a leaf path. The worst possible case would be if all the u-leaf paths at some level had length $h$. There could only be $n/h$ of them, but we would need $nh^{\alpha-1}$ processors to factor all of the u-leaf paths at the same time. Using the straightforward matrix multiplication with $\alpha = 3$, we need $nh^2$.

The above processor bound is for arbitrarily structured sparse matrix. If we
know something about the structure of the matrix that can be exploited to reduce the processor bound.

Matrices whose graphs satisfy an $n^{\beta}$-separator theorem, for some $\beta \geq 1/2$, (i.e., the graphs and their subgraphs have a separator whose size is $O(n^\beta)$, see [LT79]) can be ordered by the nested dissection algorithm to give an elimination tree of height $O(n^\beta)$. The first graph separator will form a chain of length $n^\beta$ in the tree and it will require $n^{\alpha \beta}$ processors to factor. The next set of graph separators forms chains of length at most $(n/2)^{\alpha \beta}$ so $n^{\alpha \beta}$ processors will be enough to factor the corresponding submatrices. Going down the levels of graph separators like this we see that the matrix can be factored in $O(\log^4 n)$ time with $n^{\alpha \beta}$ processors. For example, if the graph is a two-dimensional finite element graph then a wide quotient tree order (see [ZG85]) gives a running time of $O(\log^4 n)$ with $n^{\alpha/2}$ processors. A 3-dimensional grid graph satisfies a $n^{2/3}$-separator theorem. Thus, its matrix can be factored in $O(\log^4 n)$ time with $n^{2\alpha/3}$ processors.

Pan and Reif [PR87] have presented an algorithm that solves $Ax = b$ in $O(\log^3 n)$ time with $(m + n^{\alpha \beta})/\log n$ processors if $G(A)$ satisfies an $n^{\beta}$-separator theorem. Their algorithm uses nested dissection and computes a special recursive factorization of $A$. However, their algorithm is not as general as ours. Our algorithm needs $O(\log^2 n + \log n \log^3 h)$ time and $m^* + nh^{\alpha - 1}$ processors to solve a general sparse linear system.

The motivation for using a dense Cholesky factorization algorithm for the leaf paths is that for each nonzero $(i, j)$ of $H$, the rest of the row, locations $(i, j + 1), (i, j + 2), \ldots, (i, i - 1)$, will become nonzero because of fill-in. Similarly the rows
of $M$ will get filled in from the first nonzero in each row. Thus, the Cholesky factor of $R$ will be quite dense, most likely considerably more dense than the rest of $L$. However, there are cases where this is not true, but we may be able to avoid those cases if we reorder $A$ initially to reduce the height of its elimination tree. For instance, in the elimination tree for a tridiagonal matrix the longest u-leaf path is $n - 1$, so we need $n^{\alpha+0.5}$ processors to factor the tridiagonal matrix, which only has $O(n)$ nonzeros. On the other hand, when the permutation of a matrix is the result of a nested dissection in which minimal separators have always been chosen each u-leaf path is a clique in the graph and its corresponding submatrix $H$ is full.

4.4.1 An Sample Execution of the NC Algorithm

We consider again the matrix from Figure 2.1 of Chapter 2. The original elimination tree has three u-leaf paths, as we can see in Figure 4.8 (a). The first u-leaf paths, which are circled by a dashed line, are (1 3), (2 4), and (6). Since they are on the first level of the algorithm their columns in the matrix do not need to be modified by any other columns.

For the u-leaf path (1 3) the matrix $M'$ consists of rows 5, 7, and 8, which are the nonzero rows in columns 1 and 3. Thus, the matrix $R$ for u-leaf path (1 3) is $5 \times 5$ (see Figure 4.9). In the case of the u-leaf path (2 4), the matrix $M'$ has rows 5 and 8, so the matrix $R_{(24)}$ is $4 \times 4$. For the u-leaf path that only contains the vertex 6, $M'$ only has row 7, so $R_{(6)}$ is only $2 \times 2$.

We now use the NC Cholesky factorization algorithms to factor the matrices $R_{(13)}$, $R_{(24)}$, and $R_{(6)}$. This gives us the values for the columns 1, 2, 3, 4, and 6.
Figure 4.8: The u-leaf paths of the elimination tree
\[ R_{(13)} = \begin{bmatrix} X & X & X & X \\ X & \emptyset & \emptyset & X \\ X & \emptyset & 1 \\ X & \emptyset & 1 \\ X & 1 \end{bmatrix} \]

\[ R_{(24)} = \begin{bmatrix} X & X \\ X & X & \emptyset \\ X & 1 \\ X & \emptyset & 1 \end{bmatrix} \]

\[ R_{(6)} = \begin{bmatrix} X & X \\ X & 1 \end{bmatrix} \]

\[ R_{(578)} = \begin{bmatrix} X & \emptyset & \emptyset \\ \emptyset & X & X \\ \emptyset & X & X \end{bmatrix} \]

Figure 4.9: The $R$ matrices that come up in the example

When we have marked those vertices in the elimination tree we get the u-tree in Figure 4.8 (b). That tree only has the u-leaf path (5 7 8). Now we have to modify those three columns by the previous columns. Column 5 needs to be modified by columns 1, 3, and 4. Column 7 is modified by columns 1, 3, and 6, and column 8 has to modified by columns 2, 3, and 4. Because this is the last level of the algorithm the matrix $M'$ does not have any rows, so $R_{(578)}$ just has three columns/rows.

We then factor $R_{(578)}$ using the polylogarithmic time factorization algorithm
giving us the values for the last three columns of the matrix.

In Section 4.1 we needed five steps to compute the factorization, here we only needed two. However, we probably needed many more processors in this version in order to compute the factorization of the $R$ matrices. For instance, the earlier algorithm needs a total of $m^*$ processors; in our example $m^* = 21$. On the other hand, the matrix $R_{(13)}$ is $5 \times 5$, so we need $5^\alpha$ processors to factor it. If we assume that $\alpha = 3$, then we need 125 processors just to factor $R_{(13)}$. The matrix $R_{(24)}$ has to be factored at the same time and that will require another $64(= 4^3)$ processors.

### 4.5 Questions about Elimination Trees

The running time of the algorithm in Section 4.1 is proportional to the height of the elimination tree. The height depends on the nonzero structure of the matrix, but not necessarily on the sparsity. As a rule, dense matrices have tall elimination trees, but sparse matrices can also have tall elimination trees. The height can be changed to some degree by permuting the rows and columns of the matrix, which corresponds to renumbering the vertices in the elimination tree. Finding an ordering that gives the minimum height elimination tree is NP-complete [Pot88a], so we need to rely on heuristics to reduce the height. We now give an overview of recent research in this area and discuss feasible avenues for further work.

As things stand now there are basically two different approaches to reducing the height of the elimination tree. The first one is to order the matrix for low fill and then reduce the elimination tree height while maintaining the same fill
structure. The second approach is to order the matrix for low fill and a shallow elimination tree simultaneously. In the first technique we start of by trying to reduce the amount of fill by running minimum degree [GL87], nested dissection [GL81], or some other fill reduction heuristic. Then we use a method of Jess and Kees [JK82] that reorders a matrix to minimize the height of its elimination tree without introducing any additional fill. This gives us the shortest elimination tree with that particular fill structure. In practice we would not use the original algorithm of Jess and Kees but one of the more efficient versions that have been published recently [LM87, Pot88b, LP88].

The second approach is to think primarily about reducing the height of the elimination tree. An algorithm by Leiserson and Lewis [LL88] does this by generalizing nested dissection. On the test problems they considered, their algorithm generates shorter elimination trees and almost as little fill as the Jess and Kees algorithm preceded by the minimum degree algorithm. In some cases we might not care how much fill we get, as long as we get a short elimination tree. In most instances this is not true. Therefore, algorithms that try to reduce the elimination tree height will also have to worry about not introducing large amounts of fill.

Leiserson and Lewis do not provide a bound on how close their answer is to the optimal tree height. It would be interesting to know if it is possible to come up with algorithms that have such bounds for certain classes of graphs. Most sparse graphs, however, have elimination trees of height $\Omega(n)$ for all orderings. This follows from a theorem in [LRT79] which says that for all $\varepsilon > 0$ there is a constant $c(\varepsilon)$ such that almost all $n$-vertex graphs with at least $c(\varepsilon)n$ edges have a
fill-in clique of at least $(1 - \varepsilon)n$ vertices for any ordering. Thus, their elimination trees have height at least $(1 - \varepsilon)n$.

One of the variants on the Jess and Kees algorithm to find minimum height elimination trees with no additional fill is a tree rotation algorithm by Liu [Liu87a] that runs in linear time and in practice almost always finds the same tree as the Jess and Kees algorithm. The tree rotations of this algorithm reduce the height of the elimination tree without introducing any fill. It might be possible to design more powerful rotations that allow some constant (or perhaps $\log n$) amount of fill but achieve more reduction in the tree height.

This brings up the question of whether we can always trade fill for height (i.e., can we get shorter elimination trees by allowing more fill). George and Liu [GL88] have answered this in the negative. They give a sufficient condition on an elimination tree so that additional fill does not help reduce the height of the elimination tree. In particular, they prove that if every vertex $v$ in the elimination tree is adjacent to a leaf $w$ in the subtree rooted at $v$, and $w$ is as far from $v$ as any other leaf in the subtree, then the height of the shortest elimination tree over all orderings is the same as the height of the shortest tree over orderings that do not introduce any fill. This condition is rather restrictive but nested dissection ordering tends to produce elimination trees that fulfill this condition. Thus, if an elimination tree satisfies the condition then the Jess and Kees algorithm will produce the shortest elimination tree possible.

A related problem is whether minimum fill and minimum elimination tree height can be achieved simultaneously for certain classes of graphs. For any graph from
that class, is there an ordering that produces minimum fill and the shortest possible elimination tree? This does not seem likely, (e.g., it is not true for a graph that just consists of a single chain of vertices) but it is more likely that there are orderings that minimize, say, elimination tree height, while keeping the amount of fill within some factor of the minimum. However, finding such orderings in the general case is probably NP-complete.

Another open problem is to efficiently determine when we have a minimum height elimination tree. This should be easier than finding it but it could turn out to be just as hard.

A promising way to reduce the height of the elimination trees resulting from the nested dissection algorithm is to reorder vertices within separator chains in the tree so that the chains correspond to cliques in $G^*$. A chain in $T$ is a simple path consisting of one-child vertices. The following lemma describes the basic idea.

**Lemma 4.6** Let $C$ be a chain in $T$ that is not a clique in $G^*$, let $v$ be the highest numbered vertex in $C$ that is not adjacent in $G^*$ to all the other vertices in $C$, and let $w$ be the highest numbered vertex in $C$ not adjacent in $G^*$ to $v$. We can make $w$'s parent the parent of $v$ in $T$ and thus reduce the height of the chain $C$ by one.

**Proof:** Let $w'$ be the parent of $w$ and $v'$ be the child of $v$ as in Figure 4.10. Because of Lemma 2.3 from Chapter 2 if any of the descendants of $w$ were adjacent to $v$ then $(v, w)$ would be a fill edge. Therefore, none of these vertices are adjacent to $v$ and we can make $v$ the child of $w'$ by renumbering all the vertices from $w'$ to $v$. The height of this subtree is reduced by one by this transformation as can
clearly be seen in Figure 4.10. □

Note that we might incur some additional fill. If $v$ has an edge to a vertex $x$ higher up in the elimination tree then in the new tree all the vertices from $v'$ to $w'$ will also have a fill edge to $x$. On the other hand, any edge $(u, x)$ in $G$, where $u$ is a descendant of $v$ and $x$ is an ancestor of $v$, would produce a fill edge from $v$ to $x$ in the original ordering but not in the new ordering.

We may be able to use the above lemma to reduce the height of elimination trees that have chains that are not cliques in $G^*$. The lemma reduces the height of a chain which may or may not reduce the height of the elimination tree. Elimination
trees resulting from nested dissection are good candidates because they have long chains contributing to the height that might not be cliques.

This method makes all chains in the elimination tree into cliques in \( G^* \). Chains that contribute to the height of an elimination tree would have to be cliques in a minimum height elimination tree, but this is not a sufficient condition to get minimum height trees. For instance, the star graph, which has only edges between one vertex and all the others, has an elimination tree of height \( n - 1 \) which is a chain and a clique. However, the minimum height elimination tree for the star graph has height 1.

In our NC factorization algorithm of section 4.4, we use a dense factorization algorithm to factor submatrices corresponding to leaf paths. If we could make those submatrices smaller and more dense (i.e., make the leaf paths more cliquish) that would make our NC algorithm more efficient. Since chains are (parts of) leaf paths we could use the idea in Lemma 4.6 to make some leaf paths more dense. However, as most leaf paths are not chains we need a more general method. Perhaps some reordering could be done on the fly to accomplish this.

All of the methods mentioned above are sequential. Some of them could doubtless be parallelized, but no one has specifically designed a parallel algorithm to solve this problem. We know of at least two parallel ordering algorithms, by Zmijewski [Zmi87] and by George, Liu, and Ng [GLN87]. However, both those algorithms are mostly concerned with reducing the fill and achieving a good splitting of columns between the processors. It is not obvious that they get any provable reduction in the height of the elimination tree.
Chapter 5

Triangular System Solving

After the Cholesky factorization of $A$ into $LL^T$, solving the original system $Ax = b$ is equivalent to solving the two triangular systems $Ly = b$ and $L^T x = y$. Solving $Ly = b$ is called forward solve and sequentially it is done by computing $y(1)$ first, then $y(2)$, and so on. We call the solution of the upper triangular system $L^T x = y$ back solve, and in it $x(n)$ is calculated first, then $x(n-1)$, and so on.

A dense triangular system can be solved sequentially in $O(n^2)$ time, and a sparse one takes $O(m^*)$ time, where $m^*$ is the number of nonzeros. A dense system can be solved in parallel in $O(n)$ time with $n$ processors by a straightforward parallelization of the sequential algorithm. By carefully reallocating the available processors we can use the same method to solve a sparse triangular system in $O(n)$ time using $m^*/n$ processors. The best NC algorithms for triangular system solving takes $O(\log^2 n)$ time using $n^\alpha$ processors [CK75,BM75,SB77].

In this chapter we will show how to solve a sparse triangular system in $O(h \log n)$
time, where $h$ is the height of the elimination tree. We first assume that we have
$m^*$ processors, but then in Section 5.2 we show how to reduce the number of
processors to $m^*/h$. Finally we present an NC algorithm to solve sparse triangular
systems that runs in $O(\log n \log^2 h)$ time using $nh^{\alpha-1}$ processors.

5.1 Forward and Back Solve

In the forward solve $Ly = b$ we can solve for a variable $y(j)$ immediately if the
only nonzero in row $j$ of $L$ is the diagonal element. According to Lemma 4.1 of
the last chapter that occurs when vertex $j$ is a leaf of the elimination tree of $A$.
Thus, we can solve in parallel for those variables that correspond to leaves of the
elimination tree.

When we have solved for this first set of variables we need to substitute their
values into the other equations and subtract the results from the right hand side.
Then again we can solve for those variables whose rows now only have the diagonal
element nonzero.

The resulting algorithm in Figure 5.1 is very similar to the parallel Cholesky
algorithm of Section 4.1. We start with an unmarked elimination tree, work our
way up the tree and solve for the variables that correspond to the current u-leaves
(i.e., the leaves of the unmarked part of the tree) using the routine $f_{sol}$. We then
mark the u-leaves, giving a new u-tree. For each new u-leaf $j$ we substitute the
values of the already calculated variables into the equation defining the variable $j$
and subtract the results from $b(j)$. This is done with the routine $f_{sub}$.
algorithm parallel-forward-solve
begin
  unmark every vertex of $T$
  repeat
    for each u-leaf $j$ of $T$ pardo
      fsol$(j)$
      mark $j$
    od
    for each u-leaf $j$ of $T$ pardo
      for each $k < j$ with $(j,k)$ an edge of $G^*$ pardo
      fsub$(j,k)$
    od
  od
  until every vertex is marked
end

Figure 5.1: Parallel forward solve
algorithm fsol(j)
begin
  \( y(j) := b(j) / L(j, j) \)
end

algorithm fsub(j, k)
begin
  \( b(j) := b(j) - L(j, k) \times y(k) \)
end

Figure 5.2: Routines fsol and fsub

The only difference between the parallel Cholesky algorithm and the algorithm in Figure 5.1 is that we have the routines fsol and fsub instead of cdiv and cmod respectively. As the new routines, in Figure 5.2, require much less computation than the old ones, this algorithm needs fewer processors than the parallel Cholesky algorithm. For now we assume that we have \( m^* \) processors, but in Section 5.2 we reduce that number.

The computation of the fsub's for a single u-leaf \( j \), which is the inner loop in the algorithm of Figure 5.1, will take at most \( O(\log n) \) time. We first sum up all the values that need to be subtracted from \( b(j) \) and then subtract the result from \( b(j) \). The execution of a fsub is only a single floating point operation so we only need one processor for each fsub invocation. The same holds true for fsol.

We have so far only talked about forward solve. We will now show how the back solve \( L^T x = y \) can be done in parallel. In back solve we start at the root of
the elimination tree and work our way down to the leaves.

The only row in $L^T$ with no off-diagonal nonzeros is row $n$. If there was another row $j$ with no off-diagonal nonzeros then vertex $j$ would not have any ancestors in the elimination tree and would thus be the root of the elimination tree. However, $n$ is the root of the elimination tree, so all other rows must have some off-diagonal nonzeros.

We can start by solving for the variable $x(n)$. We then mark the root of the elimination tree and look at the new u-roots, which are the children of vertex $n$ in the tree and are the roots of the unmarked part of the elimination tree. For each of these new u-roots $j$ we will need to substitute the value of $x(n)$ into their equations, before we can solve for them. Then these u-roots get marked and we continue like that down the tree. The algorithm is in Figure 5.3.

In forward solve we used $f_{sol}$ to solve for a variable and $f_{sub}$ to substitute the value of a variable into an equation. Back solve uses $b_{sol}$ and $b_{sub}$ for the same purpose.

This back solve algorithm requires that given a parent we can find all its children in the elimination tree. We can do that by preprocessing the tree, but there is another way we can do the back solve. We can remember the order in which the vertices of the elimination tree were marked in the forward solve and mark them now in reverse order. That order will be slightly different from the order we get from the algorithm in Figure 5.3, but the time bound will be the same.

The time bound for the back solve is $O(h \log n)$, and we only need $n$ processors to preprocess the elimination tree, in addition to the $m^*$ processors that we assume
algorithm parallel-back-solve
begin
  unmark every vertex of \( T \)
  repeat
    for each u-root \( j \) of \( T \) pardo
      bsol(\( j \))
      mark \( j \)
    od
    for each u-root \( j \) of \( T \) pardo
      for each \( k > j \) with \( (j, k) \) an edge of \( G^* \) pardo
        bsub(\( j, k \))
      od
    od
  until every vertex is marked
end

Figure 5.3: Parallel back solve
we have. If we are not using the algorithm of Figure 5.3, but doing the vertices in the reverse order from forward solve then we can also use the same processor allocation. This is because each level of back solve has the same computation as the corresponding level of forward solve, but we are doing the levels in the reverse order.

5.1.1 An Example

The behavior of the forward solve algorithm is very similar to the numerical factorization algorithm in Chapter 4. Let us therefore look at a sample execution of the back solve algorithm. The example is again the one from Chapter 2. Figure 5.4 (a) shows the upper triangular system \( L^T x = y \) to be solved and its corresponding elimination tree \( T \).

We use the algorithm from Figure 5.3. It starts by solving for the variable corresponding to the root of \( T \), which in our case is \( x(8) \). After marking the vertex 8 the new u-root of \( T \) is vertex 7 (see Figure 5.4 (b)). The value of \( x(8) \) needs to be substituted into the 7th equation before we can solve for \( x(7) \). We can now perform \( bsol(7) \) and mark vertex 7. As we can see by Figure 5.4 (c) there are now two u-roots, vertices 5 and 6 (i.e., rows 5 and 6 are now the only ones with no off-diagonal nonzeros). We then perform the substitutions \( bsub(6, 7) \), \( bsub(5, 7) \), and \( bsub(5, 8) \). Now the variables \( x(5) \) and \( x(6) \) can be solved for. Marking vertices 5 and 6 gives the situation in Figure 5.4 (d). After performing the appropriate substitutions we can solve for \( x(3) \) and \( x(4) \) simultaneously, and then \( x(1) \) and \( x(2) \) at the same time.
Figure 5.4: A sample execution of parallel back solve
5.2 Reducing the number of Processors

In the previous section we assumed that we had a processor for every nonzero of the matrix $L$. This is unnecessarily wasteful as we can see from the following lemma.

**Lemma 5.1** Each level of the parallel forward solve algorithm in Figure 5.1 needs no more than $n$ processors to run in $O(\log n)$ time.

**Proof:** Each level of the algorithm consists of some $fsol$'s and some $fsub$'s. There are only $n$ executions of $fsol$ in the whole algorithm and each of them only needs one processor.

In the case of $fsub$ note that if we are doing $fsub(j, k)$ then we cannot be doing $fsub(i, k)$ at the same time, where $i$ is some column different from $j$. The vertex $k$ has to be in the subtree of the elimination tree rooted at $j$ and since we are only doing one level of the tree at a time $k$ can not also be in the subtree rooted at $i$. Thus, there can be no more than $n$ $fsub$ executions going on simultaneously. For each $fsub$ execution we only need one processor.

Hence the total number of processors required to do one level of the parallel forward solve algorithm in $O(\log n)$ time is at most $n$. \Box

By carefully reallocating the processors we can execute the parallel forward solve algorithm in $O(h \log n)$ time with only $m^*/h$ processors. Below we will sketch how this is done.

The parallel forward solve algorithm in Figure 5.1 consists of $h$ phases, each of which involves a number of $fsol$'s followed by some number of $fsub$'s.
We find the level number for each vertex by preprocessing the elimination tree with the tree algorithm of Tarjan and Vishkin [TV84]. We then sort the vertices on their level numbers. This can be done in $O(hn \log n/m^*)$ time using $m^*/h$ processors.

We do the first set of $f_{sol}$s on the vertices at the first level, which correspond to the first u-leaves of the elimination tree. The $f_{sol}$s are done $m^*/h$ at a time.

Now the first u-leaves are marked and we consider the next set of u-leaves. We assume that we know how many nonzeros each row of the matrix has. Then we can allocate the processors to the $f_{sub}$s that have to be done on this level in $O(\log n)$ time. The $f_{sub}$s are done $m^*/h$ at a time.

The $f_{sol}$s at the next level of the algorithm still use the same u-leaves. We do those $f_{sol}$s $m^*/h$ at a time and continue like that for the remaining levels of the algorithm.

We now look in more detail at the amount of time required. The time needed to do the $f_{sol}$s at each level is the number of u-leaves at that level divided by $m^*/h$. Thus, the total time is

$$\sum_{1 \leq l \leq h} \left\lceil \frac{u\text{-leaves}(l)}{m^*/h} \right\rceil,$$

where $u\text{-leaves}(l)$ is the number of u-leaves at level $l$. The above is no more than

$$h + \frac{h}{m^*} \sum_{1 \leq l \leq h} u\text{-leaves}(l),$$

or $h + hn/m^*$. Since $n/m^* \leq 1$ the total time for the $f_{sol}$s is $O(h)$.

Let $\#f_{subs}(i, l)$ be the number of $f_{sub}$s that need to be done in row $i$ during level $l$ of the algorithm. Then if we have $\#f_{subs}(i, l)$ processors then we can do all
the $f_{subs}$ in row $i$ in time $O(\log(\#f_{subs}(i, l)))$. Certainly $\log(\#f_{subs}(i, l)) \leq \log n$. We will assume that each row of $f_{subs}$ will take $O(\log n)$ time if we have enough processors. How many processors do we need? For each level $l$ we need

$$\sum_{1 \leq i \leq n} \#f_{subs}(i, l)$$

processors. Since we only have $m^*/h$ processors the time required for level $l$ is

$$\left\lceil \frac{\sum_{1 \leq i \leq n} \#f_{subs}(i, l)}{m^*/h} \right\rceil \log n.$$

For the whole algorithm the time is no more than

$$\log n \left( h + \frac{h}{m^*} \sum_{1 \leq j \leq h} \sum_{1 \leq i \leq n} \#f_{subs}(i, l) \right).$$

The total number of $f_{sub}$'s that need to be computed during the entire algorithm is $m^*$. Thus the above expression simplifies to

$$(h + m^* \frac{h}{m^*}) \log n,$$

which is $O(h \log n)$.

The total time for the parallel forward solve algorithm, including the pre-processing and processor allocating, is $O(h \log n)$ with only $m^*/h$ processors. The parallel back solve algorithm can be similarly modified to achieve the same resource bounds.

### 5.3 NC Algorithms

The forward and back solve can be changed into NC algorithms in a way similar to that we used in Chapter 4 for Cholesky factorization. We now use a polylogarithmic time algorithm for triangular systems on leaf paths in the elimination tree.
algorithm NC-forward-solve
begin
  unmark every vertex of $T$
repeat
  for each u-leaf path $P$ of $T$ pardo
    solve the subsystem corresponding to $P$
    mark all the vertices of $P$
  od
  for each u-leaf path $P$ of $T$ pardo
    for each vertex $j$ of $P$ pardo
      for each $k < j$ with $(j, k)$ an edge of $G^*$ pardo
        $f_{sub}(j, k)$
      od
    od
  od
until every vertex is marked
end

Figure 5.5: The NC forward solve algorithm

In the previous forward solve algorithm we started by working on all the leaves simultaneously. In this version we will, as we did in Chapter 4, start working on all the leaf paths at the same time. Each leaf path consists of one or more columns of $L$. Even though those columns are not necessarily consecutive, we will assume that they are. Thus, as before, each leaf path $P$ forms a submatrix $H$ of $L$, which has a full first band below the diagonal.

There are a few parallel algorithms for solving dense $n \times n$ triangular systems in $O(\log^2 n)$ time with $n^a$ processors [CK75,BM75,SB77]. The processor bound of
these algorithms is the same as the bound for the iterative inversion algorithm of Pan and Reif for general matrices [PR87]. However, these algorithms for triangular matrices are direct (although not numerically stable). The best processor bound for a direct algorithm to solve general systems is $n^{\alpha+0.5}$. We will use one of these direct triangular system solve algorithms to solve the system consisting of the submatrix $H$ and the corresponding right hand side, for each leaf path. Since each leaf path can be no longer than $h$, the time to solve a path is $O(\log^2 h)$. The total time for the whole algorithm will be $O(\log n \log^2 h)$ because we will only have to do $\log n$ stages. We will need no more than $h^\alpha$ processors to solve each leaf path. Hence, by the argument from Section 4.4, the maximum number of processors needed for a general matrix is $nh^{\alpha-1}$.

Back solve can be done in NC by reversing the order in which forward solve processed the elimination tree. Thus, in NC back solve, we will solve the subsystems corresponding to the u-leaf paths in the opposite order from the way in which they were solved in NC forward solve. The time and processor bounds stay the same.

### 5.3.1 An Example

Using the same example as in Section 5.1.1 the NC forward solve will first find u-leaf paths (1 3), (2 4), and (6). When those vertices have been marked the elimination tree only consists of the u-leaf path (5 7 8).

Figure 5.6 shows how the NC back solve algorithm works on that example. It first uses the dense NC triangular solve algorithm to solve the subsystem consisting
Figure 5.6: A sample execution of the NC back solve algorithm
of columns/rows 5, 7, and 8. Then the values of those variables are substituted into the other equations. Finally the three subsystems that contain columns/rows 1 and 3, 2 and 4, and 6 are solved with the NC algorithm.
Chapter 6

Conclusion

In this chapter we present an overview of the thesis and discuss its main contributions. We also consider open problems suggested by our work.

6.1 Summary

In this thesis we have presented parallel algorithms to solve a sparse symmetric positive definite linear systems of equations $Ax = b$. Our parallel model of computation has been the shared memory CRCW PRAM model. The solution of sparse linear systems via Cholesky factorization generally consists of four separate stages. The first one permutes the matrix in order to make the remaining steps of the algorithm more efficient. The second step is a symbolic factorization that helps allocate memory and processors for the third step, numeric factorization. The final step is to solve two triangular systems.

We assumed that the system had already been permuted to reduce the amount
of fill and the height of the elimination tree. We then presented parallel algorithms for the three remaining stages of sparse linear system solving.

Our parallel algorithm for symbolic factorization determines the fill that occurs during the factorization. Thus, it gives us the nonzero structure of the Cholesky factor $L$ of $A$. Its running time is $O(\log^2 n)$ and it uses $m^*$ processors, where $m^*$ is the number of nonzeros in $L$.

One part of our symbolic factorization algorithm was a parallel algorithm to compute $A$'s elimination tree. Elimination trees play a prominent role in many areas of sparse matrix computation, since they describe the data dependencies among the columns of the $L$. Thus, a fast parallel algorithm to find elimination trees is important in its own right. Our algorithm runs in $O(\log^2 n)$ time using $m$ processors, where $m$ is the number of nonzeros in $A$. As the best sequential algorithm runs in almost linear time there is some room for improvement in either the time bound or the number of processors.

Our parallel numeric factorization algorithm also uses the elimination tree to compute the numeric values of $L$. The algorithm starts at the leaves and goes up the tree one level at a time. Therefore its time bound is $O(h \log n)$, where $h$ is the height of the elimination tree, using $m^*$ processors. A pipelined version of the algorithm runs in $O(h)$ time for certain classes of matrices.

We also presented an NC algorithm to factor sparse matrices. It requires a dense NC Cholesky factorization algorithm to factor submatrices. The best dense algorithm is an iterative method due to Pan [Pan87]. It takes $O(\log^3 n)$ time using $n^\alpha$ processors and it requires the matrix to be well-conditioned. Here $\alpha$ is the best
known exponent for fast matrix multiplication, currently $\alpha \approx 2.376$. Using Pan's algorithm as a subroutine, our algorithm runs in $O(\log n \ \log^3 h)$ time using $nh^{a-1}$ processors.

Finally we presented a parallel algorithm to solve sparse triangular systems. The algorithm resembles our numeric factorization algorithm and it has the same time bounds, $O(h \log n)$. However, we need only $O(m^*/h)$ processors since triangular solve does not require as much computation as Cholesky factorization. We also presented an NC algorithm for solving sparse triangular systems. It uses a fast dense NC triangular solve algorithm as a subroutine and runs in $O(\log n \ \log^2 h)$ using $nh^{a-1}$ processors.

A significant property of our parallel algorithms is that all of them, except for the NC algorithms, use the same amount of processors as space. This is important because we believe that in future highly parallel machines processors will be as abundant as memory.

6.2 Future Work

There are many interesting problems that remain unsolved. Perhaps the most important one is how to reduce the height of the elimination tree. Finding the minimum elimination tree is NP-complete [Pot88a] but there are some useful heuristics to reduce the height. We gave an overview of the main approaches and some of the associated open problems in Section 4.5. One of the questions we ask there is whether it is possible to come up with heuristic algorithms for reducing the
elimination tree height that have bounds on how close their tree height is to the optimal height for certain classes of graphs.

One possible method to reduce the elimination tree height is to use the idea presented in Lemma 4.6 of Section 4.5 on making chains in the elimination tree into cliques in $G^*$, thus reducing the tree height. However, this idea needs more work before we can decide if it is a useful way of reducing the elimination tree height.

Some of the other open problems from Section 4.5 include deciding if there are orderings that give minimum fill and minimum elimination tree height simultaneously, efficiently determining when we have a minimum height elimination tree, and finding parallel algorithms for reducing elimination tree height.

An important open problem related to our sparse NC factorization algorithm is to find a direct, stable, and efficient NC algorithm to factor dense matrices. Using our NC algorithm that would immediately translate into an efficient algorithm for sparse systems.

The sparse NC factorization algorithm would be more efficient if we could order the matrix so that the leaf paths in the elimination tree are smaller and denser. We would then make more efficient use of the dense NC algorithm that factors the submatrices corresponding to the leaf paths.

Efficient parallel LU factorization of a sparse nonsymmetric matrix $A$ remains a challenge. Sequential algorithms with partial pivoting use the nonzero structure of the Cholesky factor of $A^TA$ to predict the nonzero structure of $L$ and $U$ [GN85]. Our symbolic factorization algorithm can be used for that. However, pivoting for
numerical stability makes it hard for parallel algorithms to run in better than linear time for the numeric part.
Bibliography


