Reordering sparse matrices for parallel elimination*

Joseph W.H. LIU

Department of Computer Science, York University, North York, Ontario, Canada M3J 1P3

Received February 1987
Revised July 1988

Abstract. We consider the problem of finding equivalent reorderings of a sparse matrix so that the reordered matrix is suitable for parallel Gaussian elimination. The elimination tree structure is used as our parallel model. We show that the reordering scheme by Jess and Kees generates an elimination tree with minimum height among all such trees from the class of equivalent reorderings. A new height-reducing algorithm based on elimination tree rotation is also introduced. Experimental results are provided to compare these two approaches. The new reordering algorithm using rotation is shown to produce trees with minimum or near-minimum height. Yet, it requires significantly less reordering time.

Keywords. Sparse matrix, parallel elimination, reordering, elimination tree, height, rotation.

1. Introduction

It has been recognized by many researchers that a promising area for advances in sparse matrix computation is the use of parallel machines and algorithms. In the direct solution of sparse symmetric positive definite linear systems, this potential can be realized by the use of parallel eliminations. Calahan [4] in 1973 first identified the condition for 'parallel pivots', that is, pivots that can be eliminated simultaneously. Since then, parallel pivoting strategies have been considered by Alaghband and Jordan [2], Betancourt [3], Huang and Wing [12], Jess and Kees [13], Peters [19], and others.

Let $A$ be a given $n \times n$ sparse symmetric positive definite matrix. In this paper, we consider the problem of finding an equivalent reordering of the given sparse matrix so that the reordered matrix can be factored effectively in parallel. This approach assumes that a good fill-reducing ordering $P$ of the matrix $A$ has already been determined (such as minimum degree or nested dissection ordering [9]). Our objective is to find a reordering $\tilde{P}$ of $A$, that preserves the filled graph of $PAP^T$ and at the same time is appropriate for parallel elimination. In this way, the final reordering $\tilde{P}$ is guaranteed to exhibit the same quality as $P$ in terms of fill reduction. Indeed, if $F$ is the filled matrix of $PAP^T$ (that is, $F = L + L^T$, where $L$ is the Cholesky factor of $PAP^T$), it has a chordal graph structure (see [20]). We can then view our problem as


determining a perfect elimination ordering (one with no fill) for a chordal graph, which also exploits parallel elimination.

This approach is used by Jess and Kees [13]. They also provide a strategy to determine such a reordering \( \hat{P} \). The first part of this paper is to examine this reordering scheme by Jess and Kees based on the elimination tree model [15] for parallel elimination. We show that the Jess and Kees’ scheme generates an elimination tree with minimum height among all such trees from the class of equivalent reorderings. In a related paper [18], Liu and Mirzaian give an efficient algorithm for this reordering strategy, which is linear with respect to the number of nonzeros in the filled graph.

The second part of this paper is to consider a new reordering strategy. Our approach is to form the elimination tree of the matrix \( PAP^T \), and then attempt to reduce its height by restructuring the tree without introducing additional fill. The restructuring will correspond to an equivalent reordering of the given matrix. The restructuring technique used is based on elimination tree rotations, introduced by the author in [16]. Each rotation operation is similar in nature to the standard binary tree rotation in the study of height-balanced or AVL-trees, but our constraint here is to preserve the set of fill. An efficient height-reducing algorithm is formulated using the structure of the elimination tree and some relevant information associated with the sparse factorization process. Experimental results are provided to demonstrate the effectiveness of this new scheme in terms of decreased running time and height-reducing capability.

In this paper, the readers are assumed to be familiar with the graph-theoretic terminology associated with sparse matrix computation: adjacent set, degree, fill, ordering, elimination graph, clique, and other related concepts. All the necessary material can be found in [9]. Moreover, notions related to tree structures are also assumed: parent/child nodes, ancestor/descendant nodes, root, path, subtree, leaf, height, and depth. They can be found in [11].

An outline of this paper is as follows. In Section 2, we review the elimination tree structure as a model for parallel Gaussian elimination. We discuss the relevance of the height of elimination trees in this context. A desirable elimination tree for parallel elimination is one with minimum height (or some related quantities). We also establish some preliminary result relating elimination tree height with paths in the graph associated with the given sparse matrix.

Section 3 contains an overview of reordering approaches. This is to justify our choice: namely to determine an equivalent reordering [16] (one that preserves the filled graph) suitable for parallel elimination. The basic parallel pivoting algorithm by Jess and Kees [13] on chordal graphs is reviewed. Then we establish the important property that the resulting reordering gives an elimination tree with minimum height.

Height reduction by elimination tree rotations is the subject of Section 4. After introducing the necessary terminology, we provide a simple sufficient condition for a rotation that reduces height. The overall reordering scheme is then described in detail. Its complexity issue is also addressed.

Section 5 contains experimental runs on some large practical sparse matrix problems from the Harwell-Boeing collection [7]. It includes results on both the minimum-height reordering algorithm as described in [18] and the height-reducing scheme by rotations. Substantial savings in reordering time by the rotation scheme are reported. Indeed, it often requires from two to six times less reordering time than the minimum-height scheme. Yet, the rotation scheme produces elimination trees with minimum or near-minimum height.

In Section 6, we consider the use of parallel completion time as the criterion for desirable elimination trees. This is shown to be a generalization of the notion of height. Section 7 contains our concluding remarks.
2. Elimination tree and parallel elimination

2.1. Elimination tree as a model for parallel elimination

Let \( A \) be a given \( n \times n \) sparse symmetric positive definite matrix, and \( F = L + L^T \) be its filled matrix with \( L \) being its Cholesky factor. Let \( G(A) \) and \( G(F) \) be its symmetric graph and its filled graph respectively. We assume that the matrix \( A \) has already been ordered by some fill-reducing ordering, such as the minimum degree or nested dissection ordering [9]. Let 
\[ x_1, x_2, \ldots, x_n \]
be the corresponding node sequence in \( G(A) \).

The elimination tree \( T(A) \) of \( A \) is defined as follows. It has the same node set as \( G(A) \), and the node \( x_i \) is the parent of \( x_j \) (with \( i > j \)) if and only if
\[ i = \min \{ r \mid I_{rj} \neq 0 \}, \]
where \( I_{rj} \) is the entry in the \( r \)th row and \( j \)th column of the Cholesky factor matrix \( L \). We further assume that the matrix \( A \) is irreducible so that this structure is indeed a tree rooted at the node \( x_n \). (The readers can refer to Fig. 3 for an example of an elimination tree.)

In [15], the elimination tree structure is used as a computational task model to study parallel sparse Cholesky factorization. It is shown to be an appropriate large-grained model for various forms of factorization scheme: column-Cholesky, row-Cholesky and submatrix-Cholesky. In each case, the node \( x_i \) in the elimination tree corresponds to a specific task associated with column/row \( i \) of the matrix in the factorization. The tree edges in the elimination tree provide precisely the precedence relation among these tasks. Therefore, the tasks associated with leaves in the tree can be started in parallel. Moreover, the connection between tasks and nodes implies that the determining of an appropriate elimination tree for parallel elimination corresponds to finding a matrix or graph reordering.

It is also important to realize that this elimination tree model is suitable to study parallel elimination on both local-memory and shared-memory parallel machine architectures. For local-memory machines, the tree provides a basis to study the task-to-processor assignment problem, which is important for achieving load-balancing [6,8] and for reducing communication cost [10]. For shared-memory systems, the task scheduling problem can be dealt with as a reordering of nodes in the elimination tree [15].

There are potential parallelism on operations within each node of the elimination tree. Exploiting them will give parallel factorization algorithms with finer granularity. In [15], a general overview of various models for parallel factorization is given, including those taking advantage of some or all parallelism within each node. Of course, one would like to exploit all possible parallelism. However, given the current number of available parallel processors (order of hundreds) and the size of large sparse systems to be solved (order of ten-thousands), large-grained parallel algorithms definitely have a practical role to play. Furthermore, a better understanding of such algorithms will aid in the study of finer-grained schemes. In this paper, we assume the use of large-grained parallelism in the study of parallel elimination.

2.2. Height of elimination tree

Having chosen the elimination tree structure as our model, we can consider various criteria for the best elimination tree for parallel elimination. The best tree depends on many factors, including:
(a) parallel architecture: local- or shared-memory,
(b) the number of parallel processors available,
(c) the algorithmic form of Cholesky factorization: column-, row-, or submatrix-Cholesky,
(d) the task assignment/scheduling strategy,
(e) the computation-to-communication cost ratio (for local-memory machines).

We contend to choose one which is generally good for most cases, but may not be the best possible for the given circumstances.

One logical choice would be the parallel completion time. Intuitively, it is the time required to complete the whole factorization exploiting all parallelism as provided by the precedence relation in the elimination tree structure. (It can be viewed as the critical path length of the elimination tree using the completion time of each node as the cost.) It is a relevant quantity to minimize for parallel pivoting. We shall formally define this notion in Section 6.

A closely-related concept is the height of the elimination tree. Indeed, if the tasks associated with the nodes are considered as tasks of equal duration, the parallel completion time is simply

$$\text{unit task time} \cdot (\text{height} + 1).$$

In most part of this paper, we shall use the height of the elimination tree as our criterion. Minimizing the height has a significant impact on the reduction of parallel completion time, even if the tasks are of unequal duration. The height notion is relatively simple, and it allows a clearer exposition of ideas and algorithms.

Our objective is to determine a reordering of a given sparse matrix so that the resulting elimination tree has small height. We emphasize that the height is used as a simple approximation to the parallel completion time and that large-grained parallelism is assumed in the numerical factorization. Exploitation of finer granularity in parallel elimination would require the use of a different and probably more complicated task model. Reducing the parallel completion time becomes even more difficult.

We now formally define the concept of the height of a tree [23]. We first introduce a function $ht_T(v)$ for each node $v$ in the tree $T$. If $v$ is a leaf, $ht_T(v) = 0$. If $v$ is an interior node, then

$$ht_T(v) = 1 + \max\{ ht_T(w) \mid w \text{ is a child of } v \text{ in } T \}.$$

The height of the tree is then defined as

$$\text{height}(T) = \begin{cases} ht_T(\text{root of } T) & \text{if the tree } T \text{ is nonempty,} \\ -1 & \text{otherwise.} \end{cases}$$

We are interested in reordering whose associated elimination tree has minimum or small height.

Consider a path $(x_{j_0}, x_{j_1}, \ldots, x_{j_r})$ in the filled graph $G(F)$. It is said to be a monotone path if $j_1 < j_2 < \cdots < j_r$.

**Lemma 2.1.** (Schreiber [22]). If $\{x_i, x_j\}$ is an edge in the filled graph $G(F)$ $(i > j)$, then the node $x_i$ is an ancestor node of $x_j$ in the tree $T(A)$.

**Theorem 2.2.** The height of the elimination tree $T(A)$ is the length of a longest monotone path in $G(F)$.

**Proof.** For any leaf node of $T(A)$, the path from it to the root is a monotone path in $G(F)$. Therefore, the height of $T(A)$ must be no greater than the length of a longest monotone path. On the other hand, consider any monotone path in $G(F)$: $(x_{i_0}, x_{i_1}, \ldots, x_{i_r})$. Since each $\{x_{i_j}, x_{i_{j+1}}\}$ is an edge in the filled graph $G(F)$, by Lemma 2.1, $x_{i_{j+1}}$ must be an ancestor of $x_{i_j}$ in $T(A)$. This means the length of this monotone path must be less than or equal to the length of the path from $x_{i_0}$ to the root in the elimination tree $T(A)$. The result then follows. □
3. Minimum-height elimination tree for chordal graphs

3.1. Why chordal graphs

In this section, we consider the parallel elimination of nodes in chordal or triangulated graphs. (A graph is chordal if every cycle of length 4 or more has a chord; or equivalently, if it has a perfect elimination ordering [11,20].) We first give the motivation to study this problem.

For a given sparse matrix \( A \), there are generally two approaches to finding a good ordering for parallel elimination. The first one attempts to find an ordering that reduces the parallel factorization completion time and preserves sparsity, based on the structure of \( A \) directly. The recent work by Alaghband and Jordan [2] and by Betancourt [3] use this approach. They face with the problem of dealing with the competing issues of parallelism and fills at the same time. Indeed, the algorithm introduced in [2] has a complexity exponential in the order of the matrix. Peters [19] uses a variant form of the minimum degree algorithm to identify parallel pivots whose degrees differ from the minimum degree by no more than a threshold value. But he does not provide any experiment on the performance of his scheme.

The second approach first determines a fill-reducing ordering \( P \) (such as the minimum degree ordering) and then finds an equivalent reordering \( \hat{P} \) suitable for parallel elimination (that is, \( \hat{P} \) has the same set of fills as \( P \)). The fill-in and parallelism issues are now handled in separate phases. The work of Jess and Kees [13] can be viewed as using this modular approach.

The success of this approach relies on the observation that a filled graph from a fill-reducing ordering has good potential for parallel elimination. This is generally true for most practical sparse problems. (A notable exception is a chain of nodes). Our experiments in Section 5 will attest to this observation.

In this paper, we shall use this modular approach of separating fill-reducing ordering and reordering for parallel elimination. Since filled graphs are chordal, our problem can be formulated as finding a perfect elimination ordering (with no fill) for a chordal graph so that its associated elimination tree has minimum or near-minimum height.

3.2. Parallel elimination algorithm by Jess and Kees

In the remainder of this section, we shall assume that the graph \( G(A) \) of the given sparse matrix \( A \) is chordal. We shall use \( G \) and \( G(A) \) interchangeably to refer to this graph of \( A \). Before we review the parallel elimination algorithm by Jess and Kees [13], we introduce some terminology.

Two nodes are said to be independent if they are not adjacent. A node is said to have no or zero deficiency if its adjacent set is a clique [20]. In the literature, such node is also referred to as simplicial [11]. It is well known that a chordal graph always has one node with no deficiency. Furthermore, if the graph is not a clique, there are at least two such independent nodes [11, p. 83]. This already provides some possibility for parallel elimination without introducing fills. In [13], Jess and Kees make use of this observation to determine a perfect elimination reordering of a given chordal graph suitable for parallel elimination. We quote the following property from [13] on nodes with no deficiency in a given chordal graph \( G \).

**Proposition 3.1** (Jess and Kees [13]). *The subgraph of nodes with no deficiency in \( G \) consists of disconnected cliques.*

For parallel elimination, we can therefore select one node from each clique subset to form a set \( R \) of representatives. Since the nodes come from disconnected cliques, they can be eliminated in parallel. Moreover, it forms a maximum set of independent nodes with no
deficiency. This observation forms the basis for the algorithm by Jess and Kees [13, procedure “e-tree”, p. 233] to determine a perfect sequence exploiting parallelism in elimination for a given chordal graph. We present it in our terminology as follows.

Algorithm 3.1. Parallel_Elimination (G)

begin

\[ G_0 \leftarrow G; \]
\[ i \leftarrow 0; \]
while \( G_i \neq \emptyset \) do

begin

\[ R_i \leftarrow \text{a maximum independent subset of non-deficient nodes}; \]

order nodes in \( R_i \) next;

\[ G_{i+1} \leftarrow G_i - R_i; \{ \text{eliminate the nodes of } R_i \text{ from } G_i \} \]
\[ i \leftarrow i + 1; \]

end;

end.

The index \( i \) serves as a step count for the ‘while’ loop. On applying Algorithm 3.1, we obtain a sequence of node subsets:

\[ R_0, R_1, \ldots, R_m, \]

where each \( R_i \) is a maximum independent set of nodes with no deficiency in the chordal subgraph \( G_i \) at step \( i \). Note that if \( G_i \) is a clique, we can simply exit the ‘while’ loop and order the remaining nodes of \( G_i \) (in any order). Let \( P \) be a reordering of the nodes obtained from this algorithm. \( P \) may not be unique due to the different choices of representative set at each step. The following results is quoted from [13].

Theorem 3.2 (Jess and Kees [13]). The height of the elimination tree \( T(\bar{P}A\bar{P}^T) \) is given by \( m \), and is independent of the choice of maximum independent non-deficient subsets in Algorithm 3.1.

The algorithm can be best illustrated by an example. Consider the undirected graph in Fig. 1. The graph is chordal, since \( a, c, f, h, b, d, g, e \) is a perfect elimination sequence.

On applying Algorithm 3.1 to this example, the following shows two possible choices of the representative sequence:

<table>
<thead>
<tr>
<th>Step ( i )</th>
<th>Selected ( R_i )</th>
<th>Alternative ( R_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{a, c, f, h}</td>
<td>{b, c, f, h}</td>
</tr>
<tr>
<td>1</td>
<td>{b, d}</td>
<td>{a, g}</td>
</tr>
<tr>
<td>2</td>
<td>{g}</td>
<td>{d}</td>
</tr>
<tr>
<td>3</td>
<td>{e}</td>
<td>{e}</td>
</tr>
</tbody>
</table>

Fig. 1. An undirected chordal graph with 8 nodes.
Figure 2 shows the corresponding elimination trees from these two sequences of representatives. Although the underlying tree structures are quite different, their heights are both 3.

The complexity of this algorithm depends crucially on the no deficiency test to determine the subset $R_i$. Jess and Kees claim that “this test is also part of the pivoting procedure that has to precede the application” of this algorithm so that it has a polynomial complexity. However, generating a perfect ordering for a chordal graph may not always require this test on no deficiency (see, for example, [21]). It is also not necessary to use this test in the symbolic factorization phase for space allocation.

Recently, a linear reordering algorithm [18] is given for this basic parallel pivoting scheme of Algorithm 3.1. The time complexity is shown to be proportional to the number of nonzeros in the given chordal graph. Experimental results on this algorithm are included in Section 5 for comparison.

### 3.3. Minimum height property

The term elimination tree is used differently by Jess and Kees in [13]. They use it to refer to the tree structure obtained from Algorithm 3.1. However, in this paper, we follow [17] to use this term for a wider class of trees as defined in Section 2.1. Indeed, in this subsection, we shall show that the elimination tree from Algorithm 3.1 is of minimum height among all elimination trees associated with perfect elimination orderings of the given chordal graph.

As before, let $\beta$ be an ordering from Algorithm 3.1 on a given chordal graph $G(A)$. To establish the minimum height property of the elimination tree $T(\beta A \beta^T)$, we need some lemmas.

**Lemma 3.3.** For any perfect elimination ordering $P$ of $A$, every leaf node in the elimination tree $T(P A P^T)$ has no deficiency in $G(A)$.

**Lemma 3.4.** Let $v$ be a node of no deficiency in the chordal graph $G(A)$, Then the minimum height of elimination trees associated with perfect orderings on $G(A) - \{v\}$ is less than or equal to that in $G(A)$.

**Proof.** Let $P$ be a perfect ordering on $G(A)$ such that the elimination tree $T(P A P^T)$ has minimum height. Define the induced ordering $P'$ on $G(A) - \{v\}$ to be $P$ restricted on $G(A) - \{v\}$. Since $v$ is of no deficiency in $G(A)$, this induced ordering $P'$ is also a perfect elimination ordering on $G(A) - \{v\}$. Consider any monotone path on $G(A) - \{v\}$ using this ordering $P'$. This sequence must form a path in $G(A)$ and it is also monotone with respect to the ordering $P$. It then follows from Theorem 2.2 that the length of a longest monotone path in $G(A) - \{v\}$ with ordering $P'$ is no greater than the height of $T(P A P^T)$. Therefore the minimum height of elimination tree associated with perfect orderings on $G(A) - \{v\}$ must be less than or equal to the height of $T(P A P^T)$. □
Theorem 3.5. If $\tilde{P}$ is an ordering from Algorithm 3.1, then the height of $T(\tilde{P}A\tilde{P}^T)$ is minimum among all elimination trees associated with perfect orderings on $G(A)$.

Proof. We prove the result by induction on $n$, the number of nodes in the chordal graph. The result is obviously true for $n = 1$ and 2. Assume that it holds true for all chordal graphs of size less than $n$. Consider a chordal graph $G(A)$ of $n$ nodes. Let $P$ be a perfect elimination ordering on $G(A)$ where the elimination tree $T(PA^P)$ has minimum height. Consider the removal of $V$, the set of leaf nodes in the tree $T(PA^P)$, from the tree. Denote the remaining tree by $T(PA^P) - V$. By Lemma 3.3, each node in $V$ has no deficiency in $G(A)$. Furthermore,

$$height(T(PA^P) - V) = height(T(PA^P)) - 1.$$ 

Consider the application of Algorithm 3.1 to the graph $G(A)$ so that the first set of representative nodes $R_0$ contains $V$. This is possible since $R_0$ is a maximum set of nodes with no deficiency and can be chosen to contain $V$. Let $\tilde{P}$ be the resulting perfect reordering from Algorithm 3.1. Note that by Theorem 3.2, the height of this tree $T(\tilde{P}A\tilde{P}^T)$ is independent of the choice of representatives. The graph $G(A) - R_0$ is chordal and with less than $n$ nodes. By inductive assumption, Algorithm 3.1 gives a perfect ordering on $G(A) - R_0$ with minimum height elimination tree. This height is given by

$$height(T(\tilde{P}A\tilde{P}^T)) - 1.$$ 

Moreover, by Lemma 3.4, this height must be no greater than the minimum height on the subgraph $G(A) - V$, which is less than or equal to

$$height(T(PA^P) - V).$$

Combining, we have

$$height(T(\tilde{P}A\tilde{P}^T)) - 1 \leq height(T(PA^P) - V) = height(T(PA^P)) - 1.$$ 

By the choice of $P$, we have proved

$$height(T(\tilde{P}A\tilde{P}^T)) = height(T(PA^P)).$$

\[\square\]

3.4. Application to matrix reordering

Algorithm 3.1 finds a perfect reordering for a chordal graph so that the resulting elimination tree is of minimum height. The approach can be adapted to determine reorderings on a given sparse matrix $A$. We can view the ordering and reordering phases as

$$G(A) \rightarrow G(PA^P) \rightarrow G(F) \rightarrow G(\tilde{P}A\tilde{P}^T),$$

where $F$ is the filled matrix of $PA^T$.

It is interesting to note that if $\tilde{F}$ is the filled matrix of $\tilde{P}A\tilde{P}^T$, the filled graph $G(\tilde{F})$ is in general a subgraph of $G(F)$. This is because the reordering $\tilde{P}$ is a perfect ordering for the filled graph $G(F)$, but some of the filled edges in $G(F)$ may not be necessary when we consider $\tilde{P}$ as a reordering of the original matrix $A$. The reordering $\tilde{P}$ obtained by Algorithm 3.1 on the filled graph $G(F)$ can sometimes be a better ordering than $P$ on $A$. In other words, $P$ and $\tilde{P}$ are equivalent orderings on the filled matrix $F$, but they may not be equivalent on $A$.

To illustrate this, consider the matrix $A$ associated with the graph in Fig. 1 except that there is no edge between $b$ and $e$. Using the following ordering $P$:

$$a, c, f, h, b, d, g, e$$
on the matrix $A$, we obtain a filled graph that will include $\{b, e\}$ as a filled edge, created during the elimination of the node $a$. In other words, the filled graph of $PAP^T$ is now the same as the graph in Fig. 1. One possible reordering produced by the application of Algorithm 3.1 to this filled graph is $\tilde{P}$

$$ b, c, f, h, a, d, g, e. $$

The filled graph of $\tilde{P}A\tilde{P}^T$ does not include the edge $\{b, e\}$. In terms of fill reduction, $\tilde{P}$ is therefore a better ordering than $P$ on the matrix $A$. Some numerical experiments in Section 5 will further illustrate this point.

It is also important to note that the ordering $\tilde{P}$ from Algorithm 3.1 gives an elimination tree of minimum height among all perfect reorderings of the filled matrix $F$. It is by no means minimum among all reorderings for $F$ or for the original matrix $A$.

4. Height reduction by elimination tree transformations

4.1. Restructuring by elimination tree rotations

The direct use of Algorithm 3.1 (Parallel Elimination) to determine reorderings of a sparse matrix $A$ requires the formation of the filled graph $G(F)$ either implicitly or explicitly. It is the purpose of this section to consider efficient determination of an equivalent reordering of $G(A)$ working only on the structure of $PAP^T$. Although the resulting tree may not always have minimum height (with respect to the filled graph), it usually does. The approach uses the following alternative view:

(a) Determine a fill-reducing ordering $P$ of $G(A)$;
(b) Form the elimination tree $T(PAP^T)$;
(c) Transform this tree $T(PAP^T)$ to one with smaller height, and record the corresponding equivalent reordering $\tilde{P}$.

The basic tree restructuring tool used here is elimination tree rotation introduced by the author in [16]. We briefly define the necessary notions here; for the details, the readers are referred to that paper. Let $T(A)$ be a given elimination tree, and if $A$ is clear from context, we use $T$ instead. For any node $v$ in the tree, we use the notation $T[v]$ to denote the subtree of $T$ rooted at the node $v$ (that is, the node $v$ together with its descendants in $T$). Also, we define $\text{Ancestor}_T(v)$ to be the set of (proper) ancestors of the node $v$ in $T$.

A node $y$ in the tree $T$ is said to be eligible for rotation if $|\text{Adj}_G(A)(T[y])| \neq |\text{Ancestor}_T(y)|$.

and $\text{Adj}_G(A)(T[v]) = \text{Ancestor}_T(v)$ for every ancestor node $v$ of $y$. An elimination tree rotation at such an eligible node $y$ is an equivalent reordering of the graph $G(A)$ such that the nodes in $\text{Adj}_G(A)(T[y])$ are labeled last, while preserving the relative order of the remaining nodes.

It is appropriate to point out that the quantity $|\text{Adj}_G(A)(T[y])|$ is the same as the number of off-diagonal nonzeros in the column of the Cholesky factor associated with the node $y$. When $G(A)$ is clear from context, we shall omit the subscript and simply use $\text{Adj}(T[y])$ for $\text{Adj}_G(A)(T[y])$.

An example will be useful here to illustrate the notion of rotation. Consider the chordal graph with a perfect elimination ordering in Fig. 3 (same graph as Fig. 1). Its corresponding elimination tree is also given.
There are two nodes eligible for rotation in this elimination tree: nodes $c$ and $h$. Indeed, we have

$$\text{Adj}(T[c]) = \{d\} \neq \{d, f, g\} = \text{Ancestor}_T(c),$$
$$\text{Adj}(T[h]) = \{d, g\} \neq \{d, f, g\} = \text{Ancestor}_T(h).$$

A rotation corresponding to each one of them is presented in Fig. 4. The new reordering for each rotation is also displayed in the figure. Note that the tree structure is changed, but the filled graph remains the same.

4.2. Height reducing rotations

We now consider the impact of elimination tree rotations on the height of the resulting tree. Our objective here is to introduce a simple sufficient condition for an elimination tree rotation to reduce the height of the tree. As in Section 2, for any node $v$ in the tree $T$, define $ht_T(v)$ as the height of the subtree $T[v]$. Following [23], we define $\text{depth}_T(v)$ to be the length of the path from the node $v$ to the root of the tree $T$. In other words,

$$\text{depth}_T(v) = |\text{Ancestor}_T(v)|.$$
To facilitate our discussion, we introduce some new terminology. For any node \( v \), define the value \( \overline{h_T(v)} \) as follows. If every subtree of \( T \) intersects \( T[v] \), then \( \overline{h_T(v)} = -1 \); otherwise,

\[
\overline{h_T(v)} = \max \{ h_T(w) \mid T[w] \cap T[v] = \emptyset \}.
\]

We may appropriately call a subtree \( T[w] \) outside of \( T[v] \) if \( T[w] \cap T[v] = \emptyset \). Then, \( \overline{h_T(v)} \) is the maximum height of subtrees outside of \( T[v] \). For example, in the elimination tree \( T \) of Fig. 3, we have

\[
\overline{h_T(f)} = \overline{h_T(g)} = \overline{h_T(d)} = -1, \quad \overline{h_T(c)} = 3,
\]

\[
\overline{h_T(h)} = \overline{h_T(e)} = \overline{h_T(b)} = \overline{h_T(a)} = 0.
\]

Let \( y \) be a node eligible for rotation in the given elimination tree \( T \). Now consider an elimination tree rotation of \( T \) at the node \( y \) to give a new elimination tree \( \tilde{T} \). We first provide a bound on the height of the new tree using quantities from the original tree \( T \).

**Theorem 4.1.**

\[
\text{height}(\tilde{T}) \leq \max \{ h_T(y) + |\text{Adj}(T[y])|, \overline{h_T(y)} + \text{depth}_T(y) \}.
\]

**Proof.** Consider a longest path from the root to a leaf in the new tree \( \tilde{T} \), that is, its length is \( \text{height}(\tilde{T}) \). We treat the following cases separately.

**Case 1:** the node \( y \) is on this path. For the node \( y \), since the rotation at \( y \) does not change the structure of the subtree at \( y \), we have

\[
h_{\tilde{T}}(y) = h_T(y).
\]

Moreover, after the rotation, the set of ancestors of the node \( y \) is given by \( \text{Adj}(T[y]) \) so that

\[
\text{depth}_{\tilde{T}}(y) = |\text{Adj}(T[y])|.
\]

This implies that

\[
\text{height}(\tilde{T}) = h_T(y) + \text{depth}_T(y) = h_T(y) + |\text{Adj}(T[y])|.
\]

**Case 2:** the node \( y \) is not on this path. Since \( y \) does not lie on this path, there must exist a node on this path, whose subtree is outside of \( T[y] \). Let \( w \) be the first such node. Then the parent of \( w \) is an ancestor of the node \( y \), and the subtree \( T[w] \) is outside of \( T[y] \). This implies that

\[
h_T(w) \leq \overline{h_T(y)}.
\]

Since the subtree \( T[w] \) is outside of \( T[y] \), the rotation at \( y \) will not affect the structure of the subtree \( T[w] \) so that \( h_{\tilde{T}}(w) = h_T(w) \). Moreover, since

\[
\text{Ancestor}_{\tilde{T}}(w) \subseteq \text{Ancestor}_T(y),
\]

we have \( \text{depth}_{\tilde{T}}(w) \leq \text{depth}_T(y) \). This implies that

\[
\text{height}(\tilde{T}) = h_{\tilde{T}}(w) + \text{depth}_{\tilde{T}}(w) \leq \overline{h_T(y)} + \text{depth}_T(y).
\]

Therefore, the height of the new tree \( \tilde{T} \) must be bounded by the maximum of the two given quantities. \( \square \)

**Theorem 4.2.** If \( \overline{h_T(y)} < h_T(y) \), then after the rotation at \( y \),

\[
\text{height}(\tilde{T}) < \text{height}(T).
\]

**Proof.** Since the node \( y \) is eligible for rotation, it follows from definition that

\[
|\text{Adj}(T[y])| < \text{depth}_T(y).
\]
Substituting it and the given condition into the bound of Theorem 4.1, we have
\[
\text{height}(\tilde{T}) \leq \max\{ h_T(y) + |\text{Adj}(T[y])|, \bar{h}_T(y) + \text{depth}_T(y) \}
\leq h_T(y) + \text{depth}_T(y) \leq \text{height}(T). \quad \square
\]

4.3. Elimination tree transformation algorithm

The result of Theorem 4.2 provides a strategy to reduce the height of an elimination tree by finding an eligible node \( y \) with the given property. We can express the strategy algorithmically as follows:

```plaintext
while \( T \) has an eligible node \( y \) with \( \bar{h}_T(y) < h_T(y) \) do
    apply an elimination tree rotation at \( y \)
    and reassign the new tree to \( T \);
```

Elimination tree rotation is introduced in [16] as a basic restructuring operation of the tree. In practical terms, it is more effective to perform a composite of rotations at one time (in effect, a sequence of rotations). An efficient implementation of composite rotations is also described in [16]. Its time and space complexity are both proportional to the number of nonzeros in the given sparse matrix \( A \) (not the filled matrix \( F \)).

The problem then becomes the search for a node appropriate for height reduction by composite rotations. Our approach is to first find an eligible node \( y \) satisfying Theorem 4.2. But instead of performing the actual rotation at \( y \), we only do it logically by recording the necessary information so that the next eligible node for the new tree can be determined. This is then repeated until no such node can be determined. We shall use the last eligible node \( x \) encountered for the composite rotations.

We now consider the necessary information that need to be recorded. Let \( y \) be the eligible node in the tree \( T \) at which we want to perform a (logical) rotation, and let \( \tilde{T} \) be the resulting tree. First note that the rotation will not affect the subtree rooted at \( y \), that is, for any \( v \in T[y] \),
\[
\tilde{T}[v] = T[v].
\]
Since the nodes in \( \text{Adj}(T[y]) \) are to be labeled last from the rotation, the new depth of the node \( y \) will be exactly given by
\[
\text{depth}_T(y) = |\text{Adj}(T[y])|.
\]
This implies that the depth reduction of the node \( y \) is then
\[
\text{depth}_T(y) - \text{depth}_T(y) = \text{depth}_T(y) - |\text{Adj}(T[y])|.
\]
This quantity occurs quite often in our discussion.

Our search algorithm will continue with the subtree \( \tilde{T}[y] = T[y] \) (without the formation of \( \tilde{T} \)). Therefore for the node \( v \in T[y] \) to be selected in \( \tilde{T} \), we need to know the values of \( |\text{Adj}(\tilde{T}[v])| \), \( \text{depth}_T(v) \), \( h_T(v) \), and \( \bar{h}_T(v) \) in order to apply Theorem 4.2 to this node \( v \). Since the rotation preserves the subtree at \( v \), we have
\[
|\text{Adj}(\tilde{T}[v])| = |\text{Adj}(T[v])|, \quad h_T(v) = h_T(v) = h_T(v).
\]
Furthermore, the new depth of \( v \) will be reduced by the same amount as \( y \) so that
\[
\text{depth}_T(v) = \text{depth}_T(v) - \{ \text{depth}_T(y) \} = \text{depth}_T(v) - |\text{Adj}(T[y])|.
\]
In the next theorem, we shall provide an upper bound on the value \( \bar{h}_T(v) \) for the new tree. This upper bound will be used in Algorithm 4.1 to perform the search for a node appropriate
for height reduction by composite rotations. We need a lemma which states that the height of subtrees outside of $T[y]$ will be increased by no more than the depth reduction of $y$. Its proof is left to the reader.

**Lemma 4.3.** Let $\tilde{T}$ be the new elimination tree after the rotation of $T$ at the eligible node $y$. Then,
$$h_{\tilde{T}}(y) \leq h_{T}(y) + \text{depth}_{T}(y) - |\text{Adj}(T[y])|.$$ 

**Theorem 4.4.** Let $v \in T[y]$, and $\tilde{T}$ be as in Lemma 4.3. Then,
$$h_{\tilde{T}}(v) \leq \max\{h_{T}(v), h_{\tilde{T}}(y) + \text{depth}_{T}(y) - |\text{Adj}(T[y])|\}.$$ 

**Proof.** It follows from Lemma 4.3 and the fact that $h_{\tilde{T}}(v) \leq \max\{h_{T}(v), h_{\tilde{T}}(y)\}$. 

**Algorithm 4.1.** Search ($T$, $x$)

- $y :=$ root of tree $T$;
- $x := 0$;
- $h_t := -1$; \{to store a bound on the value of $h_t$ in Theorem 4.4\}
- $\text{reduced} := 0$; \{to store the current amount of height reduced\}
- while $h_{T} (y) > h_t$ and $h_{T} (y) > 0$ do
  - $\text{newdepth} := \text{depth}_{T}(y) - \text{reduced}$; \{depth of $y$ in current tree\}
  - if $\text{newdepth} > |\text{Adj}(T[y])|$ then
    - $x := y$;
    - $\text{reduced} := \text{depth}_{T}(y) - |\text{Adj}(T[y])|$;
    - $h_t := h_t + \text{newdepth} - |\text{Adj}(T[y])|$; \{use Lemma 4.3\}
  - if $y$ has more than one child then
    - $m2 :=$ second largest height among the children of $y$;
    - $h_t := \max\{m2, h_t\}$;
  - $y :=$ a child of $y$ with the largest height value;
- end.

Consider the application of Algorithm 4.1 to the elimination tree of Fig. 3. The Search algorithm first sets $x$ to the node $h$ and then finally returns the node $x = b$. Once a node $x \neq 0$ has been determined by the search algorithm of 4.1, the "Composite_Rotations ($T$, $x$)" algorithm in [16] can be applied. This will give a new elimination tree with its height reduced. To make this paper self-contained, we include a description of this algorithm below. For its correctness and illustrative examples, the readers are referred to [16].

**Algorithm 4.2.** Composite_Rotations ($T$, $x$)

- all nodes are considered as unnumbered;
- $v := x$;
while $v$ is not the root do
  begin
    order the unnumbered nodes of $\text{Adj}(T[v])$ last, before those already renumbered;
    mark nodes in $\text{Adj}(T[v])$ as numbered;
    $v :=$ the parent node of $v$;
  end;
number the remaining unnumbered nodes using their original relative order;
end.

The overall reordering scheme can be summarized as follows. Here, we assume that a
fill-reducing ordering $P$ for $A$ has already been determined:

Step 1: form the elimination tree $T = T(PAP^T)$,
Step 2: compute the values $h_T(v)$ and $\text{depth}_T(v)$ for each node $v$,
Step 3: compute $|\text{Adj}(T[v])|$ for each node $v$ (the number of off-diagonal nonzeros in the
column associated with $v$ in the factor matrix),
Step 4: determine a node $x$ appropriate for height reduction by $\text{Search}(T, x)$,
Step 5: transform the tree and obtain an equivalent reordering $\tilde{P}$ by $\text{Composite}_\text{Rotations}(T, x)$.

The time complexity for this reordering algorithm is simply the sum of the complexity for
the five steps. Finding the elimination tree structure can be performed in time almost linear to
the number of nonzeros in $A$ using Tarjan's method of path compression with balancing [17,23].
And at worst, this step is linear to the number of nonzeros in filled graph $F$. It should be clear
that computing the height and depth of each node and the $\text{Search}$ algorithm can be done in
$O(n)$ time. The $\text{Composite}_\text{Rotations}$ algorithm in Step 5 is known to have a time complexity
proportional to the number of nonzeros in $A$ [16]. Finally, in Step 3, computing the values $|\text{Adj}(T[v])|$ for each node $v$ can be accomplished in time proportional to the number of
nonzeros in the filled matrix $F$. Therefore, the overall reordering scheme is linear with respect
to the number of nonzeros in $F$.

It is instructive to apply the reordering scheme to the example in Fig. 3. As noted before, the
$\text{Search}$ algorithm returns the node $x = b$. On applying $\text{Composite}_\text{Rotations}$ to this tree at the
node $b$, we obtain the tree structure and the new reordering as given in Fig. 5.

This example serves to illustrate another point: this reordering algorithm by $\text{Search}$ and
$\text{Composite}_\text{Rotations}$ does not always produce an elimination tree with minimum height. The
one in Fig. 5 after the rotations has height 4, but the minimum height is known to be 3 from

![Fig. 5. Restructuring elimination tree of Fig. 3 by Composite Rotations at node b.](image-url)
Fig. 2. However, in the next section, we shall see that for most practical sparse problems tested, this reordering scheme produces elimination trees with minimum or near-minimum height.

5. Experimental results

In this section, we provide experimental results on the minimum-height reordering scheme of Algorithm 3.1 as described in [18], and the height reducing scheme of Section 4.3. Nine problems from the Harwell-Boeing collection of practical sparse matrix examples [7] are selected. These are the same symmetric problems the author used in [17]. We shall use the same keys as used by the Harwell-Boeing tape to identify the different matrices. For a description of these problems, the readers are referred to the actual tape collection or to [17]. We have also added two \( k \times k \) regular grid problems with \( k = 40 \) ("GR 40 \( \times \) 40") and 80 ("GR 80 \( \times \) 80").

For each matrix problem, the popular fill-reducing minimum degree algorithm is used to order the equations. The version used is the recent implementation of the modified ordering algorithm by multiple elimination [14].

Table 1 provides the statistics on the number of nonzeros in the factor matrix and the time required for the different phases of the ordering. "Matrix nonz." and "Factor nonzeros" refer to the number of off-diagonal nonzeros in the original sparse matrix and the Cholesky factor matrix respectively. The label "MDeg" corresponds to the basic minimum degree ordering from [14]. The column "MDeg" gives the number of nonzeros in the factor matrix for both the minimum degree ordering and the reordering using rotations. The numbers are the same since they are equivalent orderings. The one labeled "MDeg + MinHt" is the minimum degree ordering followed by the minimum-height reordering algorithm described in [18].

Times reported are in CPU seconds on a SUN 3/50 Workstation. The ordering time "Rotn" is the time required to execute Steps 1 to 5 of the reordering algorithm in Section 4.3, and it does not include the minimum degree ordering time. The column "FillG" gives the time used to form the filled graph, while "MinHt" is the time for the execution of the minimum-height reordering scheme. We give these time components here for a better understanding of the respective ordering and reordering time.

Table 2 gives the height information and the total ordering time. The column "MaxHt bound" is obtained by using a scheme that rotates the tree to increase its height. In other
words, it gives a lower bound on the maximum height of all elimination trees from equivalent reorderings. It is included here to provide some indication on how bad the tree can be (in terms of height) even if the amount of fills and factorization operations are preserved.

The total ordering time is obtained from the time components provided in Table 1. "MDeg + Rotn" equals to the ordering time "MDeg" plus "Rotn" in Table 1, and it represents the total time to perform the ordering together with the reordering by rotations. The column "MDeg + MinHt" corresponds to the sum of ordering time "MDeg", FillG" and "MinHt"; and it is the total time for ordering and reordering to perform the minimum height scheme. It should be noted that there is a great difference in height between the quantities "MDeg + MinHt" and "MaxHt bound". Indeed, the minimum height is often 2/3 that of "MaxHt bound". It is still substantially smaller than the height from the basic minimum degree ordering. For example, for the problem "DWT 2680", the height difference is 109.

However, the elimination tree obtained by "MDeg + Rotn" is amazingly close to the minimum height. It is as small as the minimum height for all problems, except for two ("BCSPWR 10" and "DWT 2680"); and even for them, the difference is negligible. It is interesting to point out that for these two cases, the height difference can be attributed to the fact that "MDeg + MinHt" is a better reordering than "MDeg + Rotn" in terms of fill (compare "Factor nonzeros" columns in Table 1). A discussion of such situation has been given in Section 3.4.

The advantage of the “MDeg + Rotn” reordering is clear. It produces elimination trees with minimum or near-minimum height using a relatively small amount of reordering time. Comparing only the reordering time (that is, exclude the time for the basic minimum degree ordering), “MDeg + Rotn” requires two to six times less reordering time than “MDeg + MinHt”. Even for the total ordering/reordering time, there is always a substantial reduction, sometimes over 50%.

For simplicity and convenience, our implementation of “MDeg + MinHt” constructs the filled graph explicitly. This allows a simpler implementation of the minimum-height reordering algorithm. On the other hand, this incurs a significant storage requirement due to the explicit representation of the structure of the filled graph. In any case, “MDeg + Rotn” is the practical scheme to recommend.

As a final note, the ratio “n/(Elim tree height)”, which can be computed from Table 2, gives the average number of nodes that can be eliminated in parallel at each step. The number is

<table>
<thead>
<tr>
<th>Problem</th>
<th>Elim tree height</th>
<th>Total ordering time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MDeg</td>
<td>MDeg</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ Rotn</td>
</tr>
<tr>
<td>BCSPWR09</td>
<td>73</td>
<td>50</td>
</tr>
<tr>
<td>BCSPWR10</td>
<td>145</td>
<td>112</td>
</tr>
<tr>
<td>BCSSSTK08</td>
<td>241</td>
<td>197</td>
</tr>
<tr>
<td>BCSSSTK13</td>
<td>655</td>
<td>639</td>
</tr>
<tr>
<td>BCSSSTM13</td>
<td>283</td>
<td>242</td>
</tr>
<tr>
<td>BLCKHOLE</td>
<td>210</td>
<td>210</td>
</tr>
<tr>
<td>CAN1072</td>
<td>134</td>
<td>134</td>
</tr>
<tr>
<td>DWT 2680</td>
<td>373</td>
<td>265</td>
</tr>
<tr>
<td>LSHP3466</td>
<td>317</td>
<td>273</td>
</tr>
<tr>
<td>GR 40×40</td>
<td>224</td>
<td>174</td>
</tr>
<tr>
<td>GR 80×80</td>
<td>410</td>
<td>364</td>
</tr>
</tbody>
</table>
reasonably high for the problems tested. This justifies our approach of performing parallel elimination on the chordal graph of the filled matrix using a predetermined fill-reducing ordering.

6. Generalization of height to parallel completion time

As mentioned in Section 2, a relevant criterion for desirable elimination trees for parallel factorization is parallel completion time. We shall formalize this notion here, and relate it with the height criterion.

Let $\text{time}[v]$ denote the amount of execution time associated with the task/node $v$ in the elimination tree model. For our purpose, it is appropriate to use the number of multiplicative operations to represent the time required for the task. Note that this time depends on which form of Cholesky factorization is used. For our consideration here, we shall assume the column-Cholesky form.

Following [5], we define the $\text{level}$ of a task $v$ to be the sum of the execution times associated with tasks along the path from $v$ to the root in the elimination tree. In other words,

$$\text{level}[v] = \begin{cases} 
\text{time}[v] & \text{if } v \text{ is the root}, \\
\text{time}[v] + \text{level}[\text{parent of } v] & \text{otherwise}.
\end{cases}$$

The value $\text{level}[v]$ represents the least amount of time required starting at the node $v$ to completion. The parallel completion time of the elimination tree is then defined to be the maximum level value among all nodes.

![Fig. 7. Elimination trees of graph in Fig. 6 with minimum height and minimum completion time (with time/level as node labels).](image)
Given the precedence relation represented by the tree, the parallel completion time represents the minimum time required to complete the factorization even with an unlimited number of parallel processors. Minimizing the parallel completion time is hence a reasonable criterion for the best elimination tree in parallel factorization.

We discuss its relationship with the height notion. If \( time[v] = 1 \) for every node \( v \), the level value is simply the depth of the node plus 1, and the parallel completion time will then correspond to the height of the tree plus 1. Minimizing the height does not necessarily imply minimizing the parallel completion time. The simple graph in Fig. 6 serves as a counter example. In Fig. 7, we display two elimination trees, which minimize the height and the parallel completion time respectively. The label in each node corresponds to “\( time/level \)” of the node (column-Cholesky form of factorization is used to determine the \( time \) values).

7. Conclusion

We have considered the parallel elimination reordering algorithm by Jess and Kees. It is shown to produce elimination trees of minimum height, a property extremely desirable for exploiting parallelism. We have also introduced a new height-reducing scheme using the notion of elimination tree rotations. It is demonstrated experimentally to be a viable alternative as a practical reordering scheme for parallel elimination. Our next step is to incorporate this reordering scheme as a separate phase into actual parallel solvers of linear systems, such as reported in [8].

Acknowledgment

The author would like to thank Professor Andranik Mirzaian for many helpful discussions.

References


