Parallel Cholesky Factorization of Sparse Matrices

John R. Gilbert *
Hjálmtýr Hafsteinsson†

87-893
December 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*This author's research was partially supported by the National Science Foundation under grant DCR-84-51385.
†Fulbright Scholar, research partially supported by IBM Corporation under contract 573616-657516.
Parallel Cholesky Factorization of Sparse Matrices

John R. Gilbert*
Hjálmtír Hafsteinsson†

Department of Computer Science
Cornell University
Ithaca, NY

December 1987

Abstract

We describe a parallel algorithm for finding the Cholesky factorization of a sparse symmetric positive definite matrix $A$. The algorithm runs in $O(h\log n)$ time with $m^*$ processors, where $h$ is the height of $A$'s elimination tree. We then show how to speed up that algorithm, so that it runs in $O(\log n \log^2 h)$ time with increased number of processors. Also, we present corresponding parallel algorithms for forward solve and back solve with the same time bounds and similar processor bounds.

* This author's research was partially supported by the National Science Foundation under grand DCR-84-51385
† Fulbright Scholar, research partially supported by IBM Corporation under contract 573616-667516
1. Introduction

The problem of solving the system $Ax = b$, where $A$ is a large sparse symmetric positive definite matrix, arises in many different branches of science and engineering. One important method for solving this problem is Cholesky factorization. The matrix $A$ is factored into $LL^T$, where $L$ is a lower triangular matrix, and the original problem is then solved by solving the triangular systems $Ly = b$ and $L^Tx = y$.

Sequentially, Cholesky factorization takes $O(n^3)$ time on a dense matrix. A tighter bound is $O(n^a)$, where $O(n^a)$ is the fastest known time required to do matrix multiplication. Currently $a = 2.376$ [CW87]. For a sparse $A$, it takes time proportional to the number of operations required to multiply $L$ and $L^T$.

Algorithms have been devised to solve $Ax = b$ in parallel. Most of them do not take into account the sparsity of $A$. There is a relatively straightforward parallelization of the sequential Cholesky factorization algorithm that takes $O(n)$ time using $n^2$ processors. There is also a parallel algorithm, which is unrelated to Cholesky factorization, that solves $Ax = b$ in $O(\log^2 n)$ time with $O(n^{a+.5})$ processors ([PS78]). The iterative method of Pan and Reif [PR85] solves $Ax = b$ in $O(\log^2 n)$ time with $n^a$ processors. However, it needs a bound on the condition number of $A$ to converge in polylog time. None of these algorithms take any significant advantage of sparsity in $A$. However, Pan and Reif also use their algorithm with nexted dissection to get an $O(\log n \log^2 s(n))$ time algorithm with $n^a$ processors if the graph $G(A)$ is $s(n)$-separable.

There has been quite a lot of experimental work done on parallel sparse Cholesky factorization ([Z87], [GH85], [OV85]), mostly on the hypercube model.

In this paper we present an algorithm that uses the elimination forest of $A$ to exploit sparsity in the Cholesky factorization of $A$. The algorithm runs in time $O(h\log n)$, using $m^*$ processors, where $h$ is the height of $A$'s elimination forest and $m^*$ is the number of nonzero elements in the resulting matrix $L$. For a rather dense $A$, the height, $h$, of $A$'s elimination forest will be close to $n$ and $m^*$ will be close to $n^2$, so the time bound will be $O(n\log n)$ using $n^2$ processors. That is a factor of $\log n$ greater than the straightforward parallel dense Cholesky algorithm. For a sparse $A$, if we can find a $\sqrt{s}$ - separator for its corresponding graph and all its subgraphs, where $s$ is the size of the subgraph, then the graph is said to be $\sqrt{n}$ - separable. For such matrices the elimination forest will have a height of $\sqrt{n \log n}$ and the running time is
$O(\sqrt{n \log^2 n})$. For large enough $n$, $\sqrt{n \log^2 n}$ is significantly less than $n$, and so this algorithm is faster than the straightforward one, for these problems.

We then modify the algorithm so that it only takes polylog time, but takes some advantage of the sparsity of $A$ in the number of processors used. The time for this new algorithm is $O(\log n \log^2 h)$, and it uses no more than $\min\{ na + .5, Wha + .5 \}$ processors, where $W$ is the number of leaves in the elimination tree.

We also show how to do the forward solve, $Ly = b$, and the back solve, $LTx = y$, in $O(h \log n)$ time with $m$ processors, using the elimination forest. We then show how to speed up these algorithms, so that they will only take $O(\log n \log^2 h)$ time using $\min\{ na, Wha \}$ processors.

The number of processors used by the slower versions of the algorithms is of the same order as the space needed. This property is desirable for two reasons. First, in the future, increases in memory will probably be accompanied by increases in number of processors, because they are both made of the same technology. Therefore $P = \Theta(S)$ is probably a good assumption under which to design and analyze parallel algorithms. Second, since large matrix problems are generally limited by space as much as by time, there is not reason to expect that it will ever be economical to have more processors than space; thus it seems unlikely, for example, that large dense matrices will ever be multiplied using more than $n^2$ processors.

The parallel machine model we use is the PRAM-CRCW. In it concurrent reads and writes are allowed. If two or more processors try to write at the same time, an arbitrary one succeeds.

The rest of the paper is organized as follows. In section 2 we discuss sequential Cholesky factorization. Section 3 introduces elimination forests. Having covered those preliminaries, we present the parallel factorization algorithm in section 4 and the polylog time variation of it in section 5. In section 6 we look at the algorithms for solving the triangular systems $Ly = b$ and $LTx = y$. Finally section 7 contains conclusions and directions for future work.

2. Cholesky factorization

Cholesky factorization is one of the more popular sequential algorithms to solve $Ax = b$ if $A = (a_{ij})$ is symmetric positive definite. The algorithm in figure 2.1 is the column version. It computes the columns of $L = (l_{ij})$ one at a time, overwriting the
lower triangle of \( A \), starting with column 1 and ending with column \( n \). The 
subroutines \( \text{cmod} \) and \( \text{cdv} \) are in figure 2.2. The routine \( \text{cmod}(i,j) \) multiplies 
column \( j \) by \( a_{ij} \) and subtracts it from column \( i \); the routine \( \text{cdv}(i) \) divides column \( i \) 
by the square root of its diagonal element.

```
algorithm Cholesky
begin
  for \( i:=1 \) to \( n \) do
    for \( j:=1 \) to \( i-1 \) do
      \text{cmod}(i,j);
    \text{rof}
    \text{cdv}(i);
  \text{rof}
end
```

Figure 2.1

```
algorithm \text{cmod}(i,j)
begin
  for \( k:=i \) to \( n \) do
    \( A(k,i) := A(k,i) - A(k,j) * A(i,j) \)
  \text{rof}
end
```

algorithm \text{cdv}(i)
begin
  \( x := \sqrt{A(i,i)} \);
  for \( k:=i \) to \( n \) do
    \( A(k,i) := A(k,i)/x \)
  \text{rof}
end
```

Figure 2.2

In the case of a sparse \( A \), column \( i \) might not have to be modified by all the 
preceding columns. In fact it only needs those columns \( j \) in which \( a_{ij} \) is nonzero at the 
time of the execution of \( \text{cmod}(i,j) \). In other words, column \( i \) needs column \( j \) iff \( l_{ij} \) is 
nonzero. Thus to take advantage of the sparsity in computing \( \text{cmod} \) we need to know 
the fill that occurs in \( L \).

3. Elimination forests

The elimination forest has emerged as a useful data structure for solving sparse 
linear systems, both sequentially and in parallel ([Sch82], [Liu86a], [Liu86b],
Liu's paper [Liu87] contains a survey of the uses of the elimination forest in sparse factorization.

**Definition:**

Let $A$ be an $n \times n$ symmetric positive definite matrix and let $L$ be its Cholesky factor. The *elimination forest* of $A$ is a directed forest $F=(V, E)$, where $V=\{1, 2, \ldots, n\}$ and $(i, j) \in E$ iff $j = \min\{k| l_{ki} \neq 0 \text{ and } k > i\}$.

Hence, there is an edge from $i$ to $j$ if and only if the first nonzero below the diagonal in column $i$ of $L$ occurs in row $j$. Since there is a nonzero in position $(j, i)$ of $L$, we will need a nonzero multiple of column $i$ to compute column $j$. Thus we need to compute column $i$ before we can compute column $j$. The graph $F$ is a forest because $j$ is uniquely defined for each $i$, and since $j > i$ there are no cycles.

Figure 3.1 shows a Cholesky factor $L$ and its elimination forest $F$. In this example, the forest consists of a single tree. The fill that occurred in $L$ is shown as an $X$ within a square.

The elimination forest shows the dependence between the columns of $A$ in the computation of its Cholesky factorization. Thus it gives an indication of how easy it is to parallelize the computation. However, since it only takes into account the columns and not their density, there are sparse matrices with the same elimination forests as dense ones. For instance, the elimination forest for a tridiagonal matrix is just one long chain, which is the same as the elimination forest for a full matrix. The tridiagonal matrix only has $3n - 2$ nonzeros and it can be factored in $O(\log n)$ time with $n$ processors.
In an earlier paper [GH86], we presented an algorithm that finds the elimination forests in $O(\log^2 n)$ time using $m+n$ processors. We also gave an algorithm to find the fill that occurs in the Cholesky factor. That algorithm uses the elimination forest and runs in $O(\log^2 n)$ time using $m^*+n$ processors.

In what follows we will assume the elimination forest $F$ of matrix $A$ consists of a single tree $T$. Since each tree of an elimination forest corresponds to a connected component in the graph of $A$, we can reorder $A$ to get a block diagonal matrix and can solve for each block independently.

4. The parallel Cholesky factorization algorithm

Looking at the definition of elimination trees, we see that a node $i$ is a leaf iff there are no nonzero offdiagonal elements in row $i$ of $L$. Therefore each column corresponding to a leaf can be computed independently of all the other columns of the matrix. Having computed those columns, we can use them to compute the columns that depend only on columns already computed. Hence our algorithm repeatedly computes the columns that correspond to leaves of the elimination tree and deletes the leaves from the tree. The algorithm is presented more formally in figure 4.1.

```
algorithm ParallelCholesky
begin
    repeat
        for all the leaves, $i$, of $T$ pardo
            cdiv($i$);
            delete leaf $i$;
        rof
        for all the leaves, $i$, of $T$ pardo
            for all $j<i$ s.t. $(i,j)$ is nonzero in $L$ pardo
                cmod($i,j$);
            rof
    rof
    until $T$ contains no vertices
end
```

Figure 4.1

We assume that we have already run the algorithms of [GH86] on $A$ to find its elimination tree $T$ and the fill that will occur in $L$. One processor will be allocated to each nonzero element of $L$; so we need $m^*$ processors. All the processors have access
to $T$, which is in global memory. Thus each processor knows which nodes are leaves and can determine if it should be involved in some operation.

First, for every leaf $j$, we need to do $cdiv(j)$. For each nonzero $(i, j)$ of $L$, the assigned processor knows if $j$ is a leaf in $T$ and if so it divides the number at position $(i, j)$ by the square root of $a_{jj}$. This is done in $O(1)$ time. Then all the leaves of $T$ are deleted, giving $T$ a new set of leaves.

Next, we want to do $cmod$ for all the new leaves. First we need to find out for each leaf which columns it needs to be modified by. That can be done in constant time. Each processor $(i, j)$ determines if column $i$ is a leaf in $T$. If it is then column $i$ has to be modified by a multiple of column $j$ (see figure 4.2). Sequentially this can be done by subtracting a multiple of column $j$ from column $i$, 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 $i$. So for each column $j$, where $l_{ij}$ is nonzero, we begin by multiplying each nonzero by $l_{ij}$. We then sum the resulting row vectors and subtract each element from its corresponding element in column $i$.

```
  j

  i

  x
```

Figure 4.2

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. Thus to be able to use the parallel vector sum algorithm we have to make sure that each processor know which other processors it are 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. The array $AUX$ is also indexed by row number that says where in $AUX$ each row begins. The array can be initialized
in this way anytime after we have computed the fill. We then assign one processor $P_{i,j}$ to each position that will be nonzero in $L$. Each $P_{i,j}$ records $i$, and a pointer to the position in $AUX$ that contains it nonzero value. This is enough information for $P_{i,j}$ to find $j$, and also for it to find out how many nonzeros are before $L_{i,j}$ in row $i$.

Now when $i$ becomes a leaf, it needs to have $\text{cmod}(i, j_1), \text{cmod}(i, j_2), \ldots$ done on it for each nonzero $L_{i,j_p}$ in row $i$. Suppose row $k$ will participate in this $\text{cmod}$. and suppose that $L_{k,j_2}$ is nonzero (see figure 4.3). Then processor $P_{k,j_2}$ gets the value of $L_{i,j_2}$ from $AUX$, and also finds out that $L_{i,j_2}$ is the second nonzero in row $i$. Therefore, $P_{k,j_2}$ knows that its contribution to the update of $L_{k,i}$ is the second value in the sum to be subtracted from $L_{k,i}$. Now the processors involved in the update add their contributions using a parallel vector sum algorithm. That will take no more than $O(\log n)$ time and the only processors we used were the ones associated with the nonzeros in row $k$.

![Figure 4.3](image_url)

Each column is only modified by the columns that are below it in the elimination tree. Hence, as the algorithm always modifies the columns at the current leaves of the elimination tree, no two columns are modified by the same column at the same time. Thus each nonzero is only involved in one $\text{cmod}$ at a time. Also note that the total amount of space used for the computation is $m^*$. The time to do a $\text{cmod}$ is $O(\log n)$ and we only use the processors associated with the nonzero elements involved, so the maximum number of processors used is $m^*$. To determine total time for the whole algorithm, note that we are deleting all the leaves each time through the repeat-until loop and thus decreasing the height of the elimination tree by 1 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 original height of the elimination tree. The number
of processors is \( m^* \), since we only use the processors associated with the nonzero elements of \( L \).

5. Polylog time sparse Cholesky factorization

In this section we will show how to speed up the algorithm from the previous section. The time bound for that algorithm was proportional to the height of the elimination tree. In general we do not know how that relates to \( n \), the size of the matrix. We might prefer to have an algorithm for Cholesky factorization whose running time is polylog in \( n \) and that also takes some advantage of the sparsity of the matrix in the number of processors used. One way of doing just that is to combine the previous algorithm with a polylog algorithm for dense Cholesky factorization that is only used on chains in the elimination tree.

Let a leafchain be a chain of nodes in the elimination tree, starting at a leaf and going up the tree as long as each node only has one child. In figure 3.1 there are 3 leafchains, (2), (1, 3), and (4). In the previous factorization algorithm we started by working on all the leaves simultaneously. In this version we will start working on all the leafchains simultaneously. Each leafchain 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 chain. Thus each leafchain \( C \) forms a submatrix \( H \) of \( A \) (see figure 5.1). The nonzeros in the

![Figure 5.1](image-url)

columns of the submatrix \( K \) correspond to edges between descendants of the chain \( C \)
in the elimination tree and vertices in the chain. Similarly the nonzeros in \( M \) correspond to edges between vertices in \( C \) and vertices that are ancestors of it.

We are going to compute the Cholesky factorization for the columns that correspond to a chain in the elimination tree in \( O(\log^2 k) \) time, where \( k \) is the length of the chain. Since the chains cannot be longer than the height of the elimination tree the time is bounded by \( O(\log^2 h) \). We will start by modifying the submatrix \( H \) by the relevant columns in \( K \) and the submatrix \( M \) by the same columns of \( N \). That can be done in polylog time. Then we will compute the Cholesky factorization of a matrix composed of \( H \) and the nonzero rows of \( M \), using a polylog time algorithm. That will give us the correct Cholesky factorization of \( H \) and \( M \) within \( A \).

When we start considering the chain \( C \), all the columns of \( K \) and \( N \) that are relevant to \( H \) and \( M \), i.e. all the descendants of \( C \) 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 \( i \) of \( H \) and \( M \) will be modified by those columns of \( K \) and \( N \) that have a nonzero in row \( i \) of \( K \). Those columns of \( K \) and \( N \), appropriately multiplied by the entry in row \( i \) of \( K \), will be summed up and then subtracted from column \( i \) of \( H \) and \( M \). That will take \( O(\log h) \) time with no more than \( h^2 \) processors. Since the height of the elimination tree is \( h \) there have to be less than \( h \) columns involved, each with no more than \( h \) nonzeros. We will do this for all the columns of \( H \) and \( M \) simultaneously, so the total number of processors will be no more than \( h^3 \).

Next we will form a new matrix \( R \) consisting of \( H, M^T \) and \( I \), where \( M^T \) is \( M \) with the zero rows omitted and \( I \) is the identity matrix (see figure 5.2). The matrix \( R \) contains no more than \( h \) rows or columns, since the nonzero rows of \( M \) correspond to ancestors of the chain in the elimination tree.

\[
R = \begin{bmatrix}
H & M^T \\
M^T & I
\end{bmatrix}
\]

Figure 5.2

Now when we compute the Cholesky factorization of \( R \), it will give us the correct factorization of the submatrices \( H \) and \( M \) within \( A \). There are two different algorithms for finding the Cholesky factorization in polylog time. They both run in \( O(\log^2 n) \) time using \( n^{a+0.5} \) processors, where \( n \) is the size of the matrix. One of them
is similar to Cramer's Rule for solving linear systems ([Dat85]). The other one uses Schur complements to compute the factorization. We can use either of them to factor $R$.

We now know how to compute the columns corresponding to leafchains in the elimination tree in $O(\log^2 h)$ time with $h^{a+.5}$ processors. If we compute all the leafchains simultaneously we will only have to do that $\log n$ times to reduce the elimination tree to a single node. This follows from Miller and Reif's analysis of their parallel tree contraction scheme [MR85]. Thus the total time for the algorithm is $O(\log n \log^2 h)$. It is more difficult to get a good bound on the number of processors used. We will need no more than $h^{a+.5}$ processors to compute each chain, but we might have to work on many chains at the same time. An upper bound on the number of processors needed is $n^{a+.5}$, which comes out of Lemma 2.5.2 in [Gil80]. A more realistic upper bound is $\min\{n^{a+.5}, W h^{a+.5}\}$, where $W$ is the number of leaves in the elimination forest. This bound is certainly not always tight. Very often $W=O(n)$, so the bound is worse than the bound for a dense matrix!

In the case of a matrix $A$, whose graph $G(A)$ is a grid graph, we can apply nested dissection and get an elimination tree of height $\sqrt{n \log n}$. Because of the special structure of that elimination tree, $A$ can be factorized, with the above algorithm, in time $O(\log^3 n)$ using $n^{a/2}$ processors.

The motivation for using a dense Cholesky factorization algorithm for the leafchains is that for each nonzero $(i, j)$ of $H$, the rest of the row, from $(i, j)$ to the diagonal element $(i, i)$ will become nonzero because of fillin. Similarly the rows of $M$ will get filled in from the first nonzero of 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.

The method of forming the matrix $R$ and factoring it is quite closely related to the multifrontal method of Duff and Reid [DR83].

6. Forward solve and back solve

The two previous sections showed how to find the Cholesky factor, $L$, of $A$ in $O(h \log n)$ and $O(\log n \log^2 h)$ time respectively, but to solve the system $Ax=b$, we still have to solve two triangular systems, $Ly=b$ and $LTx=y$. Since $L$'s sparsity structure
is to some degree reflected in the elimination tree we should be able to use it to solve these two systems.

As we noted in section 4, a node $i$ is a leaf in the elimination tree iff there are no nonzero offdiagonal elements in row $i$ of $L$. Hence, in the forward solve, we can solve for a variable corresponding to a leaf node by just dividing the corresponding element of the right hand side by the diagonal element. Having computed the values for all such variables, we can substitute them for the variables in the other equations, and for each row of $L$, sum up the result and subtract it from the right hand side. We then delete the leaves of $T$. We thus repeatedly look at the new leaves of $T$ and solve for the corresponding variables.

For an example of how the algorithm works let us look at the system in figure 6.1. It has the elimination tree in figure 3.1. The algorithm will first find the values of

$$
\begin{bmatrix}
2 & 0 & 5 \\
0 & 2 & 0 \\
0 & 0 & 1 \\
0 & 3 & 1 \\
2 & 0 & 1 & 2 & 1 & 2
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
y_5 \\
y_6 
\end{bmatrix}
= 
\begin{bmatrix}
2 \\
10 \\
8 \\
4 \\
14 \\
30 
\end{bmatrix}
$$

Figure 6.1

$y_1, y_2, \text{and } y_4, \text{which are 1, 2, and 4 respectively. Then these values are substituted for the corresponding variables in the rows remaining; the resulting values are summed up for each row and subtracted from the right hand side. }$ The resulting system and its elimination tree are in figure 6.2. Now we can find the value for $y_3$,

$$
\begin{bmatrix}
2 & 1 & 1 \\
1 & 1 & 2
\end{bmatrix}
\begin{bmatrix}
y_3 \\
y_5 \\
y_6 
\end{bmatrix}
= 
\begin{bmatrix}
6 \\
8 \\
20
\end{bmatrix}
$$

Figure 6.2

and then in the next round the value for $y_5$ and finally the value for $y_6$.
This was the way forward solve works. For back solve we start at the root of the elimination tree and work our way down. At first we can only solve for the one variable that corresponds to the root. We then substitute the value for that variable in the other equations and subtract the result from the right hand side. This will correspond to deleting the root of the elimination tree. Then, in parallel, we repeat this process with each of the new trees.

We can continue with the example from figure 6.1 and now do the back solve, \( LTx = y \). The system is given in figure 6.3. The algorithm will start by solving for \( x_6 \)

\[
\begin{bmatrix}
2 & 0 & 2 & 0 & 0 & 2 \\
5 & 0 & 0 & 3 & 0 \\
2 & 0 & 1 & 1 \\
1 & 0 & 2 \\
1 & 1 \\
2
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6
\end{bmatrix}
= 
\begin{bmatrix}
1 \\
2 \\
3 \\
4 \\
5 \\
6
\end{bmatrix}
\]

Figure 6.3

and deleting the root of the elimination tree. Then it will substitute the value for that variable in the other rows and subtract the resulting values from the right hand side. The new system and the new elimination forest are in figure 6.4. In the new system the algorithm can obtain the values of \( x_4 \) and \( x_5 \) simultaneously, deleting their corresponding nodes in the elimination forest. In the next stage the values of \( x_2 \) and \( x_3 \) are calculated and finally the value of \( x_1 \).

Both these algorithms run in \( O(h \log n) \) time, using \( m^* \) processors. Thus the running times are proportional to the height of the elimination tree. We can speed
up these two algorithms in a way similar to that we used in section 5 for Cholesky factorization. We now use a polylog time algorithm for triangular systems on chains in the elimination tree. We will show how to do that for forward solve; back solve is similar.

In the previous forward solve algorithm we started by working on all the leaves simultaneously. In this version we will, as we did in section 5, start working on all the leafchains simultaneously. Each leafchain 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 leafchain $C$ forms a submatrix $H$ of $L$, which has a full first band below the diagonal (see figure 6.5).

![Figure 6.5](image)

There are a few parallel algorithms for solving dense $n \times n$ triangular systems in $O(\log^2 n)$ time with $n^a$ processors ([Hel74], [CK75], [BM75]). We can use any of them to solve the system consisting of the submatrix $H$, for each leafchain. Since each leafchain can not be longer than $h$, the time to solve each chain is $O(\log^2 h)$. The total time for the whole algorithm will be $O(\log n \log^2 h)$, since we will only have to do $\log n$ stages. As we will need no more than $h^a$ processors to solve each chain, we get an upper bound of $\min\{ (n - h)^a + h^a, Wh^a \}$, where $W$ is the number of leaves in the elimination tree.
7. Conclusion

We have presented parallel algorithms to find the Cholesky factor of a sparse matrix and solve the resulting triangular systems. The algorithms use the elimination forest of the matrix to exploit its sparsity. The time bound for the algorithms is proportional to the height of the elimination forest and the number of processors used equals the number of nonzero elements in the resulting Cholesky factor.

We also managed to speed up both the Cholesky factorization and the triangular solvers by using a version of parallel tree contraction. Our bounds on the processor complexity of the contraction algorithms are not as tight as possible; we are working on improving the analysis. The contraction algorithms are most efficient when the submatrices corresponding to leaf chains are full. This is the case when each leaf chain corresponds to a minimal separator in the graph corresponding to its subtree.

The algorithms presented in this paper are particularly attractive for implementation on massively parallel architectures like the Connection Machine or the AMT DAP.

There are very sparse matrices with tall and skinny elimination trees, for instance tridiagonal matrices. Our algorithm will take longer to factorize them than is necessary. Perhaps a heuristic ordering algorithm can be used to reorder the rows and columns of the matrix so that the height of its elimination tree is as small as possible. Coupled with such an ordering algorithm, our algorithm might find the Cholesky factor of a sparse matrix in close to optimal time. Liu and Mirzaian [LM87] have presented a sequential algorithm that will, by permuting the matrix, find an elimination tree of minimum height among all the matrices that have the same fill as the original one.

Acknowledgements

We would like to thank Earl Zmijewski for his careful reading of the manuscript.
References


