A CO-CONNECTIVITY ALGORITHM WITH APPLICATION TO THE PARALLEL RECOGNITION OF WEAKLY TRIANGULATED GRAPHS

S.D. Nikolopoulos and L. Palios

32 – 2001

Preprint, no 32 – 01 / 2001

Department of Computer Science
University of Ioannina
45110 Ioannina, Greece

A Co-connectivity Algorithm with Application to the Parallel Recognition of Weakly Triangulated Graphs

Stavros D. Nikolopoulos and Leonidas Palios
Department of Computer Science, University of Ioannina
P.O. Box 1186, GR-45110 Ioannina, Greece
e-mail: {stavros, palios}@cs.uoi.gr

Abstract: In this paper, we consider the problem of computing the connected components of the complement of a given graph. We describe a simple sequential algorithm for this problem, which for a graph on n vertices and m edges runs in $O(n+m)$ time and is therefore optimal. The algorithm works on the graph, and not on its complement, thus avoiding a potential $\Theta(n^2)$ time complexity. Moreover, unlike previous linear co-connectivity algorithms, this algorithm admits efficient parallelization, leading to an $O(\log^2 n)$-time and $O((n+m)/\log n)$-processor parallel algorithm on the CREW PRAM model of computation. The algorithms find applications in a number of problems, such as, the recognition of weakly triangulated graphs and the detection of antiholes. Indeed, we include a parallel recognition algorithm for weakly triangulated graphs, which takes advantage of the parallel co-connectivity algorithm and achieves an $O(\log^2 n)$ time complexity using $O((n+m^2)/\log n)$ processors on the CRCW PRAM model of computation.

Keywords: Connected and co-connected components, co-connectivity algorithms, parallel algorithms, weakly triangulated graphs, parallel recognition.

1 Introduction

We consider finite undirected graphs with no loops or multiple edges. Let G be such a graph, and let u and v be vertices in G. We say that u *is connected to* v if G contains a path from u to v. The graph G is *connected* if u is connected to v for every pair u, v in $V(G)$. The *connected components* (or components) of G are the equivalence classes of vertices under the "is connected to" relation. The *co-connected components* (or co-components) of G are the connected components of the complement of G.

The problem we study in this paper is that of computing the co-connected components of a graph. The computation of the co-connected components occupies a central place in algorithmic graph theory, both in a sequential and in a parallel process environment, and is a key step in algorithms for a large number of combinatorial problems on graphs, such as finding maximum cliques and independent sets, transitive orientations, computing the modular decomposition of an undirected graph [10, 12], recognizing weakly triangulated graphs [4], detecting antiholes in graphs [25], determining k-vertex (k-edge) connectivity for $k \leq 3$ [20, 21], constructing a minimal 2-vertex (2-edge) connected spanning subgraph [13, 21].

Sequentially, the problem of determining the connected components of a graph G on n vertices and m edges is solved by a search and label approach. A simple sequential algorithm — e.g., one based on depth-first search — runs optimally in $O(n+m)$ time, whenever the input graph G is given in adjacency list representation [9, 12].

By definition, the problem of determining the co-connected components of a graph G can be easily solved by computing first the complement $\overline{G}$ of the graph G and then applying a connectivity algorithm on $\overline{G}$. It takes $\Omega(n^2)$ time to compute the complement explicitly, and thus, this approach produces a co-connectivity algorithm which may be super-linear in the size of the input graph. Ito and Yokoyama [21] proposed a storage method for representing a simple undirected graph; it maintains in a data structure the original graph if $m \leq n(n-1)/4$, and the complement graph if $m > n(n-1)/4$, where n and m are the numbers of vertices and edges of the graph respectively. Based on this storage method, they showed that a breadth-first tree and a depth-first tree on the complement of a given graph can be constructed in linear time. This result, in turn, implies a linear time algorithm for computing the co-components of a graph. Dahlhaus *et al.* [10], in their paper on modular decomposition, described a procedure for finding a depth-first forest on the complement of a directed graph in $O(n+m)$ time. The key element of their procedure is the use of a *mixed* representation of a graph; some vertices carry a list of their non-neighbors rather than that of their neighbors. This gives a depth-first forest on the complement in time proportional to the size of the mixed representation, and, thus, it implies a linear-time co-connectivity algorithm.

Developing efficient parallel algorithms for finding the components and co-components of a graph turns out to be a more challenging problem. Early parallel connectivity algorithms appear in Hirschberg [17] and Hirschberg *et al.* [18]; the proposed algorithms compute the connected components of a graph on n vertices, which is given by its adjacency matrix, for a cost of $O(n^2 \log n)$ on the CREW PRAM model of computation. Later, this cost was improved to $O(n^2)$ by Chin *et al.* [6] with preservation of the CREW PRAM model. Specifically, Chin *et al.* presented an algorithm which runs in $O(\log^2 n)$ time and requires $O(n^2/\log^2 n)$ processors. An EREW PRAM version of the algorithm was proposed by Nath and Maheshwari [24]. An $O(\log n)$-time CRCW PRAM algorithm for determining the connected components of a graph on n vertices and m edges was described by Shiloach and Vishkin [28]. Their algorithm takes the input graph G as a list of edges and computes the connected components of G using a linear number of processors; that is, it runs in $O(\log n)$ time using $O(n+m)$ processors. The algorithm was later simplified in [2]. Other parallel connectivity algorithms were proposed by Savage and JáJá [27], among which an algorithm which runs in $O(\log^2 n)$ time using $O(n \log n + m)$ processors on the CREW PRAM model. An extensive coverage of parallel connectivity algorithms can be found in [1, 22, 26].

The parallel computation of the co-connected components of a graph can be easily done by computing the complement of the graph and then by applying one of the parallel algorithms for the connected components on the complement. However, as in the sequential case, this yields non-optimal algorithms. To the best of our knowledge no parallel algorithm which "directly" computes the co-connected components exists.

In this paper, we describe a simple sequential algorithm for computing the co-components of a graph, which for a graph of n vertices and m edges runs in $O(n+m)$ time and is therefore optimal. The algorithm works on the graph, and not on its complement, and, unlike the algorithms in [10, 21], it is not data structure-based and it employs neither breadth-first-search nor depth-first-search. Additionally, it admits efficient parallelization, leading to an $O(\log^2 n)$-time and $O((n+m)/\log n)$-processor parallel algorithm on the CREW PRAM model of computation.

As an application of the parallel co-connectivity algorithm, we present a parallel algorithm for recognizing weakly triangulated graphs. An undirected graph G is called *weakly triangulated* (or *weakly chordal*) if both G and its complement $\overline{G}$ have no chordless cycle of length greater than or equal to 5 (see [14]); a *chordless cycle* of the graph G is a simple cycle such that there are no edges of G connecting any two nonconsecutive vertices of the cycle.

Weakly triangulated graphs were introduced by Hayward [14] as a natural extension of the well-known perfect class of triangulated graphs. Hayward also proved that the weakly triangulated graphs are perfect, and that not all weakly triangulated graphs are perfectly orderable [7]; indeed, the P_5-free weakly triangulated graphs are perfectly orderable, whereas the $\overline{P_5}$-free weakly triangulated graphs are not necessarily perfectly orderable [15]. Moreover, Hoáng has shown that recognizing perfectly orderable graphs remains NP-complete for weakly triangulated inputs [19].

The problem of recognizing weakly triangulated graphs has been studied, both on its own and in the context of finding chordless cycles of length $k \geq 5$. However, most of the effort has focused on sequential algorithms ([14, 29, 16, 4]), ending with the $O(m^2)$-time algorithms of Hayward, Spinrand, and Sritharan [16], and of Berry, Bordat, and Heggernes [4]. The $O(n^3m)$-time sequential algorithm of Hayward [14] for detecting chordless cycles of length at least equal to 5 implies a parallel recognition algorithm for weakly triangulated graphs running in $O(\log n)$ time with $O(n^5)$ processors on the CRCW PRAM. On the other hand, the weakly triangulated graph recognition algorithm proposed by Spinrad and Sritharan [29] does not seem to be amenable to parallelization. Recently, Chandrasekharan et al. [5] presented a parallel algorithm for obtaining a chordless cycle of length greater than or equal to $k \geq 4$ in a graph, whenever such a cycle exists, in $O(\log n)$ parallel time using $O(n^{k-4}m^2)$ processors on the CRCW PRAM. If we set $k = 5$, we see that a chordless cycle of length greater than or equal to 5 can be found in $O(\log n)$ time using $O(nm^2)$ processors. This result leads to a parallel algorithm for recognizing weakly triangulated graphs running in $O(\log n)$ time using $O(n^5)$ processors on the CRCW PRAM.

Our parallel algorithm for recognizing weakly triangulated graphs takes advantage of the parallel co-connectivity algorithm and achieves an $O(\log^2 n)$ time complexity using $O((n + m^2)/\log n)$ processors on the CRCW PRAM model of computation. Since the currently best sequential algorithm for the problem requires $O(m^2)$ time [29, 4], our algorithm is cost efficient.

The paper is organized as follows. In Section 2 we present the notation and related terminology and we prove results on which the co-connectivity algorithms rely. In Section 3 we describe the sequential co-connectivity algorithm, establish its correctness and analyze its complexity. In Section 4 we give the parallel co-connectivity algorithm and its analysis. In Section 5 we address the problem of recognizing weakly triangulated graphs; we provide background, present the parallel algorithm and analyze its time and processor complexity. Finally, in Section 6 we conclude the paper and discuss possible extensions.

2 Theoretical Framework

We consider finite undirected graphs with no loops or multiple edges. Let G be such a graph with vertex set $V(G)$ and edge set $E(G)$. The subgraph of a graph G induced by a subset S of the vertex set $V(G)$ is denoted by $G[S]$. For a vertex subset S of G, we define $G - S := G[V(G) - S]$. A graph is *complete* if all its vertices are pairwise adjacent; a complete graph on n vertices is denoted by K_n.

The *neighbourhood* $N(x)$ of a vertex $x \in V(G)$ is the set of all the vertices of G which are adjacent to x. The *closed neighbourhood* of x is defined as $N[x] := \{x\} \cup N(x)$. The neighbourhood of a subset A of vertices is defined as $N(A) := \left(\bigcup_{x \in A} N(x)\right) - A$ and its closed neighbourhood as $N[A] := A \cup N(A)$. For an edge $e = (x, y)$, the *neighbourhood* (*closed neighbourhood*) of e is the vertex set $N(\{x, y\})$ ($N[\{x, y\}]$ resp.) and is denoted by $N(e)$ ($N[e]$).

Both the sequential and the parallel co-connectivity algorithms rely on the result stated in the following lemma.

Lemma 2.1. *Let G be an undirected simple graph on n vertices and m edges, where $m < n^2/4$. If v is G's vertex of smallest degree, then the subgraph of G induced by the neighbors of v has fewer than $n/2$ vertices and fewer than $m/2$ edges.*

Proof: Since v is G's vertex of smallest degree, then degree$(v) < n/2$, for otherwise each vertex would be incident upon at least $n/2$ edges; thus, the total number m of edges would be $1/2 \sum_{u \in V(G)} \text{degree}(u) \geq 1/2(n\,(n/2))$, in contradiction to the fact that $m < n^2/4$.

We show next that the subgraph of G induced by the neighbors of v has less than $m/2$ edges. Let d be the degree of v. Then the number of edges of the subgraph induced by the d neighbors of v cannot exceed $d(d-1)/2$; if m_1 denotes the number of these edges, then we have that

$$m_1 \leq \frac{d\,(d-1)}{2}.$$

The remaining edges are edges connecting v to its neighbors and edges incident upon the remaining $n-d-1$ vertices; the latter set of edges may at worst involve edges connecting the $n-d-1$ vertices between themselves. Since each of the remaining $n-d-1$ vertices has degree at least equal to d, the number of remaining edges is

$$m_2 \geq d + \frac{1}{2}d(n-d-1) = \frac{d\,(n-d+1)}{2}.$$

Since $d < n/2$, we have that $2d < n+2 \Longleftrightarrow d-1 < n-d+1$, which implies that $m_1 < m_2$. If we additionally take into account that $m_1 + m_2 = m$, we conclude that $m_1 < m/2$. ∎

This lemma suggests the following way to compute the co-components of a graph G: if the number of edges of G is at least equal to the square of the number of vertices divided by 4, then we can apply the linear-time connected component algorithm on the complement of G; otherwise, we recursively solve the problem for the subgraph of G induced by the neighbors of the minimum-degree vertex of G, and we use this solution to construct a solution for G. Both the sequential and the parallel co-components algorithm rely on this strategy and in fact provide different ways of computing the general solution from the partial solution.

Finally, we prove two additional lemmata which will be useful in establishing the correctness of the algorithms.

Lemma 2.2. *Let G be an undirected graph which is disconnected. Then, G's complement is connected.*

Proof: Let $Q_1, \ldots, Q_k$ ($k \geq 2$) be the connected components of G. Then, any two vertices u and v of G belong to the same connected component of the complement $\overline{G}$ of G: if u and v belong to different connected components of G, then clearly they belong to the same connected component of $\overline{G}$, since in $\overline{G}$ there exists an edge connecting them; if u and v belong to the same connected component of G, then the sequence of vertices (u, x, v), where x belongs to a connected component of G other than the one to which u and v belong, forms a path in $\overline{G}$, and thus u and v belong to the same connected component of $\overline{G}$. ∎

Lemma 2.3. *Let G be an undirected simple graph and let A and B be two disjoint subsets of $V(G)$ such that the vertices in A all belong to the same connected component of the complement $\overline{G}$ of G and so do the vertices of B. If the number of edges of G with one endpoint in A and the other in B is less than $|A| \times |B|$, then the vertices in $A \cup B$ all belong to the same connected component of $\overline{G}$.*

Proof: If the number of edges of G with one endpoint in A and the other in B is less than $|A| \times |B|$, then there exists a pair of vertices $u \in A$ and $v \in B$ such that u and v are not adjacent in G. These vertices are therefore adjacent in $\overline{G}$. The lemma follows. ∎

Remark: During the process of inputting a graph, its vertices are read in in some order; we can thus assume without loss of generality that each vertex is associated with a distinct integer from 1 to n. Therefore, in the algorithm, any reference to a vertex is meant to correspond to the vertex's unique identification number. In light of that and with a slight abuse of notation, we will be using vertices to index arrays.

3 The Sequential Co-connectivity Algorithm

In order to have the necessary flexibility, we represent a graph in a *list-of-lists representation*: there is a main list of records, each corresponding to a distinct vertex of the graph; each such record has a pointer to the adjacency list of the vertex. This representation allows us to quickly construct the list-of-lists representation of any induced subgraph of a graph: if we want to construct the representation of the subgraph of a graph H induced by a set $S \subseteq V(H)$, then we only need to link in a list the vertex records of the vertices in S and to remove from the adjacency lists of these vertices any records corresponding to vertices not in S.

Algorithm Co-components
for the computation of the connected components of the complement of a graph

Input: a simple graph G on n vertices and m edges.

Output: sets of vertices; each set corresponds to a co-component of G.

1. Compute a list-of-lists representation of the graph G.
2. Allocate space for the arrays `co-comp[1..`n`]` and `size[1..`n`]`. (These arrays will receive values in the procedure Co-connectivity which is given below. When the procedure completes its computation, then, for a vertex u of G, `co-comp[`u`]` is equal to the vertex of G (possibly u as well) which is the representative of G's co-component to which u belongs, and `size[`u`]` is equal to 0, unless u is the representative of the co-component of G, in which case `size[`u`]` is equal to the size of the co-component.)
   Additionally allocate space for an auxiliary array `num[1..`n`]` and initialize its entries to 0. The array is used by the procedure Co-connectivity and helps count edges between smaller co-components to determine whether they need to be merged.
3. Call the procedure Co-connectivity on the list-of-lists representation of G.
4. Allocate an array `co-components[1..`n`]`; each entry is the head of a list of vertex records corresponding to the vertices in the same co-component. Initialize each entry to the null pointer.
   For each vertex u of G
       attach a record for the vertex u in the list of `co-components[co-comp[`u`]]`.
   Then, the co-components of G are the non-null lists attached to the entries of the array `co-components[]`.

From the above outline, it is obvious that the heart of the algorithm is the procedure Co-connectivity, which is recursive; it is described next.

Procedure Co-connectivity

Input: a list-of-lists representation of a graph H of n_H vertices and m_H edges, which is an induced subgraph of a graph G.

Output: the entries of the arrays `co-comp[]` and `size[]` corresponding to the vertices of H have been updated with respect to the co-components of H (the remaining entries, which correspond to the vertices in $V(G) - V(H)$, have not yet been updated).

1. If the number of edges of H is at least equal to $n_H^2/4$, then compute the complement $\overline{H}$ of H (using adjacency lists or an $(n_H \times n_H)$-size adjacency matrix) and compute the DFS forest [9] of $\overline{H}$; each tree in the DFS forest is a separate co-component of H. Next, update appropriately the entries of the arrays `co-comp[]` and `size[]` which correspond to the vertices of H. Return.
2. Find v, the vertex of smallest degree; let v's degree be d.

```
3. If $d = 0$
   then   {$H$ is a single vertex or a disconnected graph; $\overline{H}$ is connected}
        for each vertex $u$ of $H$ other than $v$
             co-comp[$u$] $\leftarrow v$        {$v$ is the representative of the conn. component of $\overline{H}$}
             size[$u$] $\leftarrow 0$
        co-comp[$v$] $\leftarrow v$,      size[$v$] $\leftarrow n_H$
        Return.
```

4. Compute the sets $S_1 = N(v)$ and $S_2 = V(G) - N[v]$.
From the main list of the list-of-lists representation of the graph H remove the record and adjacency list for v and partition the remaining list into two sublists: a list L_1 for the vertices in S_1 and a list L_2 for the vertices in S_2. Next, remove from the adjacency lists of the vertices in L_1 any records corresponding to vertices not in S_1, so that this list becomes a list-of-lists representation of the subgraph H' of H induced by the d neighbors of v in H.

5. Call the procedure Co-connectivity on the subgraph H'. When the recursive call returns, the entries of the arrays co-comp[] and size[] corresponding to the vertices of H' have been correctly updated with respect to the co-components of H'.

```
6. For each vertex $u$ in the list $L_2$       {$u \in V(G) - N[v]$}
        for each vertex $x$ in the adjacency list of $u$
             if $x \in N(v)$ then num[co-comp[$x$]] $\leftarrow$ num[co-comp[$x$]] + 1.
   $k \leftarrow 0$                 {$k$ counts the vertices in $N(v)$ belonging to $v$'s connected component in $\overline{H}$}
   For each vertex $w$ in the set $S_1$      {$w \in N(v)$}
        if co-comp[co-comp[$w$]] $= v$ or num[co-comp[$w$]] $\neq (n_H - d) \times$ size[co-comp[$w$]]
        then {$w$ belongs to the same connected component of $\overline{H}$ as $v$}
              co-comp[$w$] $\leftarrow v$,             {$v$ is the representative of its conn. component in $\overline{H}$}
              size[$w$] $\leftarrow 0$,
              increment $k$ by 1.
   For each vertex $u$ in the set $S_2$
        co-comp[$u$] $\leftarrow v$,        {recall that $v$ is the representative of the co-component}
        size[$u$] $\leftarrow 0$.
   co-comp[$v$] $\leftarrow v$,      size[$v$] $\leftarrow n_H - d + k$.
   For each vertex $w$ in the set $S_1$      {zero the used entries of the array num[]}
        num[$w$] $\leftarrow 0$.
```

Correctness. Clearly the correctness of the algorithm follows from the correctness of the procedure Co-connectivity. If the procedure returns without making any recursive call, then it has obviously returned at Steps 1 or 3: the correctness of Step 1 is a consequence of the correctness of the breadth-first-search algorithm [9]; the correctness of Step 3 follows from Lemma 2.2. If a recursive call to Co-connectivity is executed, then the correctness is established by means of Lemma 3.1.

Lemma 3.1. *Let H be an undirected simple graph and let v be one of its vertices. Moreover, suppose that $C_1, C_2, \ldots, C_k$ are the co-components of the subgraph $H' = H[N(v)]$ of H induced by the neighbors of v. Then,*

(i) The vertex v and the vertices in $V(H) - N[v]$ belong to the same co-component of H.

(ii) Let r_i ($1 \leq i \leq k$) be the number of edges of H connecting vertices of C_i to vertices in $V(H) - N[v]$. If $r_i < |V(C_i)| \times |V(H) - N[v]|$, then C_i belongs to the co-component of H to which v belongs; If $r_i = |V(C_i)| \times |V(H) - N[v]|$, then C_i is one of the co-components of H.

Proof: (i) Obvious, since v is non-adjacent to any of the vertices in $V(H) - N[v]$. (ii) If $r_i < |V(C_i)| \times |V(H) - N[v]|$, then there exists a vertex of C_i and a vertex in $V(H) - N[v]$ which are not adjacent; therefore, in accordance with Lemma 2.3, C_i belongs to the co-component of H to which v belongs. Suppose now that $r_i = |V(C_i|) \times |V(H) - N[v]|$; this implies that each vertex of C_i is adjacent to all the vertices in $V(H) - N[v]$. Moreover, if we take into account that C_i is one of the co-components of H', which implies that each of its vertices is adjacent to all the vertices in $N(v) - V(C_i)$, and that all the vertices of C_i are adjacent to v, we conclude that C_i is one of the co-components of H. ∎

Step 6 uses the results of Lemma 3.1 to correctly update the entries of the array `co-comp[]` corresponding to the vertices of the graph H. Note that v is selected as the representative of the co-component of H to which it belongs. The nested loop at the top of Step 6 serves to count the edges connecting a vertex in the set S_1 to a vertex in the set S_2; in particular, for every edge with one endpoint, say, x, in S_1, and the other endpoint, say, u, in S_2, we increment by 1 the entry of the array `num[]` corresponding to the representative of the co-component of H' to which x belongs. In this way, at the completion of this nested loop, the entry `num[`z`]` of a representative z of a co-component is equal to the total number of edges connecting vertices of the co-component to vertices in S_2. If this number is equal to the product of the cardinality of S_2 (i.e., $n_h - d$) to the number of vertices of the co-component (i.e., `size[`z`]`), then the co-component is a co-component of H and remains as it is; otherwise, in accordance with Lemma 3.1, the co-component needs to be merged in the co-component of H with v as the representative. This merging is done in the second loop of Step 6: until and when the representative z of the co-component is met, the second condition of the `if` statement is true, and the entries of the arrays `co-comp[]` and `size[]` of the vertices of the co-component are appropriately updated; after the representative has been met, it is the first condition of the `if` statement which is true, and the corresponding entries are again appropriately updated. For every vertex in S_1 whose `co-comp[]` entry is set equal to v, the variable k is incremented by 1. Thus, at the completion of this loop, k is equal to the number of neighbors of v which belong to the same co-component of H as v. Then, the assignment of $n_H - d + k$ to `size[`v`]` is correct taking into account that all the vertices in S_2 belong to the same co-component of H as v (recall that there is no edge in H connecting v to any of these vertices). Because of that, the assignments to the entries of the arrays `co-comp[]` and `size[]` corresponding to the vertices in S_2 are correct as well.

Time Complexity. Let us first compute the time complexity of the call to the procedure Co-connectivity for a graph H on n_H vertices and m_H edges. Step 1 takes $O(n_H + m_H)$ time: It is important to observe that $n_H^2 = \Theta(m_H)$; then, both the construction of the complement $\overline{H}$ of the graph H, and the execution of the linear-time DFS-forest algorithm on $\overline{H}$ take $O(n_H + n_H^2) = O(n_H + m_H)$ time. Steps 2 and 3 clearly take $O(n_H + m_H)$ time each. Moreover, if either Step 1 or 3 is executed, then the procedure returns, so that no more than one of these steps will be executed in all the recursive calls of the procedure Co-connectivity. Linear time is also sufficient for Step 4: the removal of the record and adjacency list of v and the list partitioning is done by means of a traversal of the main list in the list-of-lists representation of H, and by unlinking and relinking some of the visited records, which takes $O(n_H)$ time; the cleaning of the adjacency lists of the vertices in the list L_1 is also done by traversing the adjacency list records and by unlinking and relinking some of them in $O(m_H)$ total time. Additionally, the description of Step 6 directly implies that it too takes $O(n_H + m_H)$ time. Finally, let us turn to Step 5. Note that, for Step 5 to be executed, Step 1 must not have been executed, which only happens if the number m_H of edges of H is less than $n_H^2/4$; therefore, the graph H meets the conditions of Lemma 2.1. Then, Lemma 2.1 implies that the recursive call is applied on a graph on less than $n_H/2$ vertices and less than $m_H/2$ edges.

In light of Lemma 2.1 and of the previous discussion, we conclude that there exist appropriate positive constants c_1 and c_2 such that the time $T(H)$ taken by the procedure Co-connectivity applied on the graph H satisfies the following recurrence relation

$$T(H) \;=\; \begin{cases} c_1\,(n_H + m_H) & \text{if the procedure returns in Step 1 or 3;} \\ T(H') + c_2\,(n_H + m_H) & \text{otherwise;} \end{cases}$$

where H' is an induced subgraph of H such that $|V(H')| < n_H/2$ and $|E(H')| < m_H/2$. We note that if $n_H \leq 2$ then the procedure returns; if $H = K_1$ or $H = 2K_1$ then the procedure returns in Step 3, whereas if $H = K_2$, then it returns in Step 1.

It can be easily shown by induction on the number k of recursive calls that

$$T(H) \;<\; c\,(n_H + m_H) \tag{1}$$

where $c = c_1 + 2c_2$. Indeed, if $k = 0$, i.e., there is no recursive call, we have termination by one of the Steps 3, 4, or 5; this, in accordance with the definition of $T(H)$, implies that $T(H) = c_1\,(n_H + m_H)$ which is less than $c\,(n_H + m_H)$, as desired.

Suppose now that the inequality (1) holds for all graphs such that the execution of the procedure Co-connectivity on them requires $k = k_0 \geq 0$ recursive calls. We will show that it also holds for all graphs such that the execution of the procedure on them requires $k_0 + 1$ recursive calls. Let us consider such a graph H. Then, because of the first of the $k_0 + 1$ recursive calls, we have that $T(H) = T(H') + c_2\,(n_H + m_H)$, where H' is a subgraph of H on $n_{H'}$ vertices and $m_{H'}$ edges such that the execution of the procedure Co-connectivity on H' requires k_0 recursive calls, and $n_{H'} < n_H/2$ and $m_{H'} < m_H/2$. By the inductive hypothesis, $T(H') < c\,(n_{H'} + m_{H'})$, which in turn implies that $T(H') < c/2\,(n_H + m_H)$. From this, we conclude that

$$T(H) \;<\; \frac{c}{2}\,(n_H + m_H) + c_2\,(n_H + m_H) \;=\; \Big(\frac{c_1 + 2c_2}{2} + c_2\Big)(n_H + m_H) \;<\; c\,(n_H + m_H),$$

as desired. Thus, the inductive proof is complete. Therefore, the time complexity of the execution of the procedure Co-connectivity on a graph on n vertices and m edges is $O(n + m)$.

The above result implies that Step 3 of the co-components algorithm takes time linear in the size of the input graph. It is easy to see that Steps 1, 2, and 4 also take linear time. Therefore, we have the following theorem.

Theorem 3.1. *Let G be a simple undirected graph on n vertices and m edges. Then, algorithm Co-components computes the connected components of the complement of G in $O(n + m)$ time.*

4 The Parallel Co-connectivity Algorithm

In this section we present a parallel algorithm for computing the co-connected components of a graph on n vertices and m edges. Since in a parallel process environment, processing arrays is more efficient than processing lists, the representation of a graph, which we will be using, will be based on arrays. Namely, we use an array `lists[]` of size $2m$ to store the adjacency lists of the vertices of the graph; the lists are ordered in lexicographic order of the vertices to which they are associated, and each entry contains both the vertex of the adjacency list and the vertex to which the adjacency list is associated. Accessing the adjacency list of a vertex is done by means of the array `head[]` of size n; the entry corresponding to a vertex, say, u, is equal to the position in the array `lists[]` at which u's adjacency list begins. Then, the size of the adjacency list of a vertex, or alternatively its degree, can be computed by subtracting the contents of two entries of the array `head[]`. We call the representation of a graph by means of the two arrays `head[]` and `lists[]`, its *sequential-storage* representation.

In order to avoid re-indexing every time we consider a subgraph of G (recall Lemma 2.1), we represent subgraphs as follows: an array `head[]` of size n, whose entries corresponding to vertices not in the subgraph are invalid; an array `lists[]` of twice as many entries as the number of edges of the subgraph. Note that this representation has the effect that if we need to compute the degree of a vertex of the subgraph, we need to apply array packing on a copy of the array `head[]`.

Algorithm Par_Co-components
for the parallel computation of the connected components of the complement of a graph

Input: a simple graph G on n vertices and m edges.

Output: the co-connected components of the graph G.

1. if $m < n-1$ then G is disconnected, and hence $\overline{G}$ is connected; generate a list of all the vertices of G forming a single co-component; stop.
2. Compute the sequential-storage representation of the graph G and call the procedure Par_Co-connectivity on G by supplying it with the arrays `head[]` and `lists[]`; it returns the array `co-comp[]` of length n which has the property: `co-comp[`u`]` is equal to a representative of the co-connected component containing vertex u.
3. Generate a separate list of the vertices in each co-connected component of the graph G using the array `co-comp[]`.

Procedure Par_Co-connectivity

Input: a graph H of n_H vertices and m_H edges

Output: the entries of the arrays `co-comp[]` such that `co-comp[`u`]` is equal to a representative of the co-connected component of H containing vertex u

1. If the number m_H of edges of H is at least equal to $n_H^2/4$, then compute the co-connected components of H by applying a CREW parallel connectivity algorithm on the graph $\overline{H}$, and transfer the results in the corresponding entries of the array `co-comp[]`; Return;
2. Find the vertex of smallest degree in H; let it be v.
3. If the degree of v is equal to 0, then H is either a single vertex graph or is disconnected. In either case, $\overline{H}$ is connected; set to v the entries of the array `co-comp[]` corresponding to the vertices of H; Return;
4. Compute the set $S_1 = N(v)$; compute the number $k = n_H - d(v)$ of the remaining vertices, where $d(v)$ is the degree of v in H;
5. Compute the graph H' induced by $N(v)$ on H and its number $n_{H'}$ of vertices and $m_{H'}$ of edges;
6. $S_2 = \emptyset$;
   For each vertex u in S_1, do in parallel
       if $d(u) < d'(u) + k$ {*$d(u)$ and $d'(u)$ are the degrees of u in H and H' resp.*}
       then include u in the set S_2;
7. If $m_{H'} < n_{H'} - 1$, then the graph H' is disconnected, and set all the entries of the array `co-comp[]` which correspond to the vertices of H' equal to the same vertex; otherwise, call the procedure Par_Co-connectivity on the graph H'; (When the recursive call returns, the entries of the array `co-comp[]` corresponding to the vertices of H' have been updated so that `co-comp[`u`]` $=$ `co-comp[`v`]` iff u and v are in the same co-connected component of H'.)

```
8. For each vertex $u$ not in the set $S_1$, do in parallel        $\{u \in V(H) - N(v)\}$
       set co-comp[$u$] $\leftarrow v$;
   For each vertex $u$ in the set $S_2$, do in parallel
       set co-comp[co-comp[$u$]] $\leftarrow v$;
   For each vertex $u$ in the set $S_1$, do in parallel        {note that co-comp[$v$] $= v$}
       if co-comp[co-comp[$u$]] $\neq$ co-comp[$u$], then set co-comp[$u$] $\leftarrow v$;
```

Correctness. The correctness of the algorithm follows from the correctness of the procedure Par_Co-connectivity. If the procedure returns without making any recursive call, then it has obviously returned at Steps 1, 3 or 8 if $m_{H'} < n_{H'} - 1$: the correctness of Step 1 is a consequence of the correctness of the parallel connectivity algorithm used; the correctness of Step 3 and of the updating at Step 7 follows from Lemma 2.2. Let us now consider the case of the execution of the procedure Co-connectivity when it makes a recursive call on the subgraph H', and suppose that the recursive call updates correctly the entries of the array `co-comp[]` corresponding to the vertices of H'. We show that at the completion of Step 8, the entries of the array `co-comp[]` corresponding to the vertices of H have been correctly updated. First recall that the vertex v and the vertices in $V(H) - N[v]$ belong to the same co-component of H; the first `for` loop of Step 8 just takes care of that using v as the representative of the co-component. Next, it is important to note that if for a vertex $u \in N(v)$, it holds that $d(u) < d'(u) + |V(H) - N(v)|$, where $d(u)$ and $d'(u)$ are the degrees of u in H and H' respectively, then there exists a vertex in $V(H) - N[v]$ which is not adjacent to u; this implies that u and all the vertices in u's co-component of H' belong to the same co-component of H as v. Let us now see how the second and third `for` loops of Step 8 ensure the above behavior. Let u be a vertex in $S_2 \subseteq N(v)$, i.e., $d(u) < d'(u) + |V(H) - N(v)|$. Then, in the second `for` loop, the entry of the array `co-comp[]` corresponding to the representative of the co-component of H' to which u belongs is set equal to v. In the third loop, the `co-comp[]` entries corresponding to the vertices of u's co-component in H', except for the representative, are set equal to v. Thus, the entire co-component of H' has become a part of the co-component of v in H, as desired. The above holds for any such u in S_2, so that the entries of the array `co-comp[]` corresponding to the vertices of all these co-components are correctly updated. For the remaining co-components of H', the contents of the `co-comp[]` entries corresponding to their vertices do not change: in the second `for` loop, no change occurs since none of the vertices of these co-components belongs to the set S_2; no change occurs in the third `for` loop either, since the content of the `co-comp[]` entry corresponding to the representative of each of these co-components is equal to the representative, and the condition in the `if` statement is false.

Time and Processor Complexity. Next, we analyze the time and processor complexity of the algorithm.

Step 1: The verification of the condition in the `if` statement takes constant time, while generating the single co-component, whenever the condition is true, takes $O(1)$ time using $O(n)$ processors on the CREW PRAM model.

It is important to note that if the algorithm does not stop at Step 1, then $n - 1 \leq m < n^2$, which implies that $\log m = \Theta(\log n)$.

Step 2: Let us see how to construct the sequential-storage representation of G from the standard adjacency lists representation. We compute the ranks of the elements in each of the adjacency lists. The largest rank in each adjacency list is its size; we collect those in an auxiliary array of size n and we compute parallel prefix sums on it. Then, we set `head[1]` $\leftarrow 1$, and we set `head[`i`]` equal to 1 plus the $(i-1)$-st prefix sum. The array `lists[]` is updated as follows: if the j-th record of the adjacency list of the vertex u corresponds to the vertex v, then we set the entry `lists[head[`u`]` $+j-1$`]` equal to the pair (u, v). Using list ranking takes $O(\log n)$ time using $O(m)$ processors on the EREW PRAM model. Computing parallel prefix sums on an array of size n takes $O(\log n)$ time and $O(n/\log n)$ processors on the EREW PRAM model. Finally, updating the arrays `head[]` and `lists[]` takes $O(1)$ time using $O(n+m)$ processors on the CREW PRAM. If $T(G)$ and $P(G)$ denote the time and processor complexity, respectively, for the computation of the array `co-comp[]` using the procedure Par_Co-connectivity on the graph G, then Step 2 is executed in $O(\log + T(G))$ time using $O(n+m+P(G))$ processors or in $O(\log^2 + T(G))$ time using $O((n+m)/\log n + P(G))$ processors on the CREW PRAM model.

Step 3: We can compute the number k of the co-connected components of G by counting the number of distinct elements of the array `co-comp[]`. This computation can be easily done by sorting the array `co-comp[]` and determining all its elements such that `co-comp[`$i-1$`]` $\neq$ `co-comp[`i`]`, $2 \leq i \leq n$. It is known that n elements can be sorted in $O(\log^2 n)$ time and $O(n/\log n)$ processors on the CREW PRAM model using a standard merge-sort parallel algorithm (in fact, the pipelined merge-sort requires $O(n \log n/p + \log n)$ time, where p is the number of processors; this algorithm is commonly referred to as *Cole's merge-sort algorithm* in the literature [8]). Thus, it is easy to see that the computation of the number k of co-components of G requires $O(\log^2 n)$ time and $O(n/\log n)$ processors on the CREW PRAM model. From the sorted array `co-comp[]`, we can construct k arrays, each containing the vertices of each co-connected component, in $O(1)$ time with n processors or in $O(\log^2 n)$ time with $n/\log^2 n$ processors on the EREW PRAM model. Then, we can generate a separate list of the vertices in each co-connected component of the graph G within the same time and processor bounds.

Taking into consideration the time and processor complexity of each step of the algorithm, we obtain the following result.

Lemma 4.1. *When applied on a graph G on n vertices and m edges, Algorithm Par_Co-components runs in $O(\log^2 n + T(G))$ time using a total of $O((n+m)/\log n + P(G))$ processors, where $T(G)$ and $P(G)$ are the time and processor complexity, respectively, for the computation of the array* `co-comp[]` *of length n.*

Let us now estimate the parameters $T(\,)$ and $P(\,)$ by analyzing the time and processor complexity of the procedure Par_Co-connectivity. We consider that the procedure is applied on a graph H on n_H vertices and m_H edges; the graph H is an induced subgraph of a graph G which has n vertices and m edges. It is important to note that thanks to the tests at Step 1 of the algorithm Par_Co-components and at Step 7 of the procedure Par_Co-connectivity, it holds that $n_H - 1 \leq m_H \leq n_H^2$, which implies that $n_H = O(m_H)$ and $\log m_H = \Theta(\log n_H)$.

Step 1: If the number m_H of edges of the graph H is at least equal to $n_H^2/4$, then we can compute the co-connected components of H by applying the $O(\log^2 n_H)$-time and $O(n_H^2/\log^2 n_H)$-processor CREW parallel connectivity algorithm of Chin *et al.* [6] on the graph $\overline{H}$ (an EREW version of the algorithm has appeared in [24]). To be able to use the above algorithm, we need to compute an adjacency array representation of $\overline{H}$, or similarly an adjacency array representation of H; to do that we need to apply array packing on `head[]`, to obtain an array storing the transformation to the new indices, as well as another array storing the reverse transformation. Array packing and the computation of these arrays can be done in $O(\log n)$ time using $O(n/\log n)$ processors on the CREW PRAM. The algorithm of Chin *et al.* takes $O(\log^2 n_H) = O(\log^2 n)$ time and $O(n_H^2/\log^2 n_H)$ processors on the CREW PRAM; since $m_H \geq n_H^2/4$, we have that $n_H^2 = O(m_H)$, and thus the total number of processors required for this computation is $O(m_H/\log^2 n_H) = O(m_H/\log n_H)$. After the algorithm of Chin *et al.* has returned, we use the array which stores the reverse transformation to update the array `co-comp[]`; this takes $O(1)$ time using $O(n)$ processors on the CREW PRAM model.

Step 2: The minimum degree is computed by using array packing on a copy of `head[]`, by computing and storing the degrees of the vertices, and then by finding the minimum of them. Therefore, the minimum degree of the vertices of H can be computed in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM model. Then, within the same time and processor bounds, we can compute a vertex exhibiting the minimum degree.

Step 3: Step 3 can be executed in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM model.

Step 4: The set S_1 can be computed by marking the neighbors of v from its adjacency list into an array of size n, whose entries initially are equal to 0; this can be done in $O(1)$ time using $O(n)$ processors, or in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM model.

Step 5: First, we mark as useless all the entries of the array `head[]` corresponding to vertices in $V(H) - N[v]$. Next, we mark as useless all the entries of the array `lists[]` which contain pairs, say, (u, v), where $u \in V(H) - N[v]$ or $v \in V(H) - N[v]$; then, we remove the useless entries by applying array packing. Finally, the lengths of the adjacency lists in the updated `lists[]` array are computed, and the contents of the array `head[]` are updated as well. All the above operations can be done in $O(\log n)$ time using $O((n/\log n) + (m_H/\log n_H))$ processors on the CREW PRAM; recall that $\log m_H = \Theta(\log n_H)$.

Step 6: The inclusion of a vertex in the set S_2 can be done by marking the corresponding entry of an auxiliary array of size n. If we take into account that computing degrees involves using array packing on auxiliary arrays of size n, we have that this step can be executed in $O(\log n)$ time using $O(n/\log n)$ processors on the CREW PRAM model.

Step 7: If the condition in the `if` statement is true, the updating of the `co-comp[]` can be done in $O(1)$ time using $O(n)$ processors or in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM model; otherwise, the time for the completion of the recursive call needs to be accounted.

Step 8: From its description, it is easy to see that this step can be completed in $O(1)$ time using $O(n)$ processors, or in $O(\log n)$ time using $O(n/\log n)$ processors on the CREW PRAM model. (Recall that the sets S_1 and S_2 are represented by arrays of size n whose entries corresponding to the elements of the sets are equal to 1 and the remaining entries are equal to 0; thus, the membership of a vertex in these sets is tested by checking whether an entry of an array is equal to 1 or 0.)

Summarizing, we have that the execution of the procedure Par_Co-connectivity on a graph H with n_H vertices and m_H edges takes

$$T(H) \leq \begin{cases} c_1 \log^2 n & \text{if the procedure employs a recursive call;} \\ T(H') + c_2 \log n & \text{otherwise;} \end{cases}$$

where H' is an induced subgraph of H such that $|V(H')| < n_H/2$ and $|E(H')| < m_H/2$. Then, it is not difficult to show that $T(G) = O(\log^2 n)$.

The number of processors needed for the execution of Par_Co-connectivity on the graph H is $O((n/\log n) + (m_H/\log n_H))$. We next show that this is $O((n+m)/\log n)$. We distinguish two cases:

1. $n_H \geq \sqrt[4]{n}$: Then, $\log n_H = \Theta(\log n)$. If we also take into account that $m_H \leq m$, we have that $O(m_H/\log n_H) = O(m/\log n)$.
2. $n_H < \sqrt[4]{n}$: Then, $m_H < n_H^2 < \sqrt[2]{n}$. Thus, $m_H/\log n_H < m_H < \sqrt[2]{n} < n/\log n$, which implies that $O(m_H/\log n_H) = O(n/\log n)$.

Therefore, $P(G) = O((n+m)/\log n)$. The previous discussion and Lemma 4.1 lead to the following theorem.

Theorem 4.1. *Algorithm Par_Co-components computes the co-connected components of a graph on n vertices and m edges in $O(\log^2 n)$ time using $O((n+m)/\log n)$ processors on the CREW PRAM model.*

5 Recognizing Weakly Triangulated Graphs in Parallel

In this section we present a parallel algorithm for recognizing weakly triangulated graph. The algorithm takes advantage of the parallel co-connectivity algorithm. Before presenting the algorithm, we give a brief review of the notions on which the algorithm relies.

5.1 Theoretical Background

Let G be an undirected graph with no loops or multiple edges. A vertex set $S \subset V(G)$ is called a *separator* if the graph $G - S$ has at least two connected components, an *ab-separator* $(a, b \in V(G))$ if a and b belong to different connected components of $G - S$, a *minimal ab-separator* if S is an ab-separator and no proper subset of S is an ab-separator, and a *minimal separator* if S is a minimal ab-separator for a pair $\{a, b\}$ of vertices of G [3, 4].

In general, generating minimal separators can be done by computing the neighborhoods of the connected components resulting from the removal of certain vertex sets [3]. In [23], the minimal separators in the neighborhood of a vertex x are computed in the following way: for each connected component Q_i of the graph $G - N[x]$, compute the set $N(Q_i)$; this set is a minimal separator induced in $N(x)$.

Definition 1. Let e be an edge of a graph G, and let Q_i be a connected component of the graph $G - N[e]$. Then, the vertex set $S_i(e) := N(Q_i)$ is called an *e-separator* of the graph G.

It is interesting to note that $S_i(e) \subseteq N(e)$. Moreover, it is not difficult to see that:

Lemma 5.1. *Let e be an edge of a graph G on n vertices and m edges. Then,*

(i) There are fewer than n e-separator of the edge e.

(ii) The total sum of the sizes of the e-separators of the edge e is less than m.

(iii) Each e-separator of the edge e is a minimal separator of the graph G.

Let $Q_1, Q_2, \ldots, Q_k$ be the connected components of the graph $G - N[e]$, where e is an edge of G, and let $S_1(e), S_2(e), \ldots, S_k(e)$ be the e-separators of G which correspond to the connected components. Then, with respect to an e-separator $S_i(e)$, $1 \leq i \leq k$, we define the following three sets of vertices:

$$A_i(x; e) := S_i(e) \cap (N(x) - N(y))$$
$$A_i(y; e) := S_i(e) \cap (N(y) - N(x))$$
$$A_i(xy; e) := S_i(e) \cap N(x) \cap N(y)$$

The set $A_i(x; e)$ consists of the vertices in $S_i(e)$ which are adjacent to x but not to y. Similarly, the set $A_i(y; e)$ consists of the vertices in $S_i(e)$ which are adjacent to y but not to x. The set $A_i(xy; e)$ consists of the vertices in $S_i(e)$ which are adjacent to both x and y. Clearly, $S_i(e) = A_i(x; e) + A_i(y; e) + A_i(xy; e)$, $1 \leq i \leq k$. We also define the set $S_0(e)$ as the set $N(e) - \bigcup_i S_i(e)$; note that $S_0(e) = N(e)$ if and only if the graph $G - N[e]$ is empty.

By extending the notion of a simplicial vertex [11, 23], which helps characterize triangulated graphs, Berry et al. [4] introduced the notion of an *LB-simplicial edge*, and gave a new characterization of weakly triangulated graphs.

Definition 2 ([4]). An edge $e = (x, y)$ of a graph G is *LB-simplicial* if one of the following holds:

(i) For each minimal separator S induced in $N(e)$, the edge e is S-saturating;

(ii) $N[e] = V(G)$.

The definition is based on the concept of *S-saturation* introduced by Hayward in [14]: Given a set S of vertices, an edge e of the graph $G[V(G) - S]$ is *S-saturating* if, for each connected component R_i of the complement of $G[S]$, at least one end-point of e sees all the vertices of R_i. Then, we can give an alternate equivalent definition of an LB-simplicial edge.

Definition 3. Let $e = (x, y)$ be an edge of a graph G and let $S_1(e), S_2(e), \ldots, S_k(e)$ be the e-separators of G which correspond to the edge e. Then, the edge e is *LB-simplicial* if either $S_0(e) = N(e)$ or none of the co-connected components of the graph $G[S_i(e)]$ contains vertices from both $A_i(x; e)$ and $A_i(y; e)$, $1 \leq i \leq k$.

Based on the notion of an LB-simplicial edge, Berry et al. [4] proved the following theorem.

Theorem 5.1 ([4]). *A graph G is weakly triangulated if and only if every edge of G is LB-simplicial.*

Moreover, they derived an $O(m^2)$-time algorithm for recognizing weakly triangulated graphs [4]; their algorithm is a direct application of Theorem 5.1 and thus it works by checking all the edges of the given graph for being LB-simplicial.

5.2 A parallel weakly triangulated graph recognition algorithm

In this section we present a parallel algorithm recognizing weakly triangulated graphs. The algorithm is based on the result provided by Theorem 5.1. We assume that the input graph is connected; for disconnected graphs, we apply the algorithm on each of their connected components.

```
Algorithm WT_REC for the recognition of weakly triangulated graphs
Input:    a connected graph $G$ on $n$ vertices and $m$ edges.

  1. For each edge $e$ of the graph $G$, do in parallel
       1.1 compute the set $N(e)$;
       1.2 compute the connected components $Q_1, Q_2, \ldots, Q_k$ of the graph $G - N[e]$;
       1.3 compute the corresponding $e$-separators $S_1, S_2, \ldots, S_k$ of $G$;
  2. Collect all the $e$-separators of the graph $G$ in a list $S$; if the list is empty, then $G$ is a weakly
     triangulated graph; Stop; Otherwise, remove from the list $S$ all the duplicate entries;
  3. If the number of (distinct) $e$-separators in the list $S$ is greater than $n + m$, then $G$ is not a
     weakly triangulated graph; Stop;
  4. For each $e$-separator $S_i$ in the list $S$ $(1 \leq i \leq n + m)$, do in parallel
       4.1 compute the vertex sets $A_i(x; e)$ and $A_i(y; e)$, where $e = (x, y)$ is the edge asso-
           ciated with the $e$-separator $S_i$;
       4.2 compute the induced subgraph $G[S_i]$ of $G$;
       4.3 compute the co-connected components $U_{i1}, U_{i2}, \ldots, U_{i\ell}$ of the graph $G[S_i]$;
       4.4 for $j = 1, \ldots, \ell$ do in parallel
               if $U_{ij}$ contains vertices from both $A_i(x; e)$ and $A_i(y; e)$, then set $M[i] \leftarrow 1$;
               otherwise set $M[i] \leftarrow 0$;
  5. If there exists an entry of the array $M[\,]$ which is equal to 1, then the graph $G$ is not weakly
     triangulated; otherwise, $G$ is a weakly triangulated graph.
```

Correctness The correctness of the parallel algorithm WT_REC is established through the Theorem 5.1.

Time and Processor Complexity. Below, we analyze the complexity of the algorithm step by step. Recall that the graph G is connected so that $n = O(m)$ and $\log m = \Theta(\log n)$.

Step 1: This step is executed for each of the m edges of G.

Substep 1.1: We use an array $N_e[]$ of size n. For each vertex adjacent to x, we mark the corresponding entry of $N_e[]$ with x. Next, for each vertex adjacent to y, we check the corresponding entry of $N_e[]$; if it is marked with x, then we mark it with xy instead, otherwise, we mark it with y. In this way, we have recorded in $N_e[]$ the entire neighbourhood of the edge e. The above computation can be done in $O(1)$ time using $O(n)$ processors or in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM model.

Substep 1.2: We use Shiloach and Vishkin's algorithm [28] for computing the connected components of $G - N[e]$. The algorithm receives the input graph as a collection of edges; so, we compute an array of all the edges of G in a fashion similar to that used in Step 2 of Algorithm Par_Co-components to construct the array `lists[]`. This takes $O(\log n)$ time using $O((n+m)/\log n)$ processors on the CREW PRAM model. Next, we remove from the array of edges the entries corresponding to edges incident upon at least a vertex in $N[e]$ and we use array packing to pack the array; this takes $O(\log m) = O(\log n)$ time using $O(m/\log m) = O(m/\log n)$ processors on the EREW PRAM. Since Shiloach and Vishkin's algorithm takes $O(\log n)$ time and needs $O(n+m)$ processors on the CRCW PRAM model, this substep requires $O(\log n)$ time and $O(m)$ processors on the CRCW PRAM for each edge of the graph.

Substep 1.3: Let $Q_1, Q_2, \ldots, Q_k$ be the connected components of the graph $G - N[e]$ computed in the previous substep. We check each edge $e' = (u, v)$ of G and if $u \notin N[e]$, $v \in N(e)$, and u belongs to the component Q_i, we add an entry (i, v) in an array of size $2m$; if $v \notin N[e]$, $u \in N(e)$, and v belongs to the component Q_j, we add an entry (j, u) in the array. After having processed all the edges, we sort the array lexicographically, remove duplicates, and pack the array. In this way, we have the vertices of each e-separator of e collected together and in increasing index order. We can use this array to place pointers for the vertices of each separator (the pointer points to the entry of the array storing the first vertex of the separator) and compute the sizes of the separators; these computations are identical to the construction of the array `head[]` and the computation of degrees at Step 2 of Algorithm Par_Co-components, and take $O(\log n)$ time and $O((n+m)/\log n)$ processors on the CREW PRAM. Since sorting takes $O(\log m) = O(\log n)$ time and $O(m)$ processors on the EREW PRAM model, the entire substep can be completed in $O(\log n)$ time using $O(n+m)$ processors on the CREW PRAM model.

Thus, the whole step is executed in $O(\log n)$ time using a total of $O(m)$ processors on the CRCW PRAM model.

Step 2: In the previous step, for each edge of the graph we have computed a collection of pointers to its e-separators. Then, by using list ranking or parallel prefix sums, we can rank each e-separator in the list or array of e-separators of each edge. If we use parallel prefix sums on an array which stores the number of e-separators per edge and use the ranking we mentioned earlier, we can produce an array of all the e-separators without concurrent writes. Now, we need to sort the array and remove the duplicates. Two e-separators of lengths, say, n_i and n_j, are compared based on their vertices which have been stored in increasing index order: we need to check the first $\min\{n_i, n_j\}$ vertices; if they don't match, we readily obtain an ordering of the two e-separators, whereas if they match, then the e-separator with the fewest vertices is considered smaller. Such a comparison takes $O(\log n_i)$ time using $O(n_i/\log n_i)$ processors or $O(\log n)$ time using $O(n_i/\log n)$ processors on the CREW PRAM model. Since sorting an array of size h can be done in $O(\log h)$ time using $O(h)$ processors on the EREW PRAM, sorting the array of e-separators takes $O(\log^2 n)$ time using $O(m^2/\log n)$ processors on the CREW PRAM; recall that $\sum_i n_i = O(m^2)$ in accordance with Lemma 5.1. Finally, we remove the duplicates; two e-separators are identical if they contain the same number of vertices and these vertices are identical. The removal is done by comparing pairs of consecutive e-separators in the sorted array, in order to determine whether they are identical; if they are, the one corresponding to a higher index of the array is considered useless. Then, array packing brings the different e-separators in consecutive positions in the array. Comparing consecutive entries takes $O(\log n)$ time using $O(m^2/\log n)$ processors on the CREW PRAM; array packing on an array of size $O(nm)$ takes $O(\log n)$ time using $O(nm/\log nm) = O(m^2/\log n)$ processors on the EREW

PRAM. In total, Step 2 is executed in $O(\log^2 n)$ time using $O(m^2/\log n)$ processors on the CREW PRAM model.

Step 3: From the array packing in the previous step, we know the number of distinct e-separators. Thus, this step takes $O(1)$ time using one processor on the EREW PRAM.

Step 4: This step is executed for each of the e-separators in the list S, which do not exceed $n+m$.

Substep 4.1: The vertex sets $A_i(x;e)$ and $A_i(y;e)$, represented in arrays of size n, can be computed in $O(\log n)$ time using $O(n)$ processors on the EREW PRAM model by taking advantage of the array $N_e[\,]$ which stores all the information on the neighbourhood of e; see Step 1.1.

Substep 4.2: The graph $G[S_i]$ can be constructed from a copy of the adjacency list representation of G, where records of vertices not in S_i are marked useless and are removed by means of array packing or pointer jumping. The graph can be constructed in $O(\log n)$ time using $O(m)$ processors on the CREW PRAM model. Note that arrays need to be built to hold the transformations from the old indexes to the new indexes of the graph $G[S_i]$ and back. This too can be executed in the above stated time and processor complexity.

Substep 4.3: Here, we compute the co-connected components of the graph $G[S_i]$; let n_i and m_i be the numbers of vertices and edges of $G[S_i]$. This computation can be done in $O(\log^2 n_i)$ time with $O((n_i+m_i)/\log n_i)$ processors or in $O(\log^2 n)$ time with $O((n_i+m_i)/\log n)$ processors on the CREW PRAM model using Algorithm Par_Co-components of the previous section. Since we have at most $n+m$ e-separators, each having less than n vertices and less than m edges, we have that for all the e-separators this substep takes $O(\log^2 n)$ time with $O(m^2/\log n)$ processors on the CREW PRAM model.

Substep 4.4: Let $V_1, V_2, \ldots, V_t$ be the co-components of the graph $G[S_i]$. We use an array $L_i[\,]$ of size n, which we fill by processing the vertices of G as follows. Consider a vertex u: if $u \in A_i(x;e)$ and u belongs to the co-component V_j, we set $L_i[u] \leftarrow (j,x)$; if $u \in A_i(y;e)$ and u belongs to the co-component V_j, we set $L_i[u] \leftarrow (j,y)$; if $u \notin A_i(x;e) \cup A_i(y;e)$, we set $L_i[u] \leftarrow (0,0)$. Next, we sort the array $L_i[\,]$ lexicographically, and check whether there exist two consecutive entries containing (ℓ,x) and (ℓ,y). If yes, then we set $M[i] \leftarrow 1$ else we set $M[i] \leftarrow 0$.

The above description implies that Step 4 can be executed in $O(\log^2 n)$ time with $O(m^2/\log n)$ processors on the CREW PRAM model.

Step 5: This step can be executed in $O(\log n)$ time and $O(n+m)$ processors on the EREW PRAM model by computing the maximum of the array $M[\,]$ and testing whether it is equal to 1.

Taking into consideration the time and processor complexity of each step of the algorithm, we obtain that the parallel algorithm WT_REC on a connected graph on n vertices and m edges takes $O(\log^2 n)$ time and $O(m^2/\log n)$ processors to be executed on the CRCW PRAM model. Thus, we have the following result.

Theorem 5.2. *It can be determined whether a connected graph on n vertices and m edges is a weakly triangulated graph in $O(\log^2 n)$ time using a total of $O(m^2/\log n)$ processors on a CRCW PRAM model.*

If the input graph is not connected then we apply Shiloach and Vishkin's algorithm to compute its connected components. Taking into account that this algorithm takes $O(\log n)$ time using $O(n+m)$ processors or $O(\log^2 n)$ time using $O((n+m)/\log n)$ processors, the following result can be established.

Corollary 5.1. *It can be determined whether a graph on n vertices and m edges is a weakly triangulated graph in $O(\log^2 n)$ time using a total of $O((n+m^2)/\log n)$ processors on a CRCW PRAM model.*

6 Concluding Remarks

In this paper we describe a sequential co-connectivity algorithm which, for a graph on n vertices and m edges, runs in $O(n+m)$ time and is therefore optimal. The algorithm is simple, works on the graph, and not on its complement, avoiding a potential $\Theta(n^2)$ time complexity, and admits efficient parallelization, leading to an $O(\log^2 n)$-time and $O((n+m)/\log n)$-processor CREW PRAM parallel algorithm.

The co-connectivity algorithms find applications in a number of problems, such as, the recognition of weakly triangulated graphs and the detection of antiholes. In fact, we describe a parallel recognition algorithm for weakly triangulated graphs, which takes advantage of the parallel co-connectivity algorithm and achieves an $O(\log^2 n)$ time complexity using $O((n+m^2)/\log n)$ processors on the CRCW PRAM model of computation.

Due to the work of Shiloach and Vishkin [28], the components of a graph can be efficiently computed in $O(\log n)$ parallel time, for a cost of $O((n+m)\log n)$ on the CRCW PRAM model. Thus, since our co-connectivity CREW PRAM algorithm is of the same cost, it is reasonable to ask whether the time complexity of our algorithm can be improved to $O(\log n)$, with preservation of the cost $O((n+m)\log n)$ and perhaps the CREW PRAM model. Moreover, due to the optimal sequential co-connectivity algorithm of this paper and [10, 21], another interesting question is whether we can design optimal $O(\log n)$-time or $O(\log^2 n)$-time parallel algorithms for computing the co-components of a graph. We pose both questions as open problems.

Our parallel algorithm for recognizing weakly triangulated graphs runs in $O(\log^2 n)$ time on the CRCW PRAM model, for a cost of $O((n+m^2)\log n)$ and, thus, it is cost-efficient due to the work of Hayward, Spinrand, and Sritharan [16] and Berry, Bordat, and Heggernes [4]. It is interesting to investigate whether there exist $O(\log n)$-time or $O(\log^2 n)$-time cost-optimal recognition algorithms for weakly triangulated graphs.

7 References

[1] S.G. Akl, *Parallel Computation: Models and Methods*, Prentice Hall, 1997.

[2] B. Awerbuch and Y. Shiloach, New connectivity and MSF algorithms for ultra-computer and PRAM, *IEEE Trans. Computers* **36**, 1258–1263, 1987.

[3] A. Berry, J.-P. Bordat, and O. Cogis, Generating all the minimal separators of a graph, *Proc. 25th Inter. Workshop on Graph-Theoretic Concepts in Computer Science (WG'99)*, 167–172, 1999.

[4] A. Berry, J.-P. Bordat, and P. Heggernes, Recognizing weakly triangulated graphs by edge separability, *Nordic J. Computing* **7**, 164–177, 2000.

[5] N. Chandrasekharan, V.S. Lakshmanan, and M. Medidi, Efficient parallel algorithms for finding chordless cycles in graphs, *Parallel Process. Letters* **3**, 165–170, 1993.

[6] F.Y. Chin, J. Lam, and I. Chen, Efficient parallel algorithms for some graph problems, *Communications of the ACM* **25**(9), 659–665, 1982.

[7] V. Chvátal, Perfectly ordered graphs, *Annals of Discrete Math.* **21**, 63–65, 1984.

[8] R. Cole, Parallel merge sort, *SIAM J. Computing* **17**(4), 770–785, 1988.

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms* (2nd edition), MIT Press, Inc., 2001.

[10] E. Dahlhaus, J. Gustedt and R.M. McConnell, Efficient and practical modular decomposition, *Proc. 8th ACM-SIAM Symp. on Discrete Algorithms (SODA'97)*, 26–35, 1997.

[11] G.A. Dirac, On rigid circuit graphs, *Anh. Math. Sem. Univ. Hamburg* **25**, 71–76, 1961.

[12] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs* Academic Press, New York, 1980.

[13] X. Han, P. Kelson, V. Ramachandran and R.E. Tarjan, Computing minimal spanning subgraphs in linear time, *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms (SODA'92)*, 146–156, 1992.

[14] R.B. Hayward, Weakly triangulated graphs, *J. Comb. Theory B* **39**, 200–208, 1985.

[15] R.B. Hayward, Meyniel weakly triangulated graphs - I: co-perfect orderability, *Discrete Applied Math.* **73**, 199–210, 1997.

[16] R.B. Hayward, J. Spinrad, and R. Sritharan, Weakly chordal graph algorithms via handles, *Proc. 11th ACM-SIAM Symp. on Discrete Algorithms (SODA 2000)*, 2000.

[17] D.S. Hirschberg, Parallel algorithms for the transitive closure and the connected components problems *Proc. 8th ACM Symp. on Theory of Computing (STOC'76)*, 55–57, 1976.

[18] D.S. Hirschberg, A.K. Chandra and D.V. Sarwate, Computing connected components on parallel computers, *Communications of the ACM* **22**, 461–464, 1979.

[19] C.T. Hoàng, On the complexity of recognizing a class of perfectly orderable graphs, *Discrete Applied Math.* **66**, 219–226, 1996.

[20] J.E. Hopcroft and R.E. Tarjan, Dividing a graph into triconnected components, *Information and Computation* **79**, 43–59, 1988.

[21] H. Ito and M. Yokoyama, Linear time algorithms for graph search and connectivity determination on complement graphs, *Inform. Process. Letters* **66**, 209–213, 1998.

[22] J. Jájá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[23] C.G. Lekkerkerker and J.C. Boland, Representations of a finite graph by a set of intervals on the real line, *Fund. Math.* **51**, 45–64, 1962.

[24] D. Nath and S.N. Maheshwari, Parallel algorithms for the connected components and minimal spanning trees, *Inform. Process. Letters* **14**(1), 7–11, 1982.

[25] S.D. Nikolopoulos and L. Palios, *Algorithms for detecting holes and antiholes in graphs*, manuscript.

[26] J. Reif (ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[27] C. Savage and J. JáJá, Fast, efficient parallel algorithms for some graph problems, *SIAM J. Computing* **10**, 682–691, 1981.

[28] Y. Shiloach and U. Vishkin, An $O(\log n)$ parallel connectivity algorithm, *J. Algorithms* **3**, 57–67, 1982.

[29] J.P. Spinrad and R. Sritharan, Algorithms for weakly triangulated graphs, *Discrete Applied Math.* **59**, 181–191, 1995.