A new augmentation based algorithm for extracting maximal chordal subgraphs

Sanjukta Bhowmick\textsuperscript{a}, Tzu-Yi Chen\textsuperscript{b}, Mahantesh Halappanavar\textsuperscript{c,}\textsuperscript{*}

\textsuperscript{a} University of Nebraska, PKI 1110 South 67th St, Omaha, NE, 68182, United States
\textsuperscript{b} Pomona College, 185 E. Sixth St, Claremont, CA 91711, United States
\textsuperscript{c} Pacific Northwest National Laboratory, PO Box 999, MSIN: J4-30, Richland, WA 99352, United States

\textbf{ABSTRACT}

A graph is chordal if every cycle of length greater than three contains an edge between non-adjacent vertices. Chordal graphs are of interest both theoretically, since they admit polynomial time solutions to a range of NP-hard graph problems, and practically, since they arise in many applications including sparse linear algebra, computer vision, and computational biology. A maximal chordal subgraph is a chordal subgraph that is not a proper subgraph of any other chordal subgraph. Existing algorithms for computing maximal chordal subgraphs depend on dynamically ordering the vertices, which is an inherently sequential process and therefore limits the algorithms’ parallelizability.

In this paper we explore techniques to develop a scalable parallel algorithm for extracting a maximal chordal subgraph. We demonstrate that an earlier attempt at developing a parallel algorithm may induce a non-optimal vertex ordering and is therefore not guaranteed to terminate with a maximal chordal subgraph. We then give a new algorithm that first computes and then repeatedly augments a spanning chordal subgraph. After proving that the algorithm terminates with a maximal chordal subgraph, we then demonstrate that this algorithm is more amenable to parallelization and that the parallel version also terminates with a maximal chordal subgraph. That said, the complexity of the new algorithm is higher than that of the previous parallel algorithm, although the earlier algorithm computes a chordal subgraph which is not guaranteed to be maximal.

We experimented with our augmentation-based algorithm on both synthetic and real-world graphs. We provide scalability results and also explore the effect of different choices for the initial spanning chordal subgraph on both the running time and on the number of edges in the maximal chordal subgraph.

© 2014 Elsevier Inc. All rights reserved.

1. Introduction

A graph is chordal if every cycle of length greater than three contains an edge between non-adjacent vertices. This class of graphs is of both theoretical and practical interest. Theoretically, many problems that are NP-hard on general graphs can be solved in polynomial time on chordal graphs. In practice, chordal graphs play a role in many applications ranging from sparse linear solvers [21] to computer vision [7] to computational biology [11].

However, graphs modeled from real-world applications are rarely perfectly chordal. Nevertheless, if an appropriate chordal subgraph can be found, that subgraph can be useful in a variety of ways. For example, studying that subgraph can reveal interesting properties about the application; an example is the biology study in [11]. As another example, when solving a sparse linear system $Ax = b$ for the vector $x$, it can be desirable to first apply a preconditioner to improve the behavior of a subsequently applied iterative solver. Using a preconditioner based on computing a chordal subgraph has advantages since the preconditioner can then be factored without fill [8]. More generally, chordal graphs are an important tool used in understanding the space requirements of direct methods for solving sparse linear systems; for a thorough discussion see [4].

Unfortunately, for general graphs, computing the chordal subgraph which has the maximum number of edges (the maximum chordal subgraph) is an NP-hard problem. However, there are known polynomial-time algorithms for computing maximal...
chordal subgraphs, where a maximal (also called edge-maximal) chordal subgraph is a chordal subgraph which is not a proper subgraph of any other chordal subgraph.

Current algorithms for computing maximal chordal subgraphs are based on algorithms for recognizing chordal graphs. In order to achieve maximality, these algorithms require processing the vertices in a certain order, specifically a reverse perfect elimination order (as discussed in Section 2). This requirement that the vertices have to be processed in a particular order imposes serialization. But as researchers find an increasing number of applications to which chordal graphs are relevant, and as the graphs of interest grow in size, there is a need for alternative parallelizable approaches for computing maximal chordal graphs.

Previous work describes a parallel algorithm for computing maximal chordal subgraphs that is based roughly on existing sequential order-based algorithms [14]. However, as shown in Section 3, the chordal subgraph computed by the algorithm in [14] is not guaranteed to be maximal.

In this paper we propose a new algorithm for computing maximal chordal subgraphs. Instead of iterating over the vertices, the algorithm begins with an initial spanning chordal subgraph and then repeatedly adds edges, stopping when no more edges can be added. In Section 4, we prove that the algorithm correctly terminates with a maximal chordal subgraph and analyze its running time. We describe in Section 5 how the algorithm can be parallelized and in Section 6 how the algorithm can be parallelized. We then present experimental results from running the parallel algorithms on a variety of synthetic and real-world graphs in Section 7. In addition to looking at scalability, we also look at the impact of the initial spanning chordal subgraph on the computed maximal chordal subgraph.

Contributions: The main contributions of this paper are: (i) a new parallel algorithm for computing maximal chordal subgraphs; (ii) a detailed proof of correctness and complexity analysis of the new algorithm; (iii) a proof that a previously published parallel algorithm does not achieve maximality under certain circumstances; and (iv) parallel experiments showcasing the performance of the proposed augmentation algorithm and its comparison to a serial order-based method.

2. Background and related work

In this section we provide the necessary background and notation used in this paper. We will also provide a brief discussion of related work.

Let $G = (V, E)$ be an undirected graph with a vertex set $V$ and an edge set $E$. In this paper, we assume that $G$ is connected and has neither self-loops (edges that have the same vertex as its two endpoints) nor isolated vertices (vertices with zero edges incident on them). A walk is a sequence of edges such that any two consecutive edges in the walk are incident on the same vertex. A path is a walk that does not visit an edge more than once. We will denote a path from vertex $u$ to vertex $v$ by $P_{u \rightarrow v}$. A cycle is a path that starts and ends with the same vertex. A chord is an edge that connects any two non-adjacent vertices in a cycle.

A graph is chordal if every cycle longer than three edges in the graph has a chord. In other words, the longest induced cycle in a chordal graph is of length three edges. A chordal subgraph of $G = (V, E)$ is a graph $C = (V', E')$ where $E' \subseteq E$ and $C$ is chordal. A maximum chordal subgraph is a chordal subgraph of $G$ that contains at least as many numbers of edges as any other chordal subgraph of $G$. A chordal subgraph $C$ is maximal if there does not exist an edge $(u, v) \in E \setminus E'$ such that $C' = (V', E' \cup \{u, v\})$ is also chordal.

An ordering, or permutation, of the vertices in $G$ is a bijection $\sigma : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$, where $n = |V|$. If $v_{\sigma(1)}$, $v_{\sigma(2)}$, \ldots, $v_{\sigma(n)}$ is an ordering of the vertices in $G$, then $v_j$ is a successor of $v_i$ if $\sigma(j) > \sigma(i)$ and $(v_i, v_j) \in E$. The first successor of a vertex $v$ is the successor $v_i$ that has the smallest index. A clique in $G$ is a subset of vertices such that every two vertices in this subset are connected by an edge. A perfect elimination ordering (PEO) in $G$ is an ordering of vertices such that for every vertex $v$, $v$ and the neighbors of $v$ that exist after $v$ in the ordering form a clique. A set of vertices $S \subseteq V$ is a $u$-$v$ separator if $u$ and $v$ are in separate components of the subgraph induced by $V \setminus S$. A minimal $u$-$v$ separator is a $u$-$v$ separator.

Chordal graphs, perfect elimination orderings, and vertex separators are related by the following theorem from [13]:

**Theorem 1** ([13]). Let $G$ be a graph. The following statements are equivalent:

- $G$ is a chordal graph.
- $G$ has a perfect elimination ordering.
- Every minimal vertex separator induces a complete subgraph of $G$.

The neighborhood of $v$ is denoted by $Adj_G(v) = \{u \in V | (u, v) \in E\}$. A graph $G[X]$ is the subgraph of $G$ induced by $X \subseteq V$. We denote the set of vertices that are adjacent to both $u$ and $v$ in the graph $G$ by $I_{u,v} \equiv Adj_G(u) \cap Adj_G(v)$.

A tree is an undirected graph with no cycles such that any two vertices are connected by exactly one simple path. A connected graph $G = (V, E)$, a spanning tree $G_T = (V, E')$ of $G$ is a subgraph that includes all the vertices of $G$ and is also a tree. Furthermore, for a weighted graph $G = (V, E, w)$, $w : E \rightarrow \mathbb{R}^+$, a minimum spanning tree is a spanning tree with weight less than or equal to all other spanning trees of $G$, where the weight of a spanning tree is defined as the sum of weights of all the edges in the tree. A maximum spanning tree is defined similarly. Since a tree has no cycles, it is clear that any spanning tree of a graph is also chordal.

While extracting a maximum chordal subgraph is NP-hard, maximal chordal subgraphs can be extracted in polynomial time as shown by Dearing et al. [9] in the following subsection, we discuss the Dearing et al. algorithm for computing maximal chordal subgraphs in order to give context for the new algorithms that we propose in this paper.

We note that considerable work has been done on the related problem of determining how few edges can be added to a graph to make it chordal. This is known as the minimum triangulation, or the minimum fill-in, problem. While known to be NP-hard [26], the minimal triangulation problem is tractable and surveys of sequential and parallel algorithms include [3,15].

2.1. Sequential maximal chordal subgraph algorithms

Existing algorithms for computing a maximal chordal subgraph of a graph use two facts. The first is that a graph is chordal if and only if there exists a perfect-elimination order (PEO) on the vertices [20]. The second, stated in Theorem 2, is that $C$ is maximal if no single edge $e \in G \setminus C$ can be added to $C$ such that $C \cup \{e\}$ is also chordal.

**Theorem 2** (Rose–Tarjan–Lueker [22]). Given a graph $G = (V, E)$, $C \subseteq G$ is a maximal chordal subgraph if and only if:

1. $C$ has a perfect elimination ordering $(v_1, v_2, \ldots, v_n)$, and
2. for all edges $(v_i, v_j) \in G \setminus C$ with $i < j$, the subgraph of $C \cup (v_i, v_j)$ induced by the vertices $v_1, v_2, \ldots, v_j$ is not chordal.

Existing algorithms for computing maximal chordal subgraphs are order-based in that they create a chordal subgraph $C$ by repeatedly choosing vertices in a reverse perfect elimination order, then adding as many edges as possible to the chordal subgraph given that choice of vertex. Pseudocode for a generic sequential
Algorithm 1 A generic order-based algorithm for extracting a maximal chordal subgraph. **Input:** A graph $G = (V, E)$. **Output:** A maximal chordal edge set $E_C$. **Data structures:** A vector $C$ of size $|V|$ to track the chordal set of each vertex.

1: procedure OrderBased($G(V, E)$)
2: \[ E_C \leftarrow \emptyset \]
3: for all $v \in V$ do
4: \[ C[v] \leftarrow \emptyset \]
5: Select initial vertex $v$
6: $S \leftarrow v$
7: while $|S| < |V|$ do
8: for all $u \in Adj_C(v)$ do
9: \[ \text{CheckCHORDAL}(v, u, S, C, E_C) \]
10: Select $v$ according to predefined criteria
11: $S \leftarrow S \cup \{v\}$

algorithm for computing a maximal chordal subgraph in this fashion is given in Algorithm 1.

In the pseudocode the set $S$ consists of vertices that have already been ordered and the set $E_C$ consists of edges that have been selected to be part of the chordal subgraph. The CheckCHORDAL subroutine ensures that for every vertex $v$, the set $C[u]$ is a subset of the already ordered vertices that are adjacent to $v$ and that form a clique in $E_C$. Alternatively, CheckCHORDAL is verifying that $v$ is a valid first successor of $u$ in a perfect elimination ordering of the chordal subgraph that is being constructed, as required by Theorem 3.

**Theorem 3** (Rose–Tarjan–Lueker [22]). An ordering $(v_1, v_2, \ldots, v_n)$ is a perfect elimination ordering of a chordal graph if and only if for all $i$, the first successor of $v_i$ is adjacent to all subsequent successors of $v_i$.

Dearing et al. [9] prove that this scheme results in a maximal chordal subgraph if the selection of a vertex in Line 10 of Algorithm 1 is based only on the edges included so far in the chordal subgraph and if the selection process follows a perfect elimination ordering such as those described in [22,24,23,18]. Theorem 2 can then be used to show that the subgraph is maximal chordal since the for all loop (Lines 8–9) ensures that there are no edges in $E \backslash E_C$ that could have been added while maintaining chordality for that particular ordering.

The algorithm in [9] chooses vertices based on the maximum cardinality algorithm in [23]. In particular, the algorithm in [9] repeatedly chooses an unordered vertex $v$ that is as connected as possible to the partial chordal subgraph $C$. The algorithm in [2] chooses the vertices using the lexicographic ordering in [22]. Both algorithms can be implemented to run in $O(|E| \Delta)$ time, where $\Delta$ is the maximum degree of a vertex in $G$. A slight variation consisting of essentially a one-step lookahead version of [9] is described in [25].

Since order-based algorithms depend on a particular ordering of vertices determined during execution, they are inherently serial. For vertices with large neighborhoods, iterations of the inner loop in Lines 8–10 of Algorithm 1 can be executed concurrently. However, for sparse graphs there is not enough concurrency for an efficient implementation.

### 3. Previous parallel chordal subgraph algorithm and proof of non-maximality

We now describe the parallel algorithm of Halappanavar et al. [14]; this is, to the best of our knowledge, the only parallel chordal subgraph algorithm targeting multi-threaded architectures. We then show that this algorithm is not guaranteed to find a maximal chordal subgraph.

**3.1. A parallel chordal subgraph algorithm**

For the sake of completeness, in Algorithm 2 we reproduce the pseudocode for the algorithm proposed by Halappanavar et al. While we refer the reader to [14] for details, we provide a brief explanation here in order to highlight why a chordal subgraph computed by this algorithm may not be maximal for some inputs.

**Algorithm 2** Maximal Chordal Subgraph Algorithm from Halappanavar et al. **Input:** A graph $G$. **Output:** A maximal chordal edge set $E_C$. **Data structures:** A vector $LP$ of size $|V|$ to store the lowest parent of each vertex. A vector $C$ of size $|V|$ to track the chordal set of each vertex.

1: procedure Max-Chordal($G(V, E)$)
2: \[ Q_1 \leftarrow \emptyset \]
3: \[ Q_2 \leftarrow \emptyset \]
4: for all $v \in V$ in parallel do
5: \[ w \leftarrow \text{lowest parent of } v \]
6: \[ \text{if } w \neq \emptyset \text{ then} \]
7: \[ LP[w] \leftarrow w \]
8: \[ \text{if } w \neq Q_1 \text{ then} \]
9: \[ Q_1 \leftarrow Q_1 \cup \{w\} \]
10: \[ C[v] \leftarrow \emptyset \]
11: while $Q_1 \neq \emptyset$ do
12: for all $v \in Q_1$ in parallel do
13: for all $w \in \text{Adj}_C(v)$ do
14: \[ \text{if } LP[w] = v \text{ then} \]
15: \[ \text{if } C[w] \subseteq C[v] \text{ then} \]
16: \[ C[w] \leftarrow C[w] \cup \{v\} \]
17: \[ E_C \leftarrow E_C \cup \{e_{v,w}\} \]
18: \[ x \leftarrow \text{next lowest parent of } w \]
19: \[ \text{if } x \neq \emptyset \text{ then} \]
20: \[ LP[w] \leftarrow x \]
21: \[ \text{if } x \neq Q_2 \text{ then} \]
22: \[ Q_2 \leftarrow Q_2 \cup \{x\} \]
23: \[ Q_1 \leftarrow Q_2 \]
24: \[ Q_2 \leftarrow \emptyset \]

The central idea of Algorithm 2 is to exploit parallelism based on arbitrary ordering of the vertices. The vertices have identifiers from $1$ through $|V|$ and use those identifiers as the ordering. Each vertex first identifies its smallest neighbor, also smaller than itself, as its lowest parent (stored in vector $LP$; Line 7). Furthermore, all the identified parent vertices are placed in a queue (Line 9). If no such parent exists, then the lowest parent is set to zero. All vertices in the queue are then processed in parallel — the size of the queue determines the amount of work that can be executed concurrently.

For each vertex in the queue, only those neighbors that identify that vertex as their lowest parent are processed. Since a vertex can have only one vertex as its lowest parent, conflicts are avoided. However, note that the vertices are processed in some arbitrary order.

We now show that the algorithm of Halappanavar et al. fails to guarantee maximality for certain inputs. In order to illustrate the problem consider the following situation. If executed sequentially, an edge $(i, j)$ will be considered in Algorithm 2 after an edge $(k, l)$ (assuming $i < j$ and $k < l$) if $i$ is greater than $k$, regardless of the relationship between $j$ and $l$. As a result, the algorithm is guaranteed to have found the maximal chordal subgraph using vertices $1 \ldots k$ before it considers $i$. However, in the parallel version, it is now possible for the edges $(i, j)$ and $(k, l)$ to be considered in the reverse order if $i$ appears earlier in $k$'s adjacency list than $j$ appears in $i$'s adjacency list.
Theorem 4. Given a graph \( G \), Algorithm 2 is not guaranteed to find the maximal chordal subgraph.

Proof. Consider the graph in Fig. 1. Algorithm 2 will add edges \((5, 3), (3, 1), (1, 2), (2, 4)\) in the first iteration of the for all loop in lines 12–22. At the beginning of the second iteration, \( Q_2 \) will contain vertices 2, 3, 4. In the second iteration all three remaining edges will be considered, however, if they are not considered in the order \((2, 3), (3, 4), (5, 4)\), some of the edges will not be added. Regardless, at the end of the second iteration the structure \( Q_3 \) will be empty and the algorithm will terminate. The correct output would be the entire graph.

As a corollary to this problem we see that the algorithm can also lead to disconnected graphs. Consider a line graph with edges \((1, 3), (3, 4)\) and \((4, 2)\). Even in the serial case, first edge \((1, 3)\) will be added to the chordal graph, then edges \((4, 2)\) will be added. However, edges \((3, 4)\) cannot be added since the chordal neighbors of vertex 4 will not be a subset of the chordal neighbors of vertex 3. Thus the graph becomes disconnected.

4. Serial augmentation-based algorithm

As the example from the previous section suggests, parallelizing a vertex-ordering-based sequential algorithm for computing a maximal chordal subgraph is challenging. We now describe a new algorithm that is based on repeatedly augmenting a spanning chordal subgraph by adding edges that are guaranteed to maintain chordality. Because this algorithm does not depend on vertex orderings, it is more naturally suitable for parallelization. Algorithm 3 gives pseudocode for this novel augmentation-based algorithm.

Algorithm 3 Sequential Maximal Chordal Subgraph Algorithm.

Input: A connected graph \( G = (V, E) \). Output: A maximal chordal subgraph \( C = (V, E_C) \).

1. procedure AugBasedSerial\((G)\)
2. \( C = (V, E_C) \leftarrow \) a chordal subgraph of \( G \)
3. \( Q \leftarrow \emptyset \)
4. for all \( v \in V \) do
5. \( \text{ENQUEUE}\text{EDGES}(v, E, E_C, Q) \)
6. while \( Q \neq \emptyset \) do
7. \( Q \leftarrow Q \setminus \{(a, b)\} \)
8. \( v \leftarrow \text{any vertex} \in I_{C,a,b} \)
9. \( X \leftarrow \text{Adj}_C(v) \setminus E_{C,a,b} \)
10. if \( |X| \geq 1 \) then
11. \( E_C \leftarrow E_C \cup \{(a, b)\} \)
12. \( \text{for all} \ v \in \{(a, b)\} \)
13. \( \text{ENQUEUE}\text{EDGES}(v, E, E_C, Q) \)

The algorithm begins by initializing \( C \) to a spanning chordal subgraph of \( G \) in line 2. While this could be any spanning chordal subgraph, a natural choice is a spanning tree since it is both guaranteed to be chordal and straightforward to compute, for example through a breath-first search. Furthermore, in the case of weighted graphs, initialization with a minimum (or a maximum) spanning tree will aid in the computation of a weighted maximal chordal subgraph.

The data structure \( Q \) then maintains the set of edges that are not in \( C \) that are potential candidates for adding to \( C \) without affecting the subgraph’s chordality. In order to determine eligible edges, we look for potential triangles incident on vertices, such that an eligible edge is not already part of the chordal subgraph and has both of its end-points incident on a vertex via edges in the chordal subgraph. Formally, given a vertex \( v \), all the edges \((a, b) \in E \setminus E_C\) are eligible if \((a, b) \in \text{Adj}_C(v)\). Note that the neighborhoods are considered in the chordal subgraph.

In lines 6–13 the algorithm then repeatedly removes an edge \((a, b)\) from \( Q \) and checks whether \((a, b)\) can be added to \( C \) while maintaining chordality. This check is done by looking for a path from \( v \) to \( b \) in the subgraph of \( C \) that is induced by \( X = \text{Adj}_C(v) \setminus I_{C,a,b} \) where \( v \) is any vertex in \( I_{C,a,b} \). If \((u, v) \) can be added, \( C \) is augmented by that edge, and any additional edges that are now also candidates for inclusion in \( C \) are added to \( Q \) in lines 12–13.

Fig. 2 illustrates the execution of Algorithm 3 on a simple graph in which no edges are added to \( C \) in lines 12–13.

4.1. Proof of correctness

To prove correctness we use the following theorem from [17], a paper which explores methods for generating chordal graphs.

Theorem 5 (Theorem 2, [17]). Let \( G = (V, E) \) be a connected chordal graph and let \( u \) and \( v \) be any two vertices in \( V \) such that \((u, v) \notin E\). The augmented graph \( G' = (V, E \cup \{(u, v)\}) \) is chordal if and only if \( G[V \setminus I_{C,u,v}] \) is not connected.

Proof. To show that if \( G[V \setminus I_{C,u,v}] \) is not connected then \( G' \) is chordal, we observe that if \( G[V \setminus I_{C,u,v}] \) is not connected, then \( I_{C,u,v} \) is a minimal \( u-v \) separator and it forms a clique in \( G \) (Theorem 1). Furthermore, \( I_{C,u,v} \cup \{u, v\} \) forms a clique in \( G' \). Now consider the minimal separator in \( G' \) between any pair of vertices \( y \) and \( z \). The minimal separator in \( G' \) can only be different from the minimal separator in \( G \) if one, say \( y \), is in the same component as \( u \) and \( z \) is in the same component as \( v \). However, if the minimal separator has changed, it must still be a subset of \( I_{C,u,v} \cup \{u, v\} \). Since \( I_{C,u,v} \cup \{u, v\} \)
forms a clique in $G'$, all its subsets are also cliques. Therefore, by Theorem 1, $G'$ is chordal.

To show that if $G'$ is chordal then $G[V \setminus \{u, v\}]$ is not connected, we show the contrapositive. If $G[V \setminus \{u, v\}]$ is connected, then there is a path from $u$ to $v$ in $G[V \setminus \{u, v\}]$. Furthermore, this path has length greater than 2 since we removed all vertices in $I_{C,u,v}$. This path, plus the edge $(u, v)$, is a chordless cycle of length at least 4 in $G$. □

Further, the authors of [17] add that it is sufficient to search for a path from $u$ to $v$ in $G[V \setminus \{u, v\}]$. Indeed, they assert this search can be limited to the subgraph $G[V \setminus \{u, v\}] \cap Adj(x) = G[Adj(x) \setminus \{u, v\}]$, for any $x \in I_{C,u,v}$. We first state and prove Lemma 1, then state and prove that the search can be limited to such a subgraph in Theorem 6.

**Lemma 1** (Lemma 1, [17]). Let $G = (V, E)$ be a connected chordal graph. If there is a non-empty path $P_{u,v}$, between $u$ and $v$ with minimum length in $G[V \setminus \{u, v\}]$, then $\{w\} \cup I_{C,u,v}$ is a clique in $G$ for all $w \in P_{u,v}$.

**Proof.** If $I_{C,u,v} = \emptyset$, then this is clearly true. Otherwise, for any vertex $t \in I_{C,u,v}$, the following is a cycle: $(t, u), P_{u,v}, (v, t)$. Since $G$ is chordal and $P_{u,v}$ has minimum length, every vertex in $P_{u,v}$ must have an edge to $t$. But this holds for every vertex in $I_{C,u,v}$, and $I_{C,u,v}$ is a clique itself since it is a subset of a minimal $u$-$v$ separator. Therefore $\{w\} \cup I_{C,u,v}$ is a clique in $G$ for all $w \in P_{u,v}$. □

**Theorem 6** (Stated Following Proof of Lemma 1 in [17]). Let $G = (V, E)$ be a connected chordal graph and let $u$ and $v$ be any two vertices in $V$ such that $(u, v) \notin E$. The augmented graph $G' = (V, E \cup (u, v))$ is chordal if and only if $G[Adj(x) \setminus \{u, v\}]$, for any $x \in I_{C,u,v}$, is not connected.

**Proof.** By Lemma 1, for any minimal path $P(u, v)$ in $G[V \setminus \{u, v\}]$, every vertex $x \in I_{C,u,v}$ is adjacent to every vertex in $P(u, v)$. Therefore, to determine if $G[V \setminus \{u, v\}]$ is connected, it is sufficient to look for a path in $G[Adj(x) \setminus \{u, v\}]$, for any $x \in I_{C,u,v}$. □

Now, using Lemma 1 and Theorems 5 and 6, we prove that Algorithm 3 returns a chordal subgraph that is guaranteed to be maximal.

**Theorem 7.** Given a connected graph $G = (V, E)$, Algorithm 3 returns a maximal chordal subgraph $C$.

**Proof.** $C$ is a chordal subgraph by repeated application of Theorem 5.

We prove $C$ is also a maximal chordal subgraph by contradiction. Assume $C = (V, E_C)$ is chordal but not maximal. Then by Theorem 2 there is an edge $(u, v) \in E \setminus E_C$ such that $C + (u, v)$ is also chordal. In other words, there is no path from $u$ to $v$ in $G[V \setminus \{u, v\}]$, and there exists a vertex $z$ such that $z \in I_{C,u,v}$. Therefore, at some point (and possibly more than once) the edge $(u, v)$ was in the queue $Q$.

Let $C'$ be the subgraph when the edge $(u, v)$ was last removed from $Q$ and considered for addition to the chordal subgraph. Since $(u, v)$ was not added at that point, there was a path from $u$ to $v$ in $C'[V \setminus \{u, v\}]$. But $(u, v)$ is reevaluated for addition to $Q$ after adding any edge adjacent to either $u$ or $v$, so $I_{C,u,v} = I_{C,u,v}$. Therefore $C'[V \setminus \{u, v\}]$ and $C[\setminus \{u, v\}]$ are subgraphs induced by the same set of vertices. Since the edges in $C'$ are a subset of the edges in $C$, it is impossible for a path from $u$ to $v$ to exist in $C'[V \setminus \{u, v\}]$ but not in $C[\setminus \{u, v\}]$. Therefore, there is no such edge $(u, v)$, and $C$ is a maximal chordal subgraph. □

### 4.2. Analysis of run time

Given a graph $G = (V, E)$, let $n$ and $m$ denote the number of vertices and the number of edges respectively. We will use $\Delta$ to represent the largest degree in $G$, and $\delta(v)$ to denote the degree of vertex $v$. For the following analysis, we assume an adjacency-list, or an equivalent representation that allows $O(m)$ enumeration of edges for $G$. Further, a hash table with key $(i, j)$, where the value for each entry consists of the state of that edge (either in the chordal subgraph or not) and a common neighbor of vertices $i$ and $j$, if one is known. Finally, the data structure $Q$ can be implemented as a standard queue with $O(1)$ insertion and deletion.

Constructing the initial spanning tree of the connected graph takes $O(m)$ time using a breadth first search. Identifying edges to add to $Q$, and adding them, in lines 4–5 takes time $O(n \Delta^2)$, where $\Delta$ is the maximum degree of a vertex in $G$. Given that $m = \sum_v \delta(v)$, we can further bound the time to $O(m \Delta)$.

Each iteration of the while loop takes time $O(\Delta^2)$ if the hash table allows $O(1)$ time to check the state of an edge: it takes $O(\Delta)$ time to form $I_{C,a,b}$ for lines 10–11. In order to efficiently check if a path exists between two vertices $a$ and $b$, $P_{a \rightarrow b}$ in $C[Adj_c(v) \setminus \{a, b\}]$, we can do a breadth-first search starting at vertex $a$. From a given vertex $v \in I_{C,a,b}$, build the frontier with vertices in $Adj_c(v) \setminus I_{C,a,b}$. In order to accelerate the process, $v$ can be chosen such that it has the lowest number of neighbors. Existence of $b$ in the next frontier (neighbors of vertices identified in the previous frontier) will validate the existence of a path from $a$ to $b$, and, therefore, adding the edge $(a, b)$ will violate the chordality of the subgraph. It can be noted that the breadth-first search from $a$ will always be restricted to a length of three, and one frontier set needs to be carefully built (or considered). Thus, the computational time can also be $O(n \Delta^2)$ where $\Delta = \Delta^2$, where $\Delta^2 \leq 4 \Delta$ is the maximum degree in $C$.

To bound the number of iterations of the while loop, note that at worst an edge $(u, v)$ is evaluated for addition to $Q$ any time an edge adjacent to either $u$ or $v$ is removed from $Q$ (in practice only evaluated if the edge removed from $Q$ is added to $C$). Therefore the number of overall insertions is bound by $O(m \Delta^2)$, which also bounds the total number of iterations.

Hence the total running time is $O(m \Delta^2)$.

### 5. Optimization of the augmentation-based algorithm

While the augmentation-based method in Algorithm 3 computes a maximal chordal graph without using a vertex ordering, the algorithm is computationally expensive in large part because a single edge can be inserted multiple times into $Q$. This can happen even if the edge never gets added to the chordal edge set $C$.

In this section we describe various strategies for improving the performance of Algorithm 3. First, we provide a strategy to reduce the number of times an edge is considered as eligible. We then describe a pruning strategy where we identify edges that cannot be part of the final chordal subgraph and which therefore can be removed from further consideration. Finally, we describe strategies for parallelization. Due to the idempotent nature of identifying eligible edges, Algorithm ENQUEUE EDGES PARALLEL can be parallelized by synchronizing the threads that identify the same edge concurrently. However, parallelization of Algorithm ENQUEUE VERTICES is nontrivial. The optimized algorithm including these strategies is given in Algorithm 4.

Minimizing the number of times an edge is processed: Each time a new edge $(a, b)$ is added to the chordal graph, new candidate edges formed from the neighbors of vertex $a$ or vertex $b$ are added to the queue to be checked (Lines 12–13, Algorithm 3). This procedure is called for all the neighbors of $a$ in the original graph every time an edge with the endpoint $a$ is added, even if the requisite edges are
already in the queue. To reduce the redundancy in this procedure we store the vertices whose neighbors have to be considered in a queue (Algorithm 3, procedure EnqueueVertices, Line 9) rather than adding edges to the queue after each new edge is added to the chordal graph. Once we have checked the current set of edges, we call a separate procedure EnqueueVertices to find the next of edges to be checked. Thus, we check the neighbors of a vertex only once per iteration of the main algorithm, instead of multiples times as in the original formulation.

Identifying deletable edges: As presented, the unoptimized algorithm identifies only those edges that can be added to the chordal subgraph without considering if they can be really added. The size of the queue of eligible edges can be considerably reduced if we identify edges that will not be part of the chordal subgraph and prune them from further consideration.

In order to identify such “deletable” edges, we observe that an edge \((a, b)\) is not added to the chordal graph if there exists a path from \(a\) to \(b\), i.e., \(P_{a-b}\). Adding the edge \((a, b)\) will cause a cycle in the chordal graph, that is currently larger than three. However, the edge is kept for further consideration because in the course of augmentation, the cycle might become chorded, and then it might be valid to add \((a, b)\) to the chordal graph.

As part of optimizing the algorithm, we add a method to detect whether such a cycle will ever be chordal or not. To do so, we observe the set of eligible edges that are not part of the chordal subgraph (Algorithm 4, procedure EnqueueVertices, lines 11–12).

**Theorem 8.** Let \(G\) be a graph in which there exists a path \(P_{a-b}\), and let \((a, b)\) be an edge. Then for the graph \(G\) to be chordal there must exist at least one vertex \(v \in P_{a-b}\), such that \(v \in Adj_C(a)\) and \(v \in Adj_C(b)\), i.e. \(v\) is a common neighbor of both \(a\) and \(b\).

**Proof.** We use a proof by contradiction. Let us assume that the graph \(G\) is chordal, but there does not exist any \(v \in P_{a-b}\) which is a common neighbor of \(a\) and \(b\). Let \(v_a \in P_{a-b}\) and \(v_b \in P_{a-b}\) be two neighbors of \(a\) and \(b\) in the path respectively, such that \((i) P_{a-b}\) is a subpath of \(P_{a-b}\), and \((ii)\) apart from \(v_a\) and \(v_b\), there are not vertices in \(P_{a-b}\) that are neighbors of \(a\) and \(b\). As \(a\) and \(b\) have no common neighbors, therefore \(v_a \neq v_b\).

Although the length of \(P_{a-b}\) can be small because the vertices with the path may be connected, the shortest distance will be \(1\), i.e. \(v_a \in Adj_C(v_b)\). This will lead to a cycle of size 4 with the vertices \(a, v_a, v_b, b\). However, this cannot happen since \(G\) is chordal. Therefore \(v_a = v_b\), i.e. \(a\) and \(b\) must have at least one common neighbor.

The size of the initial chordal graph is also an important factor in determining both the size of the maximal chordal graph as well as the execution time. If the graph is nearly chordal, then only a few edges have to be considered and this reduces the time as well. A natural choice would be using the nearly-chordal graph obtained from the order-based method in Halappanavar et al. (Algorithm 2). However, as discussed in Section 3, there is a chance that the obtained chordal graph might be disconnected, even if the original graph is connected. The augmented algorithm does not connect the disconnected pieces, therefore in these cases we will obtain disconnected chordal graphs. These disconnected graphs can then be connected by joining them across a common edge.

### 6. Parallel augmentation-based algorithm

A key challenge in developing a parallel implementation of the sequential algorithm is identifying edges that can be added in parallel. We can do so using a two-step process. First, we find the set of candidate edges in parallel that can be added by considering the fact that they form a triangle. This set is marked as **Viable**. However, if all the edges in the set **Viable** are added simultaneously, then two simultaneously added edges could affect each other as in Fig. 3. Therefore, we also need to identify a subset of the edges in **Viable** that can be added concurrently. This subset is marked as **Addable**. The parallel algorithm in Algorithm 5 gives a template for how the **Viable** and **Addable** sets are determined.

In order to produce a maximal chordal graph, the set **Addable** must satisfy the following necessary and sufficient requirements:

- **Progress requirement:** If there is only a single edge marked **Viable**, then that edge must also be marked **Addable**.
- **Concurrency requirement:** If \(C\) is a chordal graph, adding the set of **Addable** edges to \(C\) must maintain chordality.

**Theorem 9.** Let \(C\) be a chordal graph. If adding edges to **Addable** satisfies both the concurrency and progress requirements, then Algorithm 5 terminates with a maximal chordal subgraph.

**Proof.** It is clear that both conditions are necessary. It is also clear that the algorithm returns a chordal subgraph. We prove by contradiction that the algorithm returns a maximal chordal subgraph. Assume the algorithm returns a graph \(C = (V, E_C)\) that is chordal but not maximal. Then by Theorem 2 there is an edge \((u, v) \in E \setminus E_C\) such that \(C + (u, v)\) is also chordal. That edge would have been marked **Viable** and, by the progress requirement, added.
Algorithm 5 Parallel Maximal Chordal Subgraph Algorithm. Input: A connected graph \( G = (V, E) \). Output: A maximal chordal subgraph \( \mathcal{C} = (V, E_{\mathcal{C}}) \).

1: **procedure** AUGESENTEDPARALLEL\((G(V, E))\)
2: \( \mathcal{C} \leftarrow \text{a spanning chordal subgraph of } G \)
3: \( Q \leftarrow \emptyset \)
4: \( R \leftarrow \{1 \ldots |V|\} \)
5: while \( R \neq \emptyset \) do
6: \( \text{ENQUEUEEDGESPARALLEL}(E, E_{\mathcal{C}}, Q, R) \)
7: \( R \leftarrow \emptyset \)
8: while \( Q \neq \emptyset \) do
9: for all \( (a, b) \in Q \) in parallel do
10: \( \text{Viable}(a, b) \leftarrow \text{false} \)
11: \( X \leftarrow \text{Adj}(v) \setminus I_{C,a,b} \)
12: if \( \exists I_{\text{adj}} \text{in } C[X] \) then
13: \( \text{Viable}(a, b) \leftarrow \text{true} \)
14: for all \( (a, b) \) do and \( \text{Viable}(a, b) = \text{true} \)
15: \( \text{ADDABLE}(a, b) \leftarrow \text{false} \)
16: Identify Set of Edges \((a, b)\) that can be added concurrently.
17: \( \text{ADDABLE}(a, b) \leftarrow \text{true} \) for the set of edges that can be added concurrently.
18: for all \( (a, b) \) and \( \text{ADDABLE}(a, b) = \text{true} \) in parallel do
19: \( E_{\mathcal{C}} \leftarrow E_{\mathcal{C}} \cup (a, b) \)
20: \( R \leftarrow R \cup \{a, b\} \)
1: **procedure** ENQUEUEEDGESPARALLEL\((E, E_{\mathcal{C}}, Q, R)\)
2: \( Q \leftarrow \emptyset \)
3: for all \( v \in R \) in parallel do
4: for all \( (a, b) \in E \setminus (E_{\mathcal{C}}) \) such that \( a, b \in \text{Adj}(v) \) do
5: if \( (a, b) \notin Q \) then \( \triangleright \) Synchronize with competing threads
6: \( Q \leftarrow Q \cup \{(a, b)\} \)

6.1. Creating the addable set

The progress requirement is straightforward to satisfy. More difficult is the concurrency requirement. The reason an edge \((u, v)\) might be viable sequentially, but not addable in parallel is because some other added edge, or edges, might create a path in \( C[V \setminus I_{V}^u_{v}] \), where \( C \) is the chordal graph with some edges added, when one did not exist in \( C[V \setminus I_{V}^u_{v}] \). This can happen for one of two reasons:

- \( I_{V}^u_{v} \neq I_{V}^a_{v} \)
- Another added edge \((a, b)\) creates a path between \( u \) and \( v \) that did not exist in \( C[V \setminus I_{V}^u_{v}] \).

For example, if \( \text{ADDABLE} \) can be any vertex-disjoint subset of edges of Viable, then \( I_{V}^u_{v} = I_{V}^a_{v} \), but two added edges could affect each other if one creates a path from \( u \) to \( v \) on the example in Fig. 3.

That said, there are options for finding the set \( \text{ADDABLE} \). All of them result in a correct parallel algorithm, however three are implications for the running time, both in the number of iterations of the while loop and in the execution cost. One possibility is to add a vertex disjoint set of edges such that there is no vertex \( x \in V \) that is adjacent to all four endpoints of any two added edges (adjacent or equal to). Now for any added edge \((u, v)\), \( C[\text{Adj}(x) \setminus I_{V}^u_{v}] = C[\text{Adj}(x) \setminus I_{V}^a_{v}] \), so if there is no path in the latter then there is no path in the former. In other words, for each pair of edges \((a, b)\) and \((c, d)\); vertices \( c \) and \( d \) are \( k > 2 \) distance away from both vertex \( a \) and vertex \( b \). This can be achieved by applying vertex coloring on the original graph and eliminating all the distance-1 and distance-2 neighbors of the edges in Viable. The remaining edges form the set \( \text{ADDABLE} \).

6.2. Analysis of run time

Given a graph \( G = (V, E) \), recall that \( n \) and \( m \) denote the number of vertices and edges respectively and that \( \Delta \) is the largest degree in \( G \). We let \( p \) denote the number of processing units available.

The first part of the parallel algorithm requires creating a spanning chordal graph. One possibility is to use a tree computed using, say, a parallel breadth-first search such as [1,5,16]. Another possibility is to initialize the algorithm using a chordal subgraph computed using the parallel algorithm in [14]. This algorithm has an upper bound of \( O(\Delta^2) \) in a massively multithreaded environment, where every vertex can be assigned to a thread. Hence, if the number of threads is bounded by \( p \), the upper bound on the running time is \( O(\frac{n^2}{p} \Delta^2) \). We note, however, that in practice the running time of [14] is generally less than the actual number of iterations tends to be much less than the \( \Delta \) used in the computing the bound.

After the initial spanning chordal subgraph is created, identifying the edges to be added is done in parallel. As discussed in Section 4.2, this takes time \( O(\Delta^2) \) for each vertex, so on \( p \) processors the total time for this step is \( O(\frac{n^2}{p} \Delta^2) \). We then identify the Viable set which, as discussed in Section 4.2 takes \( O(\Delta^2) \) for each edge, where \( \Delta_C \leq \Delta \) is the maximum degree in \( C \). Hence, an upper bound on the time needed to determine which edges can be added to Viable is \( O(\frac{n^2}{p} \Delta^2) \). In the next step the algorithm identifies a subset of the edges in Viable that it marks as ADDABLE. A naive implementation could compute a subset where every vertex is at least distance 3 apart in \( C \) using a breadth first search. If every vertex in \( V \) is also in Viable, then an upper bound on this step is \( O(\frac{n^2}{p} \Delta^2) \).

The bound on the number of iterations is difficult to compute, because an edge that is in the set Viable but not put in the set ADDABLE can potentially be added again to the queue later. With the ADDABLE edges chosen to be those whose endpoints are at
least distance 3 from the endpoints of any other edges in \textsc{Addable}, each time an edge is marked viable, but not addable (due to the concurrency requirement) it has the potential to be sent back to the queue again. However, because of the progress requirement we know that at least one edge must be added in each iteration, hence \( m \) is an upper bound on the total number of iterations. Therefore an overall upper bound is \( O(\frac{m^2}{p} \Delta_v^2) \).

7. Experimental results

In this section we discuss the empirical results from executing the parallel augmentation-based algorithm. We examine the scalability of the parallel implementation, investigate how different algorithms for initializing the chordal spanning graph can affect the augmentation process, and compare the parallel augmentation based approach with serial order-based approaches.

Although the most parallel version is Algorithm 5, in Section 6 we discussed the fact that we do not yet have a good method for finding addable edges in parallel. Using vertex coloring repeatedly on the graph is more expensive and can potentially increase the number of iterations. Therefore, in our parallel implementation we create the initial spanning chordal graph in parallel and then add the eligible edges to a queue in parallel. However, selecting the edges to be added is performed in serial, because this is currently more efficient than implementing a parallel distance-2 coloring of the network. We initialize the chordal subgraph in two different ways: first by using a spanning tree computed via a parallel breadth-first search (BFS) tree, and second by using the parallel chordal subgraph algorithm from \[14\].

7.1. Test suite of networks

We tested these algorithms on a combination of synthetic graphs and real-world networks. We summarize some of their properties in Table 1, then describe them in greater detail.

7.1.1. Synthetic networks

We used two classes of synthetic graphs: (i) generated using the RMAT algorithm \[6\], and (ii) generated using the random geometric graph (RGG) model.

The input parameters for RMAT include the size of the graph in terms of the number of vertices and edges and a set of four probabilities that sum to one. In order to randomly determine the edges in this graph, RMAT employs a recursive structure to traverse the adjacency matrix of this graph where the rows and columns represent vertices and the elements of the matrix represent edges (between the two vertices representing the row and the column); duplicates and self-loops are removed. The matrix is recursively divided into four quadrants, based on the four probabilities specified as input, until a particular matrix element is reached. This matrix element forms the new edge in the graph. This procedure is repeated for the desired number of edges. Depending on the probabilities specified, RMAT can generate graphs with a wide range of characteristics.

For our test suite we set the number (SCALE) of vertices to be powers of two and then set the number of edges to 16 times the number of vertices. We created RMAT networks with probabilities \{0.45, 0.15, 0.15, 0.25\} which exhibit a large variation in degree distribution similar to graphs popularly known as scale free small world networks.

In contrast to the RMAT family of synthetic networks, generating a d-dimensional random geometric graph (RGG) requires two input parameters: the number of vertices \( n \) and a threshold distance \( r \). The graph is generated by first placing \( n \) vertices uniformly at random in a d-dimensional space and then connecting all pairs of vertices that are closer than \( r \) in Euclidean distance. For our test suite we used 2-dimensional space with a threshold distance computed as a function of the number of vertices, given by \( r = (r_v + r_t) / 2 \), where \( r_v = \sqrt{\ln n / \pi n} \) and \( r_t = \sqrt{\ln n / \pi} \). The value of \( r_v \) is set to 2.0736 for 2-dimensional graphs. The main goal in choosing a value for \( r \) is to maintain sparsity while creating a giant component in the graph. A detailed treatment of random geometric graphs can be found in the monograph by Penrose \[19\].

Comparison between properties of RMAT and RGG. Graphs generated with these two methods, RMAT and RGG, have significantly different characteristics. To illustrate this we consider two instances with 1024 vertices; one generated with RMAT and the other with RGG using the parameters specified above.

We consider the following graph characteristics that are important for our algorithms: average clustering coefficient, average path length, and the diameter. The local clustering coefficient of a vertex \( v \) is the ratio between the actual number of edges among the neighbors of a vertex, \( adj(v) \), to the total possible number of edges among \( adj(v) \). The average clustering coefficient in the graph is the sum of all local clustering coefficients of vertices divided by the number of vertices. Intuitively, this metric captures the number of triangles, and therefore, the number of edges that are eligible for addition to the chordal edge set in the proposed algorithm. While the average clustering coefficient for RMAT is 0.034, that for RGG is 0.605. Consequently, we should expect that the number of eligible edges, and therefore, runtime, for graphs with low clustering coefficient should be small. In our experiments, we observed a strong

---

Table 1

Properties of the test suite of graphs. The numbers in RGG and RMAT graphs denote the scale, which determines the number of vertices (\( 2^{\text{SCALE}} \)) of the graph. The real-world instances arise from different contexts such as biology, web crawl and ocean metagenomics.

<table>
<thead>
<tr>
<th>Name</th>
<th>Vertices</th>
<th>Edges</th>
<th>Avg degree</th>
<th>Max degree</th>
<th>Std. deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGG-18</td>
<td>1,209,272</td>
<td>262,144</td>
<td>9.2</td>
<td>26</td>
<td>3.05</td>
</tr>
<tr>
<td>RGG-19</td>
<td>2,498,880</td>
<td>524,288</td>
<td>9.5</td>
<td>26</td>
<td>3.07</td>
</tr>
<tr>
<td>RGG-20</td>
<td>1,048,576</td>
<td>5,151,475</td>
<td>9.8</td>
<td>28</td>
<td>3.13</td>
</tr>
<tr>
<td>RGG-21</td>
<td>2,097,152</td>
<td>10,623,000</td>
<td>10.1</td>
<td>33</td>
<td>3.19</td>
</tr>
<tr>
<td>RGG-22</td>
<td>4,194,304</td>
<td>21,848,815</td>
<td>10.4</td>
<td>32</td>
<td>3.23</td>
</tr>
<tr>
<td>RMAT-21</td>
<td>2,097,152</td>
<td>16,777,216</td>
<td>16</td>
<td>877</td>
<td>18.8</td>
</tr>
<tr>
<td>RMAT-22</td>
<td>4,194,304</td>
<td>33,554,432</td>
<td>16</td>
<td>1,105</td>
<td>19.47</td>
</tr>
<tr>
<td>RMAT-23</td>
<td>8,388,608</td>
<td>67,108,864</td>
<td>16</td>
<td>1,297</td>
<td>20.15</td>
</tr>
<tr>
<td>RMAT-24</td>
<td>16,777,216</td>
<td>134,217,728</td>
<td>16</td>
<td>1,756</td>
<td>20.86</td>
</tr>
<tr>
<td>RMAT-25</td>
<td>33,554,432</td>
<td>268,435,456</td>
<td>16</td>
<td>2,093</td>
<td>21.56</td>
</tr>
<tr>
<td>RMAT-26</td>
<td>67,108,864</td>
<td>536,870,912</td>
<td>16</td>
<td>2,631</td>
<td>22.29</td>
</tr>
<tr>
<td>brc a</td>
<td>48,803</td>
<td>687,783</td>
<td>28.1</td>
<td>125</td>
<td>13.5</td>
</tr>
<tr>
<td>ctrl</td>
<td>48,803</td>
<td>1,636,877</td>
<td>67</td>
<td>416</td>
<td>44.34</td>
</tr>
<tr>
<td>cnr</td>
<td>325,537</td>
<td>2,738,969</td>
<td>16.8</td>
<td>18,236</td>
<td>219.15</td>
</tr>
<tr>
<td>gt</td>
<td>1,280,000</td>
<td>102,268,735</td>
<td>159.7</td>
<td>148,155</td>
<td>369.29</td>
</tr>
</tbody>
</table>

---

correlation between the performance and the average clustering coefficient.

The average path length of a graph is the average of the shortest paths between all pairs of vertices. This distance is small for graphs with small-world properties. The diameter of a graph is the longest shortest path between any two vertices. While the average path length in RMAT is 2.94, that in RGG is 18.515; the diameter of RMAT is 6, and that of RGG is 49. Since the eligible edges can be further apart in graphs with longer paths, we expect the augmentation-based algorithm to require more iterations. This is again supported in our experiments.

7.1.2. Real-world networks

We used real-world networks from a range of applications. As a result, they exhibit wide variation in vertex degrees and contain vertices with large degrees. Therefore, we expect significant performance differences for the augmentation-based algorithm across these instances.

Two of our networks are gene correlation networks downloaded from NCBI’s GEO database [12]. One is of cancer-related mutations from normal breast tissue (ctrl) and the other is of non-familial breast cancerous tissue (brca). These networks were built by comparing the Pearson correlation coefficients \( \rho \leq 0.0005 \) of all gene-pairs in each dataset; genes with high correlations \( 0.95 \leq \rho \leq 1.00 \) were connected to form the network. These biological networks exhibit a power law degree distribution and form communities. More detailed descriptions of these input graphs can be found in [14].

The network cnr comes from a small web crawl of the Italian CNR domain and can be accessed from the DIAMCS#10 Challenge website [10]. The network gt is constructed from ocean metagenomics data.

7.2. Hardware platform

All experiments were conducted on a 32-core 4-socket Intel Xeon X7560 platform with 256 GB of system memory. Each socket has 8 cores, and each core has two hardware threads (hyper-threads) leading to a total of up to 64 threads that can be used for experiments. The base clock frequency is 2.266 GHz with a maximum turbo frequency of 2.666 GHz. The processor has 32 KB of L1 cache, 256 KB of L2 cache, and 24 MB of cache. Each socket has 64 GB of DDR3 memory with a peak bandwidth of 34.1 GB per second.

7.3. Implementation details

For experiments with the serial implementation we use the order-based algorithm of Dearing et al. We use a heap data structure to maintain an active list of vertices based on the number of nodes incident on the chordal subgraph at a given time step. Given \( n \) insertions, the cost of inserting, deleting or modifying an element is \( \Theta(\log n) \). The chordal subsets are maintained using BOOST dynamic_bitset data structure.

Our parallel implementation of finding the initial chordal graph was based on (i) the parallel implementation of finding a chordal graph as described in [14], and (ii) a frontier-based level synchronous implementation of parallel BFS. In the parallel BFS, we start with the root node, its neighbors form the initial frontier and can be processed simultaneously. As we go to higher levels, the frontier expands to include more nodes that can be processed in parallel.

The multithreaded algorithms are implemented using OpenMP constructs in the C/C++ language. The queue data structures used are implemented as fixed size (the number of vertices) vectors. We use architecture dependent atomic operator \( \_syn\_fetch_and_add() \) to add elements into the queue. In order to enable efficient read and write into a queue, we maintain two vectors. While we read from one (in parallel), we write into another (using atomic operations). The two vectors are swapped at the end of each iteration, which consists of processing all the vertices enqueued at a certain step of execution. Chordal subsets of every vertex is maintained in a single data structure in a compact format. Note that the maximum size of this set is limited to twice the number of edges in the graph. Atomic operators can be again used to efficiently add new elements to the chordal sets in parallel. In the optimized variant we maintain the chordal sets in order such that the subset conditions can be tested in linear time proportional to the size of the chordal sets. The augmentation-based algorithm is also implemented similarly.

The code was compiled with GNU GCC version 4.8.2 with the options ‘-O3-fopenmp’. For execution, we request for thread pinning via the GOMP_CPU_AFFINITY variable. The threads are scattered to maximize the memory bandwidth. For example, if two threads are requested, then they are placed on two different sockets. If four threads are requested, then two threads are placed on the two sockets, but on different processors. We further request NUMA-aware (non-uniform memory access) memory allocation using the numaclt command with the option ‘interleave = all’.

7.4. Scalability results

We first demonstrate the strong scalability results for the parallel augmented algorithm, based on creating the initial graph using: (i) a spanning tree using a breadth-first search (BFS) on the graph; and (ii) a potentially non-maximal chordal spanning subgraph obtained from [14]. We present strong scaling results by experimenting on two to 32 threads in powers of two. We also provide weak scaling results by experimenting with different scales of inputs. For the synthetic graphs, we create different instances relative to powers of two. While specific connectivity patterns are not guaranteed to be preserved, the general structure will remain the same for the two classes of synthetic graphs used for these experiments.

As discussed in Section 6, while the most parallel version is Algorithm 5, we do not yet have a good parallel method for finding edges to include in the set Addable. Using vertex coloring to find these edges is relatively expensive and can also potentially increase the number of iterations. Therefore, for the parallel experiments described here, we create the initial spanning chordal graph and then add the eligible edges to the queue in parallel. However, for efficiency, the edges for the set Addable are selected serially. We show results from initializing the chordal subgraph in two different ways: using a spanning tree computed using a parallel breadth-first search and using the parallel chordal subgraph algorithm from [14].

In Fig. 5, we show the strong scalability results and the time breakdown for the three parts of the algorithm: (i) parallel implementation of the initial chordal graph; (ii) parallel implementation of the finding the eligible edges; and (iii) sequential code for finding the addable edges. For these results we selected the largest network from each of the three groups (RGG, RMAT and Real-World). As can be seen from the charts, the stage of the computation that requires the most time depends on the input.

For the RGG graphs, the algorithm spends a significant percentage of time in finding the chordal graph and in enqueuing the edges. Therefore, despite the large serial section (process edges), the times gradually decrease as the number of threads increase. The time for processing edges in the RMAT graphs is the least of all the graphs and therefore we get the best scalability on them. The breakdown of time in the gr graph show that the serial part dominates, and therefore the parallel algorithm does not have any
noticeable effect. We also observe that the time to compute the BFS spanning tree is minimal in all the graphs.

In **Fig. 6** we show scalability results for the largest two networks from each group. Since RMAT is the largest in size and the algorithm spends the least proportion of its time in the serial part on these graphs, it is the most scalable. RGG networks are also reasonably scalable. However, because of the amount of time they spent on the sequential part of the algorithm, the real-world graphs demonstrate no scalability. On these real-world graphs, in contrast to the situation with RMAT and RGG graphs, using the chordal graph from [14] as the initial graph leads to better performance than using a BFS tree to initialize the algorithm. The spike in 16 processors for the c2r graph initialized with BFS appears to be an outlier. We believe this is due to a particularly poor vertex ordering in which most of the edges in the queue in each iteration were judged not to be addable and so they were returned to the queue in subsequent iterations. As a result the queue size did not decrease proportionately.
Scaling results of the parallel algorithm for different inputs. The number of threads is along the X-axis, and the compute time (in seconds) is plotted along the Y-axis.

Table 2
Performance of the augmentation-based algorithm when initialized with a potentially non-maximal chordal subgraph from [14]. All results are for runs with eight threads, and the time is in seconds.

<table>
<thead>
<tr>
<th>Input</th>
<th>Num edges</th>
<th>Num iters</th>
<th>Time (s)</th>
<th>Edges added</th>
<th>Num iters</th>
<th>Time to enqueue</th>
<th>Time to process</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>brc</td>
<td>73,441</td>
<td>10</td>
<td>0.10</td>
<td>81,315</td>
<td>5</td>
<td>0.126</td>
<td>0.434</td>
<td>0.672</td>
</tr>
<tr>
<td>ctrl</td>
<td>53,115</td>
<td>8</td>
<td>0.12</td>
<td>93,239</td>
<td>5</td>
<td>0.331</td>
<td>2.62</td>
<td>3.02</td>
</tr>
<tr>
<td>cnr</td>
<td>346,563</td>
<td>62</td>
<td>0.55</td>
<td>198,881</td>
<td>8</td>
<td>4.47</td>
<td>62.05</td>
<td>66.55</td>
</tr>
<tr>
<td>gt</td>
<td>1,141,868</td>
<td>17</td>
<td>2.4</td>
<td>694,481</td>
<td>10</td>
<td>4.64</td>
<td>23.31</td>
<td>31.4</td>
</tr>
<tr>
<td>RGG18</td>
<td>370,658</td>
<td>10</td>
<td>0.24</td>
<td>217,856</td>
<td>4</td>
<td>0.33</td>
<td>0.32</td>
<td>0.91</td>
</tr>
<tr>
<td>RGG19</td>
<td>749,103</td>
<td>11</td>
<td>0.45</td>
<td>452,745</td>
<td>4</td>
<td>0.71</td>
<td>0.76</td>
<td>1.46</td>
</tr>
<tr>
<td>RGG20</td>
<td>1,502,638</td>
<td>9</td>
<td>0.84</td>
<td>946,475</td>
<td>4</td>
<td>1.59</td>
<td>1.70</td>
<td>4.14</td>
</tr>
<tr>
<td>RGG21</td>
<td>3,018,625</td>
<td>12</td>
<td>1.7</td>
<td>1,964,436</td>
<td>5</td>
<td>3.48</td>
<td>3.77</td>
<td>9</td>
</tr>
<tr>
<td>RGG22</td>
<td>6,065,305</td>
<td>10</td>
<td>3.17</td>
<td>4,077,535</td>
<td>5</td>
<td>7.03</td>
<td>8.41</td>
<td>18.87</td>
</tr>
<tr>
<td>RMAT21</td>
<td>1,825,242</td>
<td>3</td>
<td>1.78</td>
<td>127</td>
<td>1</td>
<td>1.47</td>
<td>0.004</td>
<td>3.3</td>
</tr>
<tr>
<td>RMAT22</td>
<td>3,634,852</td>
<td>3</td>
<td>3.5</td>
<td>156</td>
<td>1</td>
<td>3.14</td>
<td>0.008</td>
<td>7.1</td>
</tr>
<tr>
<td>RMAT23</td>
<td>7,238,756</td>
<td>3</td>
<td>7.4</td>
<td>186</td>
<td>1</td>
<td>6.48</td>
<td>0.011</td>
<td>14.72</td>
</tr>
<tr>
<td>RMAT24</td>
<td>14,418,604</td>
<td>3</td>
<td>15.8</td>
<td>184</td>
<td>1</td>
<td>13.65</td>
<td>0.01</td>
<td>31.16</td>
</tr>
<tr>
<td>RMAT25</td>
<td>28,718,508</td>
<td>3</td>
<td>33.17</td>
<td>196</td>
<td>1</td>
<td>28.88</td>
<td>0.02</td>
<td>65.32</td>
</tr>
<tr>
<td>RMAT26</td>
<td>57,207,892</td>
<td>3</td>
<td>69.99</td>
<td>196</td>
<td>1</td>
<td>59.26</td>
<td>0.01</td>
<td>135.36</td>
</tr>
</tbody>
</table>

7.5. Comparison between different approaches

In Tables 2 and 3 we show the results from using a chordal graph and from using a BFS tree to initialize the algorithm, respectively. While the results presented use 8 processors, similar edge counts were observed for other numbers of processors as well. In general initializing with a BFS tree leads to faster overall computation of a maximal chordal graph, but that maximal chordal graph also has fewer total edges than when the algorithm is initialized with a chordal subgraph. The RMAT graphs are an exception. Here the BFS initialization yields the larger chordal subgraph. This is due to the fact that the ordering of the vertices in the RMAT graphs leads to an initial chordal graph that is disconnected. This is an issue with the [14] algorithm that we discussed in Section 3. Because the augmentation happens on each component separately, the subgraph is never connected. By way of contrast, the BFS tree starts with the largest connected component and can find a larger overall chordal subgraph.
Finally, in Table 4 we compare the performance of the serial chordal algorithm by Dearing et al. and that of our parallel augmented method on two processors using the chordal graph initialization. We show results on the real-world and RGG graphs since they output one large connected component. The results show that the Dearing method identifies a larger maximal chordal subgraph, although this might be an effect of the vertex ordering. For the real-world graphs the parallel code also takes longer than the sequential code because most of the time in the parallel algorithm is spent on the sequential edge processing stage. For the RGG graphs, however, the parallel times are much lower than the serial times.

8. Conclusions

In this paper we explore serial and parallel techniques for computing a maximal chordal subgraph of a graph. Existing algorithms for computing a maximal chordal subgraph are based on ordering the vertices and are therefore difficult to parallelize efficiently. We present a new algorithm that is not dependent on any particular vertex order; instead, our algorithm repeatedly augments a chordal subgraph until it is provably maximal. However, the proposed algorithm can be more expensive sequentially and fully parallelizing all the steps is non-trivial. We analyze the expected behavior of a sequential implementation of the algorithm and describe how to partially parallelize the algorithm. We then present experimental results on several instances of synthetic and real-world graphs and show that the performance of our augmentation-based algorithm is competitive with previous approaches. As a result, we believe this algorithm can potentially have a significant impact on applications that require extracting maximal chordal subgraphs from large-scale inputs. Our future work includes implementing and testing a fully parallelized version of our augmentation-based code.

Acknowledgments

This work was made possible by the College of Information Science and Technology, University of Nebraska at Omaha and Grant Number P20RR16469 from the National Center for Research Resources (NCRR), a component of the National Institutes of Health (NIH). Its contents are the sole responsibility of the authors and do not represent the official views of NCRR or NIH. This work was also funded in part by the Center for Adaptive Super Computing Software-MultiThreaded Architectures (CASS-MT) at the US Department of Energy’s Pacific Northwest National Laboratory, which is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

References


Table 3

<table>
<thead>
<tr>
<th>Input</th>
<th>Initialization Edges</th>
<th>Augmentation Edges</th>
<th>Time (parallel)</th>
<th>Chordal edges (serial)</th>
<th>Time (serial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>brca</td>
<td>48,802</td>
<td>75,116</td>
<td>0.08</td>
<td>0.17</td>
<td>0.27</td>
</tr>
<tr>
<td>ctrl</td>
<td>48,802</td>
<td>118,933</td>
<td>0.33</td>
<td>2.83</td>
<td>3.19</td>
</tr>
<tr>
<td>cnr</td>
<td>32,556</td>
<td>357,917</td>
<td>14.39</td>
<td>201.61</td>
<td>216.04</td>
</tr>
<tr>
<td>gt</td>
<td>1,123,916</td>
<td>929,337</td>
<td>11.52</td>
<td>213.11</td>
<td>225.77</td>
</tr>
<tr>
<td>RGG18</td>
<td>262,091</td>
<td>181,960</td>
<td>0.26</td>
<td>0.29</td>
<td>0.59</td>
</tr>
<tr>
<td>RGG19</td>
<td>524,224</td>
<td>365,993</td>
<td>0.57</td>
<td>0.71</td>
<td>1.32</td>
</tr>
<tr>
<td>RGG20</td>
<td>1,048,477</td>
<td>739,008</td>
<td>1.12</td>
<td>1.62</td>
<td>2.82</td>
</tr>
<tr>
<td>RGG21</td>
<td>2,097,024</td>
<td>1,483,146</td>
<td>2.35</td>
<td>3.57</td>
<td>6.06</td>
</tr>
<tr>
<td>RGG22</td>
<td>4,194,120</td>
<td>2,990,974</td>
<td>3.99</td>
<td>7.92</td>
<td>12.18</td>
</tr>
<tr>
<td>RGG23</td>
<td>8,947,978</td>
<td>4,600,722</td>
<td>7.68</td>
<td>11.02</td>
<td>16.70</td>
</tr>
</tbody>
</table>

Table 4

<table>
<thead>
<tr>
<th>Input</th>
<th>Edges from initialization</th>
<th>Edges from augmentation</th>
<th>Time (parallel)</th>
<th>Chordal edges (serial)</th>
<th>Time (serial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>brca</td>
<td>73,357</td>
<td>81,773</td>
<td>0.648</td>
<td>222,942</td>
<td>1.18</td>
</tr>
<tr>
<td>ctrl</td>
<td>53,106</td>
<td>93,418</td>
<td>3.19</td>
<td>230,053</td>
<td>2.43</td>
</tr>
<tr>
<td>cnr</td>
<td>346,575</td>
<td>198,880</td>
<td>69.14</td>
<td>1,393,178</td>
<td>45.62</td>
</tr>
<tr>
<td>RGG18</td>
<td>370,482</td>
<td>225,927</td>
<td>1.63</td>
<td>0.021</td>
<td>0.99</td>
</tr>
<tr>
<td>RGG19</td>
<td>4,128,561</td>
<td>257</td>
<td>3.46</td>
<td>0.03</td>
<td>4.26</td>
</tr>
<tr>
<td>RGG20</td>
<td>8,243,779</td>
<td>328</td>
<td>7.39</td>
<td>0.063</td>
<td>9.06</td>
</tr>
<tr>
<td>RGG21</td>
<td>16,460,670</td>
<td>363</td>
<td>15.99</td>
<td>0.08</td>
<td>19.26</td>
</tr>
<tr>
<td>RGG22</td>
<td>32,864,788</td>
<td>388</td>
<td>32.83</td>
<td>0.11</td>
<td>39.04</td>
</tr>
</tbody>
</table>

Sanjukta Bhowmick is an Assistant Professor in the College of Information Science and Technology at the University of Nebraska at Omaha. She received her Ph.D. in Computer Science from the Pennsylvania State University. Her core research area is in high performance computing with a focus on the synergy of combinatorial and numerical methods. Her current projects focus on designing parallel, efficient and robust algorithms for analyzing large-scale dynamic networks.

Tzu-Yi Chen is an associate professor of computer science at Pomona College. She received her Ph.D. in Computer Science from University of California at Berkeley. Her current research interests include network/graph analysis, combinatorial scientific computing, high performance computing, and computer science education.

Mahantesh Halappanavar is a senior research scientist in the Fundamental and Computational Science Directorate at the Pacific Northwest National Laboratory. He received his Ph.D. in Computer Science from the Old Dominion University. His research focuses on developing efficient parallel graph algorithms and their applications to several domains including the analysis of electric power grids, information analytics, sparse linear algebra, and cyber security. He explores the interplay of algorithm design, architectural features, and data characteristics targeting massively multithreaded architectures such as the Cray XMT and emerging multicore and manycore platforms.