Practical Parallel Algorithms for Chordal Graphs

Extended Abstract

Eric S. Kirsch
Department of Computer Science
University of Tennessee
Knoxville, TN 37996
kirsch@cs.utk.edu

Jean R. S. Blair[†]
Department of Computer Science
University of Tennessee
Knoxville, TN 37996
blair@cs.utk.edu

Abstract

Until recently, a large majority of theoretical work in parallel algorithms has ignored communication costs and other realities of parallel computing. This paper attempts to address this issue by developing parallel algorithms that not only are efficient using standard theoretical analysis techniques, but also require a minimal amount of communication. The specific parallel algorithms developed here include one to find the set of maximal cliques and one to find a perfect elimination ordering of a choral graph.

1. Introduction

Most previous theoretical results pertaining to parallel algorithms have focused on proving the corresponding problem to be in NC, i.e., striving for poly-logarithmic running time while letting the number of processors grow with the problem size. This approach ignores communication costs which are incurred on real machines. Thus, these algorithms are not necessarily the most practical or even the fastest algorithms for existing machines. Recent results have focused on the communication delay involved in parallel processing. For example, in [2], communication delay is used as an integral part the analysis of the time complexity of parallel algorithms. This leads to the conclusion that algorithms which require no communication are superior algorithms since they can be run on any number of processors without incurring communication delay.

In this extended abstract, new parallel algorithms are presented to solve two problems on chordal graphs: finding the set of maximal cliques and finding a perfect elimination ordering(PEO). These new algorithms have several advantages. First, during the maximal clique algorithm, each processor performs its work without use of shared memory or communication. Previously developed algorithms use shared memory to communicate intermediate results to other processors. This means that these algorithms will incur a communication delay and require processors to operate synchronously. During the maximal clique algorithm presented here, each processor can perform its work without any knowledge of results from other processors, thus, avoiding any communication delays. Second, the partitioning of data for both algorithms is simple while, in other parallel

[†] Portions of this research were performed at the Mathematical Sciences Section of Oak Ridge National Laboratory and partially supported by the Applied Mathematics Science Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-840R21400 with Martin Marietta Energy Systems Inc.

algorithms, the data requirements are often generated dynamically, making shared memory or communication a requirement. Third, while the algorithms can run on shared memory machines, the algorithms are designed for distributed memory machines, taking into consideration synchronization and communication costs. Fourth, the algorithms exhibit good scalability. That is, as processors are added, equal speed-up is achieved. Since we define *work* to be the running time of the algorithm times the number of processors, this means that on a parallel machine with a fixed number of processors (the only kind in existence) these new algorithms will run faster when the size of the problem exceeds the number of processors. Fifth, the work performed by the algorithms here is less than or comparable to the work performed by algorithms presented elsewhere for sparse graphs. Since one of the prime uses for the algorithms presented here is found in the solution to sparse matrices, the algorithms presented here are of practical importance. Previous algorithms make no use of the fact that the graphs corresponding to these matrices are also sparse.

The remainder of the paper is organized as follows. Section 2 presents some definitions and a basic version of the maximal clique algorithm. Section 3 gives a fast version of the maximal clique algorithm, and section 4 presents the PEO algorithm. Concluding remarks are in section 5.

2. The Basic Maximal Clique Algorithm

In this section, we present a basic version of a practical algorithm to find all maximal cliques for chordal graphs. We will develop the algorithm further in the next section. We begin with some definitions.

An undirected graph $G = (V, E)$, is *chordal* if every cycle of length 4 or greater has an edge between vertices that are not consecutive in the cycle. Chordal graphs have important applications in sparse matrix computation[8, 9] and in acyclic database schemes[1]. A *clique* is a set of vertices whose induced subgraph is complete. A clique is *maximal* if there is no clique that properly contains it. We use the term $adj(v)$ to refer to the set $\{u|(u, v) \in E\}$. We let $d_v = |adj(v)|$ and c_v be the number of maximal cliques to which vertex v belongs.

For the maximal clique algorithm, the data will be maintained as n adjacency lists, where $n = |V|$. The ith adjacency list is a list of vertices to which v_i is adjacent. We assume an implicit (arbitrary but consistent) ordering of these $v_1, v_2, ...v_n$ and that all adjacency lists adhere to this ordering. (That is, for each adjacency list, if v_i and v_j are both in the list and $i < j$, then v_i appears before v_j in the list).

Each processor will develop the set of maximal cliques for a single vertex p. To do this, each processor needs to have the adjacency vectors for each $u \in adj(p)$. This distribution of data can be accomplished using a direct access, concurrent read file. The time required for this operation on a distributed memory parallel processor will be limited by the number of I/O processors but will be at most $O(nd)$. For a shared memory machine, n processors could read the appropriate adjacency vectors in $O(d)$ time.

The key idea that we will use to develop maximal cliques is that if a vertex p is adjacent to a vertex v, then p and v are in a maximal clique together. As the algorithm proceeds, pro-

cessor p will maintain a *Candidate Clique List*($\boldsymbol{CCL}_p$) for vertex p. Initially, we have one *Candidate Clique*(**CC**), the adjacency vector for p; upon termination, CCL_p will contain all maximal cliques(**MC**) to which p belongs. During processing, CCL_p is revised by considering each $v \in adj(p)$ as a *refinement vertex*(**rv**) in turn. Given an rv, each $CC_i \in CCL_p$ is checked to see if it contains rv. When rv is contained in CC_i, we check to see if rv is adjacent to each $v \in CC_i$. If it is, we continue to assume that this CC_i is correct. If it is not, we create two new CCs to replace CC_i. The first will contain rv and all $v \in CC_i$ such that $v \in adj(rv)$. The second will be $CC_i - \{rv\}$. It is important to note that as we make these refinements it is possible to develop candidate cliques that cannot lead to a maximal clique. For example, if CC_i is a subset of CC_j, then CC_i cannot lead to a maximal clique since the refinement process can only remove corresponding vertices from CC_i and CC_j and no vertices can be added.

While we will expand upon the algorithm described above, we will prove the correctness of this basic approach and then later prove that we can add additional steps to reduce the time complexity without violating its correctness. We begin by defining a few terms. Consider each candidate clique CC_i prior to refinement by rv. Let the prefix P_i of CC_i be the set of vertices in CC_i that have already been used to refine the CCL_p. Let the suffix S_i be the remaining set of vertices in CC_i. Note that rv may or may not be in S_i.

The following theorem shows that every maximal clique to which p belongs is in CCL_p at the completion of the algorithm.

Theorem 1 *A processor p using the above algorithm will halt with CCL_p containing every maximal clique to which vertex p belongs.*

Proof: Let $R = \{v_1, v_2, ..., v_x\}$ be the set of vertices that have been used to refine CCL_p and $\{M_1, M_2, ..., M_c\}$ be the set of maximal cliques for G. Let us define an equivalence relationship Λ on the set of maximal cliques defining classes $\lambda_1, \lambda_2, ...\lambda_l$ such that for each class λ_i defined by Λ, both M_a and M_b are in λ_i, if and only if $M_a \cap R = M_b \cap R$. Obviously, since one more rv is added to R after each refinement, these classes may change as the algorithm progresses. We also let $N_i = \bigcup_{M_a \in \lambda_i} M_a$. We will show, using induction on the number of refinements, that each equivalence class λ_i will be represented in CCL_p by a CC_i which consists of a prefix $P_i = N_i \cap R$, and a suffix $S_i = N_i - R$. We will also show that, after each refinement, $\bigcup_{\lambda_i \in \Lambda} \lambda_i = \{M_1, M_2, ..., M_c\}$ and $\bigcap_{\lambda_i \in \Lambda} \lambda_i = \emptyset$. Now, after n refinements, R contains all n vertices and it is not possible that $M_a \cap R = M_b \cap R, a \neq b$. Therefore, each class λ_i contains one unique maximal clique and there is a $CC_i \in CCL_p$ for each maximal clique.

It remains to be shown that each equivalence class λ_i is represented by a $CC_i = P_i \cup S_i$ where $P_i = N_i \cap R$ and $S_i = N_i - R$ and that $\bigcup_{\lambda_i \in \Lambda} \lambda_i = \{M_1, M_2, ..., M_c\}$ and $\bigcap_{\lambda_i \in \Lambda} \lambda_i = \emptyset$. We will use induction on the number of vertices used as the refinement vertex. The base case is the starting condition of one $CC_1 = adj(p)$, and Λ defines one class λ_1 which contains all maximal cliques. Obviously, $\bigcup_{\lambda_i \in \Lambda} \lambda_i = \{M_1, M_2, ..., M_c\}$ and $\bigcap_{\lambda_i \in \Lambda} \lambda_i = \emptyset$. $R = \emptyset$, so $N_1 \cap R = \emptyset = P_1$. Now, p is in a maximal clique with a vertex v if and only if $v \in adj(p)$, so $N_1 = adj(p)$ is correct. It is also true that $CC_1 - P_1 = S_1 = adj(p) = N_1 - R$.

Now, assume that after $k-1$ refinements the above conditions are true, and consider the kth refinement by vertex rv. Consider a CC_i that defines a class λ_i such that $rv \notin CC_i$ before refinement by rv. It must be true that no $M_a \in \lambda_i$ contains rv and therefore N_i does not contain rv. Thus, after refinement by rv, $N_i \cap R$ is the same as is the prefix. Thus, $P_i = N_i \cap R$. $N_i - R$ also remains the same and since no $M_a \in \lambda_i$ contained rv, S_i remains the same and $S_i = N_i - R$. CC_i is the same after refinement by rv and correctly defines the same class λ_i.

Now, consider a CC_i that defines a class λ_i which does contain rv. There are two possibilities. First, $\forall u \in S_i$, $u \in adj(rv)$. In this case, CC_i does not split and defines the same class. Consider any maximal clique $M_a \in \lambda_i$. Since rv is adjacent to all $u \in CC_i$, it must be true that any M_a represented by CC_i must contain rv. Thus, it is still true that $P_i = N_i \cap R$. Since S_i has been reduced by only rv which is now R, it is still true that $S_i = N_i - R$. Again, CC_i is the same after refinement by rv and correctly defines the same class λ_i.

For the second possibility, suppose that rv is not adjacent to some set of vertices L in the suffix of CC_i defining the class λ_i. After refinement by rv, we will have two new candidate cliques $CC_r = CC_i - L$ and $CC_s = CC_i - \{rv\}$. We also refine the class λ_i into two classes: $\lambda_r = \{M_a | M_a \in \lambda_i \text{ and } rv \in M_a\}$ and $\lambda_s = \{M_a | M_a \in \lambda_i \text{ and } rv \notin M_a\}$. Obviously, $\lambda_r \cup \lambda_s = \lambda_i$ and $\lambda_r \cap \lambda_s = \emptyset$. Now, $rv \in N_r$ for λ_r and, since P_r also contains rv, $P_r = N_r \cup R$. Also, since any M_a which contains rv cannot contain any $u \in L$, N_r does not contain any $u \in L$ and $S_r = N_r - R$. For λ_s, $rv \notin N_s$ and $rv \notin P_s$. Therefore, $P_s = N_s \cap R$. Now, since the only element added to R is rv and no $M_a \in \lambda_s$ can contain rv, $N_s = N_i - \{rv\}$ and $S_s = S_i - \{rv\}$. Thus, $S_s = N_s - R$.

Since each CC_i either defines the same class after refinement or breaks into CC_r and CC_s such that $\lambda_r \cup \lambda_s = \lambda_i$ and $\lambda_r \cap \lambda_s = \emptyset$, after every refinement, it is true that $\bigcup_{\lambda_i \in \Lambda} \lambda_i = \{M_1, M_2, ..., M_c\}$ and $\bigcap_{\lambda_i \in \Lambda} \lambda_i = \emptyset$. Thus, after n refinements, each λ_i represents just one maximal clique and is represented by $CC_i \in CCL_p$. $\square$

Theorem 1 shows that upon completion of the algorithm, there exists a $CC_i \in CCL_p$ for each maximal clique containing p. It should also be clear that at that point each $CC_i \in CCL_p$ represents a clique containing p. Removal of non-maximal cliques from CCL_p at this point is straight forward.

3. A Fast Maximal Clique Algorithm

Theorem 1 in the preceding section showed that our basic algorithm correctly develops the set of maximal cliques to which a vertex p belongs. To do this, the algorithm essentially scans CCL_p for each vertex in $adj(p)$. Unfortunately, the number of candidate cliques in CCL_p may become large, thereby making the running time of the algorithm expensive. The next results examine some possibilities for removing candidate cliques from CCL_p whenever it becomes clear that they cannot lead to a maximal clique. When we determine that a CC_i can not lead to a maximal clique we say it "can be removed from CCL_p". We define a CC_i to be a *Maximal Candidate Clique* ($\boldsymbol{MCC}_i$) if P_i is the prefix of at least 1 maximal clique. The following easy to prove properties greatly reduce the possible size of CCL_p.

Lemma 1 *If CC_i and CC_j are two distinct candidate cliques in CCL_p, then*
(i) *$P_i \neq P_j$, and*
(ii) *if $CC_i \subset CC_j$ then CC_i cannot represent a maximal clique and can be removed from CCL_p.*

Consider a modified version of our algorithm that removes candidate cliques as soon as they become subsets of other candidate cliques. We have the following result.

Lemma 2 *Using the modified algorithm, every candidate clique $CC_i \in CCL_p$ is either MCC_i or has the form $CC_i = \{\bigcap_{x=1}^{k} P_{j_x}\} \cup \{\bigcup_{x=1}^{k} S_{j_x}\}$ for some set $\{CC_{j_1}, CC_{j_2}, ...CC_{j_k}\}$, $k \geq 2$.*

Proof (sketch): To prove this, we consider what possible forms CC_i might take on during the modified algorithm and show that each possible form either: (1) guarantees that CC_i is an MCC_i, (2) won't happen because subset candidate cliques are removed, or (3) has the form described in the statement of this lemma. □

The following lemmas (proofs omitted) show that the modified algorithm will never have more than $O(d_p^2)$ candidate cliques in CCL_p.

Lemma 3 *A vertex p can belong to at most d_p maximal cliques.*

Lemma 4 *There are never more than d_p MCCs in CCL_p for a vertex p.*

Lemma 5 *There can be no more than $O(d_p^2)$ candidate cliques in CCL_p for vertex p that are not MCCs.*

Although the modified version of our algorithm has a time complexity that is substantially less than that of our basic algorithm, it is still too expensive. In order to remove the candidate cliques which become subsets, after each refinement each $CC_i \in CCL_p$ must be checked to see if it is a subset of another CC_j. Thus, to allow $O(d_p^2)$ candidate cliques in CCL_p slows the algorithm considerably. However, using the fact that G is chordal, we can detect more candidate cliques that can be removed from the CCL_p during the processing.

Let CC_i be of the form, $CC_i = \{\bigcap_{x=1}^{k} P_{j_x}\} \cup \{\bigcup_{x=1}^{k} S_{j_x}\}$ for some set $\{CC_{j_1}, CC_{j_2}, ...$ $CC_{j_k}\}$, $k \geq 2$, of candidate cliques in CCL_p. That is, the prefix of CC_i is contained in the prefixes of all the candidate cliques in the set, and S_i is the union of the suffixes of all the candidate cliques in the set. We call CC_i the **child** of each CC_{j_x}, $1 \leq x \leq k$, and each such CC_{j_x} the **parent** of CC_i.

Theorem 2 *Suppose CC_{p_1} and CC_{p_2} are distinct parents of CC_i. If there is another candidate clique CC_g such that $P_{p_1} \subset P_g$ and $P_{p_2} \subset P_g$, then CC_i can be removed from CCL_p.*

Proof: It must be true that $P_{p_1} \neq P_{p_2}$ by Lemma 1. It must also be true that $S_{p_1} \neq S_{p_2}$. Otherwise, there would be another candidate clique $CC_s = P_{p_1} \cup P_{p_2} \cup S_{p_1}$ which implies that both CC_{p_1} and CC_{p_2} would be removed from CCL_p. Since $P_{p_1} \subset P_g$ and $P_{p_2} \subset P_g$ it must be true that all $v \in P_{p_1}$ are adjacent to all $u \in P_{p_2}$. Let us consider $u \in P_{p_1}, u \notin P_{p_2}$, $v \in P_{p_2}, v \notin P_{p_1}$ $w \in S_{p_1}, w \notin S_{p_2}$ and $x \in S_{p_2}, x \notin S_{p_1}$. There are two possibilities. First, these vertices might imply a chordless cycle: $u \rightarrow v \rightarrow x \rightarrow w \rightarrow u$. Since there cannot be a cordless cycle in our chordal graph, we can remove CC_i. The other possibility is that u is adjacent to all $x \in S_{p_1}, x \notin S_{p_2}$ (and v is adjacent

to all $w \in S_{p_2}, w \notin S_{p_1}$). But this implies the existence of a $CC_k = P_i \cup \{u\} \cup S_i$ and CC_i can be removed. □

The fact that parent candidate cliques CC_{p_1} and CC_{p_2} both have prefixes which are subsets of a single prefix is necessary to ensure that all $v \in P_{p_1}$ and all $u \in P_{p_2}$ are adjacent. Thus, for any arrangement of cliques where a CC_i has two parents and those two parents have a common ancestor, CC_i can be removed. This property gives us the final modification to our basic maximal clique algorithm. We will call the basic algorithm with the two modifications incorporated (removal of subset candidate cliques and removal of candidate cliques that have parents that have prefixes where elements in these prefixes are adjacent) the maximal clique algorithm. We can show:

Lemma 6 *Given a set of MCCs, there must be at least $O(d_p)$ refinement vertices for there to be $O(d_p)$ candidate cliques in CCL_p.*

Corollary 1 *There are at most $O(d_p)$ candidate cliques in CCL_p during the maximal clique algorithm.*

We are now ready to formally present the algorithm for finding maximal cliques.

Algorithm: AlgMaxCl.

Input: A Chordal Graph G.

Output: The set of maximal cliques for G.

Method: Initially, each processor p is given the adjacency list for p and the adjacency list for each $v \in adj(p)$. The processor for vertex p will maintain a candidate clique list (CCL_p). Each $CC_i \in CCL_p$ is refined by each vertex $rv \in adj(p)$. After each refinement, each $CC_i \in CCL_p$ is checked to determine if it is a subset of another CC_j or if it violates the chordality of G according to Theorem 2, and thus can be removed. After n refinements, each $CC_i \in CCL$ is a maximal clique. Details are given in Figure 1.

Using our previous results, it is easy to show:

Theorem 3 *A processor/vertex p using AlgMaxCl correctly develops the set of maximal cliques to which p belongs.*

Theorem 4 *The running time for AlgMaxCl on processor/vertex p is $O(d_p^4)$*

Proof(sketch): The loop of statement 2 will be executed d times. The loops in statements 2(a), 2(c), 2(c)(i), 2(d), and 2(d)(i) are each executed $|CCL_p|$ times. We need to show that $|CCL_p| \in O(d_p)$. This follows from the fact all candidate cliques which have the form described in Theorem 2 are removed by the algorithm. To see this, note that any parents of a CC_i, which can be removed according to Theorem 2 must have at least two parents CC_{p_1} and CC_{p_2} which will be elements of $Parents_i$. When $Parents_{p_1}$ is unioned with $Grandpars_i$ and then $Grandpars_i$ is intersected with $Parents_{p_2}$ the intersection will be the parent(s) that CC_{p_1} and CC_{p_2} share. Thus, if the intersection is non-empty, CC_i will be removed. Therefore, all candidate cliques which can be removed according to Theorem 2 are removed by AlgMaxCl. Thus, by Corollary 1, $CCL_p \in O(d_p)$.

Each of the set operations in steps 2(a)(i)(A), 2(a)(i)(B), 2(c)(i)(A), 2(C)(i)(B), 2(d)(i)(A), and 2(d)(i)(B) require d_p time. Thus, the running time of AlgMaxCl on each processor $p \in O(d_p^4)$. □

Corollary 2 *The work performed by a group of processors to find the maximal cliques for graph G with n vertices is $O(nd^4)$, where $d = max(d_p)$*

1. $CCL_p \leftarrow adj(p)$
2. for each $rv \in adj(p)$
   (a) for each $CC_i \in CCL_p$
      (i) if $rv \in CC_i$
         (A) $result \leftarrow CC_i \cap adj(rv)$ (is rv adjacent to all $u \in S_i$?)
         (B) if $result \neq CC_i$ (rv not adjacent to all $u \in S_i$)
            $CCL_p \leftarrow CCL_p \cup result$
            $CC_i \leftarrow CC_i - \{v\}$
   (b) for each $CC_i \in CCL_p$
      (i) $Parsuffix_i,\ Parents_i,\ Grandpars_i \leftarrow \emptyset$
   (c) for each $CC_i \in CCL_p$ (determine if CC_i is subset)
      (i) for each $CC_j \in CCL_p, i \neq j$ (and determine parents of CCi)
         (A) if $CC_i \subset CC_j$
            $CCL_p \leftarrow CCL_p - CC_i$ (remove CC_i from list)
         (B) else if $P_i \subset P_j$ and $P_i \neq \emptyset$
            $Parents_i \leftarrow Parents_i \cup \{j\}$ (add CC_j to parent list)
            $Parsuffix_i \leftarrow Parsuffix_i \cup S_j$
   (d) for each $CC_i \in CCL_p$
      (i) for each $CC_j \in CCL_p, i \neq j$
         (A) if $CC_j \in Parents_i$ (CC_i meets
            and $Grandpars_i \cap Parents_j \neq \emptyset$ conditions
            and $S_i \subseteq Parsuffix_i$ of Lemma 2)
            remove CC_i from CCL_p
         (B) else $Grandpars_i \leftarrow Grandpars_i \cup Parents_j$

Figure 1: Details of AlgMaxCl

4. The PEO Algorithm

In this section, we will look at an algorithm to develop a Perfect Elimination Ordering (PEO) for a chordal graph. A *PEO* is an ordering of the vertices of a chordal graph G such that each vertex v_i is simplicial on the induced subgraph $G - \{v_j | j < i\}$. A vertex v is *simplicial* if $adj(v)$ is a clique.

The PEO algorithm presented here works with the information developed during the maximal clique algorithm, but requires no information transfer to update each processor. That is, to begin it only needs a list of its Maximal Cliques ($\boldsymbol{MCL}_p$), which it can initialize to CCL_p from the previous algorithm. Thus, each processor may begin its work on the PEO as soon as it completes its work on the maximal cliques. That is, it is not necessary for all processors p to have completed before a processor starts on its portion of the PEO algorithm. The algorithm is based on the following widely accepted lemma.

Lemma 7 *Any simplicial vertex is a member of exactly one maximal clique.*

A consequence of Lemma 7 is that we can find a PEO from the maximal cliques that have already been found. It is possible to determine if a vertex is simplicial by determining when it is a member of exactly one clique. If each vertex that becomes simplicial is removed from all processors' set of maximal cliques, and as maximal cliques become subsets of other maximal cliques they too are removed, then it can be determined when a vertex is a member of one maximal clique and is, therefore, simplicial.

1. $timestamp \leftarrow 0$
2. while $|MCL| > 1$
   (a) receive(wait) simplicial vertex message($v, rtimestamp$)
   (b) find $MC_i \in MCL$ such that $v \in MC_i$
   (c) $MC_i \leftarrow MC_i - \{v\}$
   (d) for each $MC_j \in MCL, i \neq j$
   if $MC_i \subset MC_j$ then
   $MCL \leftarrow MCL - MC_i$
   (e) if $rtimestamp > timestamp$ then
   $timestamp \leftarrow rtimetamp$
3. $timestamp \leftarrow timestamp + 1$
4. send msg(p,$timestamp$) to all $v \in$ final MC and to host

Figure 2: Details of processor AlgPEO

Any vertex v which is simplicial in the original graph (of which there must be at least 2) will end AlgMaxCl with $|MCL_v| = 1$. These processors will send messages to all $p \in MC_v$, where MC_v is the maximal clique to which v belonged, indicating that v is simplicial. Included in this simplicial message will be a timestamp. Thus, a simplicial message from a vetex v means that v is simplicial on the induced subgraph $G - \{x|$ has timestamp less than $v\}$. Vertices which are simplicial on the original graph will send simplicial messages with a timestamp of 1.

Each $p \in MC_v$ will, in turn, remove v from MC_v in CCL_p. Now, because the induced subgraph of any chordal graph is chordal, it must be true that there are at least two new vertices/processors p for which $|MCL_p| = 1$. These processors now send simplicial messages with a timestamp equal the largest timestamp received plus 1. This process continues until all p have sent their simplicial messages. A host processor will collect the simplicial messages in order of these timestamps. The PEO algorithm, AlgPEO, is as follows.

Algorithm: AlgPEO

Input: For each processor p, a list of all maximal cliques containing p and a count of the number of cliques in that list.

Output: A PEO on the host processor.

Method: Each processor p will receive messages from simplicial vertices and remove these vertices from the appropriate maximal clique. When some $MC_j \subseteq MC_i$ for MC_j, $MC_i \in MCL_p$ then MC_j is removed. When $|MCL_p| = 1$ then a simplicial message is sent to all vertices in the final clique and the host processor. Each processor will also keep track of the largest timestamp that it receives and will send its simplicial message with a timestamp equal to the largest timestamp that it received plus 1. The details of the vertex processor algorithm are given in Figure 2.
The host processor will maintain a set of lists, one for each possible timestamp. Then, when it receives a simplicial message from vertex p, it places p in the list corresponding to the appropriate timestamp. The details of the host processor algorithm are given in Figure 3.

Theorem 5 *Given the set of maximal cliques for a chordal graph G, a group of n processors using AlgPEO will correctly develop a PEO on a host processor.*

Proof: To prove this we need to show that a vertex p is simplicial on the graph $G - \{x|x$ has timestamp less than $p\}$ and that p will send a message when it is simplicial.

We use induction on the timestamp to prove the first part. The base case is vertices which are simplicial on the original graph which send a message with a timestamp of 1. Obviously they are

1. $msgcount \leftarrow 0$
2. while $msgcount < n$
   (a) receive(wait) simplicial msg(v,$timestamp$)
   (b) add v to $PEO_{timestamp}$
   (c) $msgcount \leftarrow msgcount + 1$

Figure 3: Details of Host AlgPEO

simplicial on $G - \{x | x$ has timestamp less than $p\}$. For the induction step, assume true for vertices that send a message with timestamp $k-1$. If p is going to send a message with $timestamp = k$ then it must be true that $adj(p) \cap \{x | x$ has timestamp less than $p\}$ have sent messages to p which made $|MCL_p| = 1$. All vertices of this subset were simplicial by the induction hypothesis and since $|MCL_p| = 1$, p is simplicial.

For the second part of the proof, we again use induction on the timestamp. For the base case, a processor p which is simplicial on the original graph must have an $|MCL_p| = 1$ and thus will never enter the while loop of step 2. These processors will send their simplicial message when simplicial. For the induction step, assume true for all processors which send messages with $timestamp = k-1$. If a processor p is simplicial on the graph $G - \{x | x$ has timestamp less than $k\}$, then it must be true that $adj(p) \cap \{x | x$ has timestamp less than $k\}$, which we'll call T, have all sent messages to p by the induction hypothesis. It must be true that $adj(p) - T \subseteq MC_p$ else it would contradict the fact that p is simplicial. Thus all $MC_i \in MCL_p$ have been removed except one. Consequently, $|MCL_p| = 1$, the while loop in step 2 will terminate, and p will correctly send its simplicial message. □

Theorem 6 *The time complexity of the AlgPEO on the vertex processors is $O(d^2c)$ plus some possible wait time, where $d = max(d_p)$ and $c = max(c_p)$ in G.*

Proof: The loop of step 2 can be executed a maximum of d_p times before $|MCL_p| = 1$, since only d_p elements make up the MCs. The set operations of steps 2(b) and 2(d) may require d_p time. Step 2(b) may also require that each $MC \in MCL_p$ be checked and so may require c_p checks. Step 2(d) will require at most $c_p - 1$ checks. All other steps take constant time. Thus the AlgPEO has a running time of $O(d^2c)$ plus whatever wait time is incurred at step 2(a). □

The following theorem will be proven for synchronous distributed memory models. Order of magnitude results will hold for any reasonable asynchronous model of computation.

Theorem 7 *The overall time complexity of AlgPEO, including message wait time is $O(d^2C)$ where C is the total number of maximal cliques in G and $d = max(d_p)$.*

Proof (sketch): To see this, we can look at the combined elimination sequence (for all processors) as a sequence of removed cliques. The first clique would be removed at time 1 the second at time 2 (or earlier) etc. Thus we would have C clique removals. If our actual elimination sequence removed several cliques at once, then the time to remove the cliques is less than C.

Now after each removal of a clique, some number of vertices become simplicial. Let v be one of the vertices that is about to become simplicial. Using our worst case model, v can only have

two cliques remaining: the clique that was just removed and the one in which it will be simplicial. In order to determine that v is simplicial, it's processor will have to receive, at most, d_p messages and use, at most, $2d_p$ steps to remove each of those vertices from one of its two remaining maximal cliques. Each processor corresponding to a simplicial vertex then sends, at most, d_p messages. Thus we have C clique removals each of which may require $O(d^2)$ steps for a time complexity of $O(d^2C)$. $\square$

5. Conclusions

The new parallel algorithms presented here have several practical advantages over results presented elsewhere. One advantage is a faster or comparable running time for sparse graphs. Specific comparisons to some other work are difficult because problems are approached in different orders. In [7], maximal cliques are found in $O(n^4 log^4 n)$, and in [4], maximal cliques are found in $O(n^3 log^2 n)$. The algorithms in this paper are approached differently and therefore the running time is based on the degree of each vertex. The work performed by the maximal clique algorithm in this paper is $O(nd^4)$ which is obviously better for sparse graphs. The algorithm to find a PEO presented in [7] requires $O(n^2 log^2 n)$ work to find a PEO using the set of maximal cliques. The algorithm in this paper requires $O(nd^2c)$ work, again a significant reduction for sparse graphs. The algorithms also achieve equal speedup as processors are added (up to n).

Comparisons to the work in [6] are complicated by the fact that a PEO is found before finding maximal cliques. However, assuming that we are finding both, the algorithm in [6] will take $O((n+m)log^2 n)$ work on a CRCW PRAM. If we assume that we have a sparse graph with $d_v \leq log\ n\ \forall v \in G$ and we also take away the power of the concurrent read (adding $O(log\ n)$ to the time complexity of [6]) then we have $O(nlog^4 n)$ work in [6] and $O(nlog^4 n)$ work for the algorithms presented here. However, in [6] (and in all of the other above mentioned work) communication delays are not considered, while they are included in the analysis included here.

Future work in this area will be in three directions. First, we want to examine the possibility of using some sort of data structure other than a list to hold the candidate cliques. The hope is that this data structure would enable us to refine the list and determine non-maximal cliques in $O(d^2)$ time rather than $O(d^3)$. Second, we wish to explore the possibility of developing portions of a PEO on a single processor from the data structure used for developing maximal cliques. Third, we want to examine the possibility of developing a clique tree simultaneous with the development of a PEO with no increase in running time.

References

[1] C. Beeri, R. Fagin, D. Maier and M. Yannakakis, "On the desirability of acyclic database schemes," *Journal of the Association for Computing Machinery* vol. 30, no. 3, pp. 205-212, 1974.

[2] C. Papadimitriou and M. Yannakakis "Towards an Architecture Independent Analysis of Paralllel Algorithms," *SIAM Journal of Computing*

[3] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, London, 1980.

[4] C. Ho and R. C. T. Lee, "Efficient parallel algorithms for finding maximal cliques, clique trees, and minimum coloring on chordal graphs," *Information Processing Letters* Vol. 28, pp. 301-309, 1988.

[5] C. Ho and R. C. Lee, "Counting clique trees and perfect elimination schemes in parallel," *Information Processing Letters* Vol. 31, pp. 61-68, 1989.

[6] P. N. Klein, "Efficient parallel algorithms for chordal graphs," *Symposium on Foundations of Computer Science*, pp. 150-161, October 1988.

[7] J. Naor, M. Naor, A. A. Schaffer, "Fast parallel algorithms for chordal graphs," *Proceedings 19th Annual ACM Symposium on Theory of Computing*, pp. 355-364, 1987.

[8] D. J. Rose, "Triangulated Graphs and the Elimination Process," *Journal of Mathematical Analysis and Applications*, Vol. 32, pp. 597-609, 1970.

[9] D. J. Rose, "A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations," in *Graph Theory and Computing*, ed. R. Read, pp. 183-217, Academic Press, New York, 1973.