Theory Comput. Systems **37**, 527–546 (2004)
DOI: 10.1007/s00224-003-1074-x

Theory of Computing Systems

An Optimal Parallel Co-Connectivity Algorithm

Ka Wong Chong,[1] Stavros D. Nikolopoulos,[2] and Leonidas Palios[2]

[1] Department of Computer Science, The University of Hong Kong,
Porfulam Road, Hong Kong
kwchong@cs.hku.hk

[2] Department of Computer Science, University of Ioannina,
GR-45110 Ioannina, Greece
{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 works on the input graph and not on its complement, and which for a graph on n vertices and m edges runs in optimal $O(n+m)$ time. Moreover, unlike previous linear co-connectivity algorithms, this algorithm admits efficient parallelization, leading to an optimal $O(\log n)$-time and $O((n+m)/\log n)$-processor algorithm on the EREW PRAM model of computation. It is worth noting that, for the related problem of computing the connected components of a graph, no optimal deterministic parallel algorithm is currently available. The co-connectivity algorithms find applications in a number of problems. In fact, we also 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 EREW PRAM model of computation.

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 of vertices u, v of 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 number of combinatorial problems on graphs, such as finding maximum cliques, independent sets, and transitive orientations [12], [24], computing the modular decomposition of an undirected graph [10], [12], recognizing weakly triangulated graphs [4], and detecting antiholes in graphs [23].

Sequentially, the problem of determining the connected components of a graph is solved by a search and label approach. For a graph on n vertices and m edges given in adjacency list representation, a simple sequential algorithm—e.g., one based on depth-first search—runs optimally in $O(n + m)$ time [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 [19] showed that a depth-first-search tree and a breadth-first-search 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-search 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. As their algorithm computes a depth-first-search forest on the complement in time proportional to the size of the mixed representation, it implies a linear-time co-connectivity algorithm. It must be noted that the depth-first-search tree algorithms in both [19] and [10] rely on linear-time solutions to special cases of the disjoint set union problem.

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 [16] and [17]; the proposed algorithms compute the connected components of a graph on n vertices, which is given by its adjacency matrix, in $O(\log^2 n)$ time using $O(n^2/\log n)$ processors on the CREW PRAM model of computation. Later Chin et al. [6] presented an algorithm which runs in $O(\log^2 n)$ time and requires $O(n^2/\log^2 n)$ processors on the CREW PRAM, thus improving the cost to $O(n^2)$. An EREW PRAM version of the algorithm exhibiting the same time and processor complexity was proposed by Nath and Maheshwari [22]. An $O(\log n)$-time $O(n + m)$-processor CRCW PRAM algorithm for determining the connected components of a graph on n vertices and m edges was described by Shiloach and Vishkin [26]; the algorithm was later simplified by Awerbuch and Shiloach [2]. Other parallel connectivity algorithms were proposed by Savage and JáJá [25], among which was an algorithm which runs in $O(\log^2 n)$ time using $O(n \log n + m)$ processors on the CREW PRAM model. Recently, Chong et al. [8] described a parallel algorithm for computing the minimum spanning tree/forest of a graph which runs in $O(\log n)$ time using $O(n+m)$ processors on the EREW PRAM; the algorithm can be used to compute the connected components of a graph within the same time and processor complexity. Additionally, Chong et al. [7] presented an algorithm for fast integer sorting which enabled them to achieve the EREW PRAM computation of

minimum spanning trees in $O(\log n)$ time using $O((n+m)/\sqrt{\log n})$ processors, and in $O(\log n)$ time using $O(n^2/\log n)$ processors; note that the latter algorithm is optimal for dense graphs. An extensive coverage of parallel connectivity algorithms can be found in [1], [20].

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], [19], it is not data structure-based and it employs neither breadth-first search nor depth-first search. Additionally, it admits efficient parallelization, leading to the first optimal $O(\log n)$-time and $O((n+m)/\log n)$-processor parallel algorithm on the EREW 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 [13]); a *chordless cycle* of the graph G is a simple cycle such that there are no edges of G connecting any two non-consecutive vertices of the cycle. The class of weakly triangulated graphs was introduced by Hayward [13] as a natural extension of the well-known class of triangulated graphs. The weakly triangulated graphs have been shown to be perfect, although not all weakly triangulated graphs are perfectly orderable [13]; 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 [14]. Moreover, Hoáng has shown that recognizing perfectly orderable graphs remains NP-complete for weakly triangulated inputs [18].

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 [13], [27], [15], [4], ending with the $O(m^2)$-time algorithms of Hayward et al. [15], and of Berry et al. [4]. The $O(n^3m)$-time sequential algorithm of Hayward [13] 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 [27] does not seem to be amenable to parallelization. Recently, Chandrasekharan et al. [5] presented a parallel algorithm for obtaining a chordless cycle of length at least 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. The application of this algorithm for $k = 5$ both on the graph and on its complement gives a parallel algorithm for recognizing weakly triangulated graphs running in $O(\log n)$ time using $O(n^5)$ processors on the CRCW PRAM model.

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 EREW PRAM model of computation. Since

the currently best sequential algorithm for the problem requires $O(m^2)$ time [27], [4], our algorithm is EREW 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; its vertex set and edge set are denoted by $V(G)$ and $E(G)$, respectively. The subgraph of a graph G induced by a subset S of $V(G)$ is denoted by $G[S]$. For a vertex subset S of G, we define $G - S := G[V(G) - S]$.

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

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 graph on n vertices and m edges. If v is G's vertex of minimum degree, then the subgraph of G induced by the neighbors of v has fewer than $\sqrt{2m}$ vertices.*

Proof. Since v is G's vertex of minimum degree, then $\sum_x \text{degree}(x) \geq n\,\text{degree}(v)$, which implies that $\text{degree}(v) \leq (\sum_x \text{degree}(x))/n = 2m/n$. Additionally, since $m \leq n(n-1)/2 < n^2/2$, we have that $n > \sqrt{2m}$. The combination of these two inequalities yields that $\text{degree}(v) < 2m/\sqrt{2m} = \sqrt{2m}$, as desired. □

This lemma implies that time linear in the size of a graph G suffices to compute explicitly the complement of the subgraph $G[N(v)]$ induced by the neighbors of the minimum-degree vertex v of G, as well as its connected components. Thus, we can compute the co-components of a graph G as follows: we 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 algorithms rely on this strategy and in fact provide different ways of computing the general solution from the partial solution.

Finally, we include a well-known fact and prove an additional lemma 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.*

Lemma 2.3. *Let G be an undirected 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 $\overline{G}$ 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 $|A| \cdot |B|$, 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 $|A| \cdot |B|$, 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 inputing a graph, its vertices are read 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 use vertices to index arrays.

3. The Sequential Co-Connectivity Algorithm

Although an optimal parallel algorithm readily implies an optimal sequential algorithm, we chose to devote this section to the description of the sequential algorithm for the co-connectivity problem, thus introducing the way we take advantage of Lemma 2.1 and at the same time giving an alternative implementation of the computation.

We assume that the input graph G has n vertices and m edges and is given in adjacency-list representation. The algorithm uses three arrays of size n, namely, `co-comp[ ]`, `size[ ]`, and `num[ ]`. 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. The array `num[ ]` helps count the edges between smaller co-components to determine whether they need to be merged (see Lemma 2.3).

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

Input: an undirected graph G on n vertices and m edges.
Output: arrays `co-comp[ ]` and `size[ ]` as described above.

1. Find v, a vertex of G of minimum degree; let v's degree be d;
2. If $d = 0$
   then {*G is a single vertex or a disconnected graph*; *$\overline{G}$ is connected*}
   for each vertex w of G other than v do
   `co-comp[`w`]` $\leftarrow v$; {*use v as the representative*}

```
        size[w] ← 0;
    co-comp[v] ← v;      size[v] ← n;
    Stop.
```

3. Allocate space for the arrays `co-comp[ ]`, `size[ ]`, and `num[ ]`; initialize the entries of the arrays `size[ ]` and `num[ ]` to 0;
4. Construct the complement $\overline{G}[N(v)]$ of the subgraph $G[N(v)]$ induced by the neighbors of v in G, and compute its connected components;
   for each vertex u adjacent to v in G do
   `co-comp[u]` ← the representative of the connected component of $\overline{G}[N(v)]$ to which u belongs;
   increment `size[co-comp[u]]` by 1;
5. For each vertex w in $V(G) - N[v]$ do
   {*add w to the same connected component of $\overline{G}$ as v*}
   `co-comp[w]` ← v; {v: *representative of the component of* $\overline{G}$}
   `size[w]` ← 0;
   for each vertex x adjacent to w in G do
   if $x \in N(v)$
   then `num[co-comp[x]]` ← `num[co-comp[x]]` + 1;
   $k \leftarrow 0$;
   {*k counts the vertices in $N(v)$ belonging to v's connected component in $\overline{G}$*}
   for each vertex u adjacent to v in G do
   if `co-comp[co-comp[u]]` $= v$ or
   `num[co-comp[u]]` $\neq (n - d - 1) \cdot$ `size[co-comp[u]]`
   then {*u belongs to the same connected component of $\overline{G}$ as v*}
   `co-comp[u]` ← v;
   `size[u]` ← 0;
   increment k by 1;
   `co-comp[v]` ← v;
   `size[v]` ← $n - d + k$;

The nested loop at the top of Step 5 serves a twofold purpose. Firstly, we include the vertices in $V(G) - N[v]$ to the co-component of G to which v belongs by appropriately updating the corresponding entries of the array `co-comp[ ]` (and `size[ ]`); this should be so, since, in $\overline{G}$, v is adjacent to all the vertices in $V(G) - N[v]$. Note that v is selected as the representative of the co-component of G to which it belongs. Secondly, we count the edges connecting a vertex in $N(v)$ to a vertex in $V(G) - N[v]$; in particular, for every edge with one endpoint, say, x, in $N(v)$, and the other endpoint in $V(G) - N[v]$, we increment by 1 the entry of the array `num[ ]` corresponding to the representative of the co-component of $G[N(v)]$ 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 of $G[N(v)]$ is equal to the total number of edges connecting vertices of the co-component to vertices in $V(G) - N[v]$. If this number is equal to the product of the cardinality of $V(G) - N[v]$ (i.e., $n - d - 1$) and the number of vertices of the co-component (i.e., `size[z]`), then the co-component is a co-component of G and remains as it is; otherwise, in accordance with Lemma 3.1, the co-component needs to be merged into the co-component of G to which v belongs.

This merging is done in the second loop of Step 5: 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 $N(v)$ 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 G as v. Then the assignment of $n-d+k$ to `size[v]` is correct taking into account that all the vertices in $V(G)-N[v]$ belong to the same co-component of G as v.

The algorithm does not compute the co-components of the input graph G as collections of vertices; nevertheless, this can be easily obtained from the array `co-comp[ ]` as follows: we allocate an array `co-components[ ]` of size n, whose entries are heads of lists of vertex records, initialized to the null pointer; next, for each vertex u of G, we 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[ ]`.

Correctness. The correctness of Step 2 follows from Lemma 2.2, while the correctness of Step 4 results from the correctness of the connected component algorithm used. Then the correctness of the algorithm ensues from the correctness of Step 5 which is established by the preceding detailed description and by means of Lemma 3.1.

Lemma 3.1. *Let H be an undirected 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 $H[N(v)]$ of H induced by the neighbors of v. Then the following statements hold*:

(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 $r_i < |V(C_i)| \cdot |V(H) - N[v]|$, then C_i belongs to the co-component of H to which v belongs; if $r_i = |V(C_i)| \cdot |V(H) - N[v]|$, 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 $r_i < |V(C_i)|\cdot|V(H)-N[v]|$, 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 $r_i = |V(C_i)|\cdot|V(H)-N[v]|$; 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 the subgraph $H[N(v)]$, which implies that each of its vertices is adjacent in G 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. □

Time and Space Complexity. Clearly, Step 1 takes $O(n + m)$ time, while Steps 2 and 3 take $O(n)$ time. In Step 4 the construction of the complement $\overline{G}[N(v)]$ of the

subgraph of G induced by the neighborhood $N(v)$ of vertex v relies on a re-indexing array which allows us to map $N(v)$ to the set $\{1, 2, \ldots, d\}$, thus enabling the construction of an adjacency-matrix representation of $\overline{G}[N(v)]$ in $O(d^2) = O(m)$ time and space (see Lemma 2.1). Moreover, the computation of the connected components of $\overline{G}[N(v)]$ takes an additional $O(n + m)$ time, while the for-loop in Step 4 takes $O(n)$ time. Step 5 also takes $O(n + m)$ time; note that the tests whether a vertex x belongs to the neighborhood $N(v)$ of v or to the set $V(G) - N[v]$ can be carried out in constant time by means of an array of size n, in which we have marked the neighbors of vertex v.

Summarizing, we have the following theorem.

Theorem 3.1. *Let G be an 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 and space.*

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. As in the description of the sequential co-connectivity algorithm, we assume that the input graph is given in adjacency-list representation. We also assume that, for each edge uv, the two records in the adjacency lists of u and v are linked together; this helps us re-index the vertices in any subgraph of the given graph fast.

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

Input: an undirected graph G on n vertices and m edges.
Output: the co-connected components of the graph G.

1. Compute the degrees of the vertices of G and store them in an array $\mathtt{d}_G[\,]$ of size n; locate a vertex, say, v, of G of minimum degree;
2. If $m < n - 1$ or $\mathtt{d}_G[v] = 0$
   then {*G is a single vertex or a disconnected graph*; $\overline{G}$ *is connected*}
   for each vertex u of G, do in parallel
   $\mathtt{co\text{-}comp}[u] \leftarrow v$; {*use v as the representative*}
   Stop.
3. Compute the graph $\overline{G}[N(v)]$ which is the complement of the subgraph of G induced by the neighbors of v in G;
   compute the degrees of the vertices of $\overline{G}[N(v)]$ and store them in $\mathtt{d}_{\overline{G}[N(v)]}[\,]$;
   compute the connected components of $\overline{G}[N(v)]$;
4. For each vertex u adjacent to v in G, do in parallel
   $\mathtt{co\text{-}comp}[u] \leftarrow$ the representative of the connected component of $\overline{G}[N(v)]$ to which u belongs;
   if $\mathtt{d}_G[u] + \mathtt{d}_{\overline{G}[N(v)]}[u] < n - 1$
   then {*there exists a vertex x in $V(G) - N[v]$ such that $ux \notin E(G)$*}
   mark the representative of u;

5. For each vertex u of G, do in parallel
   if $u = v$ or $uv \notin E(G)$
   then `co-comp[u]` $\leftarrow v$;
   else $\{u \in N(v)\}$
      if the representative of u is marked then `co-comp[u]` $\leftarrow v$;

Correctness. The correctness of Step 2 follows from Lemma 2.2. The objective of Step 4 is to locate among the neighbors of v those which are not adjacent to at least one vertex in $V(G) - N[v]$ and to mark the representatives of the co-components of $G[N(v)]$ to which these vertices belong; Lemma 4.1, given below, establishes the correctness of the condition used in Step 4 to locate these vertices.

Lemma 4.1. *Let G be an undirected graph on n vertices and let v be a vertex of G. Then a vertex $u \in N(v)$ is non-adjacent to at least one vertex in $V(G) - N[v]$ if and only if $\mathtt{d}_G[u] + \mathtt{d}_{\overline{G}[N(v)]}[u] < n-1$, where $\mathtt{d}_G[u]$ and $\mathtt{d}_{\overline{G}[N(v)]}[u]$ denote the degree of u in G and in $\overline{G}[N(v)]$, respectively.*

Proof. Let k and ℓ be the numbers of neighbors of u which belong to $N(v)$ and $V(G) - N[v]$, respectively. Then, clearly, $\mathtt{d}_{\overline{G}[N(v)]}[u] = |N(v)| - k - 1$, and $\mathtt{d}_G[u] = k + \ell + 1$. Then the condition $\mathtt{d}_G[u] + \mathtt{d}_{\overline{G}[N(v)]}[u] < n-1$ is equivalent to $|N(v)| + \ell < n-1$ and thus to $\ell < n - 1 - |N(v)|$. The lemma follows, since the quantity in the right-hand side of the last inequality is precisely the number of vertices in $V(G) - N[v]$. □

Next, we show that at the completion of Step 5, all the entries of the array `co-comp[ ]` have been correctly updated; note that at the end of Step 4, only the entries corresponding to the neighbors of v have been updated and in such a way as to reflect the co-components of $G[N(v)]$. First, clearly, the vertex v and the vertices in $V(G) - N[v]$ belong to the same co-component of G; Step 5 correctly sets the entries of the array `co-comp[ ]` for these vertices using v as the representative of the co-component. Additionally, if S is the set of neighbors of v which are not adjacent to at least one vertex in $V(G) - N[v]$, then the co-component of G to which v belongs also contains all the vertices in S (note that in $\overline{G}$ any such vertex is adjacent to a vertex in $V(G) - N[v]$) as well as all the vertices belonging to the same co-component of $G[N(v)]$ as any vertex in S. Step 4 locates all the vertices in S and marks the representatives of their co-components in $G[N(v)]$ while Step 5 sets to v the entries of the array `co-comp[ ]` corresponding to the vertices of the marked co-components of $G[N(v)]$. For any remaining co-component of $G[N(v)]$, the contents of the `co-comp[ ]` entries corresponding to its vertices do not change in Step 5 and it correctly becomes a co-component of G; note that each vertex of any such co-component of $G[N(v)]$ is adjacent to v, to all the vertices in $V(G) - N[v]$, and to all the vertices in all other co-components of $G[N(v)]$.

Time and Processor Complexity. Next, we analyze the time and processor complexity of the algorithm. For details on the PRAM techniques mentioned below, see [1], [20]. We note that augmenting the adjacency-list representation of the input graph so that, for each edge uv, it contains pointers linking the record of u in the adjacency list of v and

the record of v in the list of u, can be easily established in optimal $O(1)$ time using $O(m)$ processors on the EREW PRAM model using an auxiliary array.

Step 1. The computation of the degree of a vertex u of the graph G can be done by applying list ranking on the adjacency list of u and by taking the maximum rank; this can be done in $O(\log n)$ time using $O(\text{degree}(u)/\log n)$ processors on the EREW PRAM. The computation for all the vertices takes $O(\log n)$ time and $O(m/\log n)$ processors on the same model of computation. Locating the vertex v of minimum degree in G can be executed in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM.

Step 2. 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, or $O(\log n)$ time using $O(n/\log n)$ processors, on the EREW PRAM model.

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

Step 3. An adjacency-list representation of the subgraph $G[N(v)]$ can be obtained by appropriate processing of copies of the adjacency lists of G. Then re-indexing (based on the ranks of the vertices in the adjacency list of v) is applied to map the vertices in $N(v)$ to the integers $\{1, 2, \ldots, \mathtt{d}_G[v]\}$. To do that, each vertex in $N(v)$ broadcasts its new index number to its adjacency list; next, for each edge, the two adjacency-list records associated with it, exchange the new index information. Then the adjacency-list representation of $G[N(v)]$ can be readily converted into the new indexing scheme. An adjacency-list representation of the graph $\overline{G}[N(v)]$ can be obtained by first constructing an adjacency matrix for $G[N(v)]$, and then by building the appropriate adjacency lists. The above computations can all be completed in $O(\log n)$ time using $O((n+m)/\log n)$ processors on the EREW PRAM, thanks to the fact that $|N(v)| = O(\sqrt{m})$ (Lemma 2.1).

Computation of the degrees of the vertices in $\overline{G}[N(v)]$ is done in a fashion similar to that described in Step 1, and it thus can be done in $O(\log m)$ time using $O(m/\log m)$ processors on the EREW PRAM; $\overline{G}[N(v)]$ has $O(\sqrt{m})$ vertices and $O(m)$ edges. Computation of the connected components of $\overline{G}[N(v)]$ is done by applying the minimum spanning tree/forest parallel algorithm of Chong et al. for dense graphs [7]. For a graph on N vertices, their algorithm takes $O(\log N)$ time and uses $O(N^2/\log N)$ processors on the EREW PRAM; since the graph $\overline{G}[N(v)]$ has $O(\sqrt{m})$ vertices (Lemma 2.1), execution of the algorithm on $\overline{G}[N(v)]$ takes $O(\log m)$ time and $O(m/\log m)$ processors on the EREW PRAM. The minimum spanning tree algorithm works by constructing supervertices to represent the current minimum spanning subtrees, from which a representative-based representation of the connected components is obtained. It is important to note that the component information needs to be re-indexed back to the original indexing scheme. This can be easily done, while avoiding concurrent reads, by using one copy of the re-indexing array for each vertex in $N(v)$; since $|N(v)| = O(\sqrt{m})$, the copying can be done in $O(\log m)$ time using $O(m/\log m)$ processors on the EREW PRAM, and the re-indexing in $O(\log n)$ time using $O(\sqrt{m}/\log n)$ processors on the same model of computation.

Step 4. The updating of the entries of the array `co-comp[]` can be executed in $O(1)$ time using $O(n)$ processors, or in $O(\log n)$ time using $O(n/\log n)$ processors, on the

EREW PRAM. The marking of the representatives of the co-components results in concurrent writing if executed as described in the algorithm, since there may be several vertices with the same value in their co-comp[ ] entries. In order to ensure exclusive writing, we use an auxiliary array $P[\,]$ of size n, which we update as follows: for each vertex u for which $\mathrm{d}_G[u] + \mathrm{d}_{\overline{G[N(v)]}}[u] < n - 1$, we set $P[u] \leftarrow$ co-comp[u], while the remaining entries are considered invalid. Next, we pack the valid entries of the array $P[\,]$, we sort them, and we mark duplicate entries: the packing takes $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM; the sorting takes $O(\log m)$ time using $O(\sqrt{m})$ processors on the same model of computation, since $G[N(v)]$ has fewer than $\sqrt{2m}$ vertices (Lemma 2.1), and thus fewer than $\sqrt{2m}$ co-components; marking the duplicate entries takes $O(\log m)$ time using $O(\sqrt{m}/\log m)$ processors on the EREW PRAM. Then, by means of the valid non-duplicate entries of the array $P[\,]$, we can mark the representatives of the co-components of $G[N(v)]$ that need to be marked in $O(1)$ time using $O(\sqrt{m})$ processors in an EREW fashion. Thus, Step 4 can be executed in $O(\log n)$ time using $O(n/\log n + \sqrt{m}) = O((n+m)/\log n)$ processors on the EREW PRAM model.

Step 5. Testing whether a particular vertex of G is not adjacent to v can be done in $O(1)$ time using one processor on the EREW PRAM by means of an array of size n storing the neighbors of v in G. On the other hand, testing for a vertex u whether the representative co-comp[u] is marked results in concurrent reading, if executed as described. To avoid it, we use another auxiliary array $R[\,]$ of size n, which stores pairs of vertices of G; for each vertex $u \in N(v)$, we set $R[u] \leftarrow$ (co-comp[u], u), whereas the entries corresponding to all vertices in $V(G) - N(v)$ are invalid. Next, we pack the valid entries and sort them. Then the pairs with the same first element appear in consecutive positions in $R[\,]$; we identify the leftmost entry of each such run of pairs and we assign to a processor associated with this entry the task of verifying whether the representative is marked or not; if (r, u) is such an entry, it suffices to check whether r is marked or not. If the representative is marked, then the value v is sent to all the entries in the same run (by using interval broadcasting on $R[\,]$; see [1]); otherwise, the value sent is r. Then, for each valid pair (r, u) in $R[\,]$, the entry co-comp[u] is set equal to the value sent to it. Since $|N(v)| = O(\sqrt{m})$, it is not difficult to see that all the above operations can be completed in $O(\log n)$ time using $O(n/\log n + \sqrt{m}) = O((n+m)/\log n)$ processors on the EREW PRAM model.

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

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

5. Recognizing Weakly Triangulated Graphs in Parallel

In this section we present a parallel algorithm for recognizing weakly triangulated graphs, which takes advantage of the parallel co-connectivity algorithm described in the previous

section. 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 of a graph G can be done by computing the neighborhoods (in G) of the connected components of subgraphs resulting after the removal of certain vertex sets [3]. In [21], the minimal separators in the neighborhood of a vertex x are computed in the following way: for each connected component Q_i of the subgraph $G - N[x]$, compute the set $N(Q_i)$ in G; this set is a minimal separator of G. This approach can be extended to edges of G [4]. In particular, we define the notion of an edge-separator as follows:

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 $N(Q_i)$ is called an *edge-separator*[1] of G (contributed by e) and is denoted by $S_i(e)$.

Figure 1 shows an edge e contributing two edge-separators $S_1(e) = \{a, g, q\}$ and $S_2(e) = \{a, f, g, p\}$. For an edge e of a graph G, let $S_1(e), S_2(e), \ldots, S_k(e)$ be the edge-separators of G corresponding to the connected components of the graph $G - N[e]$. 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) *the edge e contributes fewer than n edge-separators*;
(ii) *the total sum of the sizes of the edge-separators contributed by the edge e is less than m*;
(iii) *each edge-separator contributed by the edge e is a minimal separator of the graph G.*

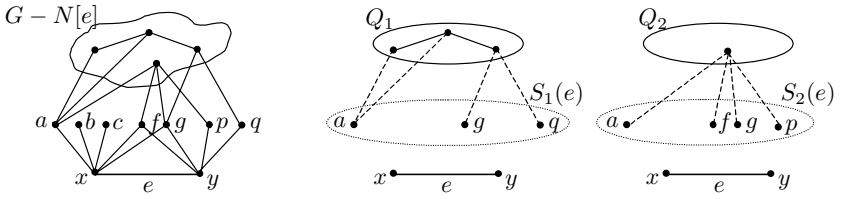

Fig. 1. The edge $e \in E(G)$ contributes the edge-separators $S_1(e)$ and $S_2(e)$.

[1] The notion of the edge-separator has been used in [4] as well, but it has been referred to as "a minimal separator (included in the neighborhood of e)"; as this expression is more general, for the sake of clarity, we chose to define and use the term "edge-separator."

Proof. (i) Clearly true, since the graph $G - N[e]$ has fewer than n vertices and hence fewer than n connected components. (ii) It follows from the fact that for a connected component Q_i of $G - N[e]$, the size of the edge-separator $S_i(e)$ does not exceed the number of edges of G connecting vertices in Q_i to vertices in $N(e)$; note that any such edge contributes to no connected component of $G - N[e]$ other than Q_i. (iii) Consider a connected component Q_i of $G - N[e]$, and let u be a vertex of Q_i. Then the edge-separator $S_i(e)$ is a minimal ux-separator, where x is an endpoint of e. □

By extending the notion of a simplicial vertex [11], [21], 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 of a graph G is *LB-simplicial* if one of the following holds:

(i) $N[e] = V(G)$;
(ii) for each edge-separator $S_i(e)$, the edge e is $S_i(e)$-saturating.

The definition is based on the concept of *S-saturation* introduced by Hayward in [13]: Given a set S of vertices, an edge e of the graph $G - S$ is *S-saturating* if, for each connected component Q of the complement of $G[S]$, at least one endpoint of e is adjacent to all the vertices of Q. If $e = xy$, we define the following three sets of vertices:

$$A(e; x) = N(x) - N[y],$$

$$A(e; y) = N(y) - N[x],$$

$$A(e) = N(x) \cap N(y),$$

which clearly partition the neighborhood $N(e)$ of the edge e (for example, in Figure 1, $A(e; x) = \{a, b, c\}$, $A(e; y) = \{p, q\}$, and $A(e) = \{f, g\}$). Then we can give an alternate definition of an LB-simplicial edge.

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

It is not difficult to see that Definition 3 is equivalent to Definition 2. The edge e is $S_i(e)$-saturating for an edge-separator $S_i(e)$ if and only if for each connected component Q of the complement of $G[S_i(e)]$ at least one endpoint of e is adjacent to all the vertices of Q; the latter is equivalent to $Q \subseteq A(e; x) \cup A(e)$ or $Q \subseteq A(e; y) \cup A(e)$, that is, Q does not contain vertices from both $A(e; x)$ and $A(e; y)$.

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], which is a direct application of Theorem 5.1 and thus it works by checking whether all the edges of the given graph are LB-simplicial. The algorithm also takes advantage of the following result.

Observation 5.1 [4]. *Let G be a weakly triangulated graph on n vertices and m edges. Then the number of distinct edge-separators of G does not exceed $n + m$.*

5.2. *The Parallel Recognition Algorithm*

Our parallel algorithm for recognizing weakly triangulated graphs 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.
Output: yes, if G is a weakly triangulated graph; otherwise, no.

1. For each edge $e = xy$ of the graph G, do in parallel
   1.1. compute the sets $A(e; x)$, $A(e; y)$, and $N[e]$;
   1.2. compute the connected components $Q_1(e), Q_2(e), \ldots, Q_k(e)$ of $G - N[e]$;
   1.3. compute the corresponding edge-separators $S_1(e), \ldots, S_k(e)$ of G;
2. Collect all the edge-separators of the graph G in a list $\mathcal{S}$; if the list is empty, then G is a weakly triangulated graph; Stop;
3. Select all the distinct entries $\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_\ell$ of the list $\mathcal{S}$; if their number ℓ is greater than $n + m$, then G is not a weakly triangulated graph; Stop;
4. For each edge-separator $\widehat{S}_i$ $(1 \leq i \leq \ell)$, do in parallel
   4.1. compute the induced subgraph $G[\widehat{S}_i]$ of G;
   4.2. compute the co-connected components of $G[\widehat{S}_i]$;
5. For each edge-separator S_i in the list $\mathcal{S}$, do in parallel
   let $\widehat{S}_j$ be the edge-separator among $\widehat{S}_1, \ldots, \widehat{S}_\ell$ which is identical to S_i;
   if there exist vertices $u, v \in S_i$ such that
   u, v belong to the same co-component of $G[\widehat{S}_j]$ and
   $u \in A(e; x)$ and $v \in A(e; y)$, where the edge $e = xy$ contributed S_i
   then the graph G is not weakly triangulated; Stop;
6. The graph G is a weakly triangulated graph.

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

Time and Processor Complexity. Below, we compute the complexity of the algorithm using a step-by-step analysis; all complexities mentioned are analyzed under the EREW PRAM model of computation. Recall that the input graph G is connected so that $n = O(m)$ and $\log m = \Theta(\log n)$; we also assume that it is given in adjacency-list representation, where, as in the case of the parallel co-connectivity algorithm, for each

edge uv, the record of u in the adjacency list of v and the record of v in the list of u are linked together.

Step 1. This step is executed for each of the m edges of the input graph G. In order to achieve an EREW execution of it, we copy m times the adjacency-list representation of G. This computation can be done in $O(\log n)$ time with $O(m^2/\log n)$ processors.

Substep 1.1. We first compute the vertex set $N[e]$, where $e = xy$. For this computation we use an array $N_e[\,]$ of size n. Then, 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. Finally, we mark the entries of $N_e[\,]$ which correspond to x and y with X and Y, respectively. In this way, we have recorded in $N_e[\,]$ the entire closed neighborhood of the edge e. The above computations can be done in $O(1)$ time using $O(n)$ processors.

The vertex sets $A(e; x)$ and $A(e; y)$ are needed in the execution of Step 5. Storing each of them in an array of size n results in concurrent reading during Step 5; to avoid that, we represent each vertex w as a pair $(w, t(e, w))$, where $t(e, w)$ is equal to 1 or 2, if w belongs to $A(e; x)$ or to $A(e; y)$, respectively, and is equal to 0 otherwise. Computing these pairs during the processing of an edge e in Substep 1.1 can be done in $O(1)$ time using $O(n)$ processors. Note that the second field $t(e, w)$ is only used in Step 5; in Steps 2–4 it is ignored in the computations, but it is copied or moved whenever the associated vertex is copied or moved, which simply results in a constant factor overhead in the computation.

In total, for each edge e, Substep 1.1 takes $O(1)$ time using $O(n)$ processors on the EREW PRAM model.

Substep 1.2. We use Chong et al.'s algorithm [8] for computing the connected components of the graph $G - N[e]$. The algorithm receives the input graph as a collection of edges given in lexicographic order. An array storing the edges of $G - N[e]$ can be easily obtained from an array storing all the edges of G; the latter array can be constructed as follows: compute the ranks of the elements in each of the adjacency lists of G; find the largest rank in each adjacency list, which is its size; collect those in an auxiliary array of size n and compute parallel prefix sums on it; then, if the ith adjacency list is the adjacency list of vertex u, and the jth record of this adjacency list corresponds to the vertex v, set the $(ps[i-1]+j)$th entry of the edge array equal to the pair (u, v), where $ps[i-1]$ denotes the $(i-1)$st prefix sum. Thus, the construction of the array of edges of G takes $O(\log n)$ time using $O((n+m)/\log n)$ processors. After this array has been constructed, we remove from it 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. Since Chong et al.'s connectivity algorithm takes $O(\log n)$ time and needs $O(n+m)$ processors on the EREW PRAM model, this substep computes the connected components of the graph $G - N[e]$ in $O(\log n)$ time using $O(m)$ processors on the same model of computation.

Substep 1.3. Let $Q_1(e), Q_2(e), \ldots, Q_k(e)$ be the connected components of the graph $G - N[e]$ computed in the previous substep. In order to collect the vertices of

each edge-separator contributed by e, we form one pair (i, v) for each edge uv where u belongs to the component Q_i and $v \in N(e)$. To do that, we work as follows on a copy of the adjacency-list representation of the graph G: using list ranking and prefix sums, the adjacency-list information can be copied on an array $P_e[\,]$ of size $2m$; using interval broadcasting on $P_e[\,]$, each vertex broadcasts to its neighbors an integer which is equal to i if the vertex belongs to the component Q_i, or 0 otherwise, and this information is exchanged between each pair of records corresponding to the same edge (recall that, for each edge ab, the two records in the adjacency lists of a and b are linked together); then each neighbor, say, v, of each vertex u (which has received an integer i_u from u and an integer i_v from v) writes in the corresponding entry of $P_e[\,]$ the pair (i_u, v) if $i_u \neq i_v$, and $(0, v)$ otherwise. Observe that this implies that $P_e[\,]$ contains exactly one pair (i, v) for each edge uv such that u belongs to the component Q_i and $v \in N(e)$, whereas the remaining entries are of the form $(0, w)$. Since copying, list ranking, and interval broadcasting can be executed optimally on an EREW PRAM model, the array $P_e[\,]$ is updated in $O(\log n)$ time and $O((n+m)/\log n)$ processors on this model of computation.

Next, the array $P_e[\,]$ is sorted lexicographically, duplicate entries and entries whose first field is equal to 0 are removed, and the array is packed. In this way we have the vertices of each edge-separator of the edge e collected together and in increasing index order. We can use this array to create 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 take $O(\log n)$ time and $O((n + m)/\log n)$ processors. 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 same model.

Thus, as the above substeps are executed for each edge of G, the whole step is executed in $O(\log n)$ time using a total of $O(m^2)$ processors on the EREW PRAM model.

Step 2. In the previous step, for each edge of the graph we have computed a collection of pointers to its edge-separators. Then, by using list ranking or parallel prefix sums, we can rank each edge-separator in the list or array of the edge-separators of each edge. If we use parallel prefix sums on an array which stores the number of edge-separators per edge and use the ranking we mentioned earlier, we can produce an array of all the edge-separators without concurrent writes. Thus, this step takes $O(\log n)$ time using $O(m^2/\log n)$ processors on the EREW PRAM model.

Step 3: In this step we need to identify and select the distinct entries $\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_\ell$ of the list $\mathcal{S}$ or, equivalently, to remove the duplicates from a copy of the list $\mathcal{S}$; this can be done by sorting the array of all the edge-separators, and then by comparing adjacent entries of the sorted array. Two edge-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 $n_{i,j} = \min\{n_i, n_j\}$ vertices; if they do not match, we readily obtain an ordering of the two edge-separators, whereas if they match, then the edge-separator with the fewest vertices is considered smaller. Such a comparison takes $O(\log n_{i,j})$ time using $O(n_{i,j}/\log n_{i,j})$ processors or $O(\log n)$ time using $O(n_i/\log n)$ processors. 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 edge-separators takes $O(\log^2 n)$ time using

$O(m^2/\log n)$ processors; recall that $\sum_i n_i = O(m^2)$ in accordance with Lemma 5.1. Finally, we remove the duplicates; two edge-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 edge-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. Comparing consecutive entries and marking the duplicate ones takes $O(\log n)$ time using $O(m^2/\log n)$ processors (note that in order to guarantee Exclusive-Read execution, the processing is performed in two phases: in the first, we process all pairs of consecutive entries located in positions $2i+1$ and $2i+2$, $i \geq 0$; in the second, we process all the remaining pairs). Finally, array packing brings the distinct edge-separators in consecutive positions in the array; 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. Then the number of distinct edge-separators $\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_\ell$ can be easily extracted from the packed array.

In total, Step 3 is executed in $O(\log^2 n)$ time using $O(m^2/\log n)$ processors on the EREW PRAM model of computation.

Step 4. This step is executed for each of the edge-separators $\widehat{S}_1, \ldots, \widehat{S}_\ell$; from Observation 5.1, their number ℓ does not exceed $n+m$. As in Step 1, we make ℓ copies of the adjacency-list representation of the graph G in order to achieve an EREW execution of this step of the algorithm.

Substep 4.1. The subgraph $G[\widehat{S}_i]$ can be constructed from a copy of the adjacency-list representation of G, where records of vertices not in $\widehat{S}_i$ are marked useless and are removed by means of array packing. The graph can be constructed in $O(\log n)$ time using $O(m)$ processors. Note that arrays need to be built to hold the transformations from the old indices to the new indices of the graph $G[\widehat{S}_i]$ and back. This too can be executed in the above stated time and processor complexity.

Substep 4.2. Here we compute the co-connected components of the graph $G[\widehat{S}_i]$; let n_i and m_i be the numbers of vertices and edges of $G[\widehat{S}_i]$. This computation can be done in $O(\log n_i)$ time with $O((n_i+m_i)/\log n_i)$ processors or in $O(\log n)$ time with $O((n_i+m_i)/\log n)$ processors on the EREW PRAM using algorithm Par_Co-components of the previous section; the array `co-comp[ ]` returned by algorithm Par_Co-components is copied in an array $co\text{-}C_i[\,]$ of size n such that $co\text{-}C_i[u] = j$ if u belongs to the jth co-component of $G[\widehat{S}_i]$, and $co\text{-}C_i[u] = 0$ if u is not a vertex of $G[\widehat{S}_i]$. Since we have at most $n+m$ distinct edge-separators, each having fewer than n vertices and fewer than m edges, we have that for all the distinct edge-separators this substep takes $O(\log n)$ time with $O(m^2/\log n)$ processors.

Thus, the entire Step 4 is executed in $O(\log n)$ time with $O(m^2/\log n)$ processors on the EREW PRAM model of computation.

Step 5. This step is executed for each edge-separator S_i in the list $\mathcal{S}$. We use three auxiliary arrays, namely, $A_i[\,]$ of length $|S_i|$, and $B[\,]$ and $M[\,]$ of length m^2 each. For each $A_i[\,]$, the goal is to set its entries as follows: if S_i has been contributed by the edge e of G, is a copy of some $\widehat{S}_j$, and its vertices are $u_1, u_2, \ldots, u_{p_i}$, then, for $1 \leq q \leq p_i$, $A_i[q] = \big(i, co\text{-}C_j[u_q], 1\big)$ if $u_q \in A(e; x)$, $A_i[q] = \big(i, co\text{-}C_j[u_q], 2\big)$ if $u_q \in A(e; y)$, and $A_i[q] = \big(i, co\text{-}C_j[u_q], 0\big)$ otherwise; in light of the definition of the field $t(e, u_q)$ associated with each occurrence of the vertex u_q (see Substep 1.1), we have

that $A_i[q] = \big(i, co\text{-}C_j[u_q], t(e, u_q)\big)$. To avoid concurrent reading while updating the arrays $A_i[\,]$, we first construct these arrays for each of the edge-separators $\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_\ell$; thanks to the arrays $co\text{-}C_j[\,]$ and the field $t(\,,\,)$, this takes $O(1)$ time and requires $\sum_j |\widehat{S}_j| = O(n\ell)$ processors.

Next, for each edge-separator S_i, which is a copy of some $\widehat{S}_j = S_h$, the array $A_i[\,]$ is initialized as a copy of $A_h[\,]$, while the final values of $A_i[\,]$ can be obtained by setting the first field of each of its entries to i, and the third field to the correct $t(\,,\,)$ (note that although S_i and S_h are duplicates, they have been contributed by different edges, say, e and e', and thus the values of $t(e, w)$ and $t(e', w)$ for any of their vertices w may differ). Thus, the computation of the arrays $A_i[\,]$ can be completed in $O(1)$ time using $|S_i|$ processors.

Next, we copy the elements of all the arrays $A_i[\,]$ into the array $B[\,]$, and the array $B[\,]$ is sorted lexicographically; copying and sorting take $O(\log n)$ time with $O(m^2)$ processors. The array $M[\,]$ is filled by processing the elements of the array $B[\,]$ as follows: we set $M[1] \leftarrow 0$; for every $i = 2, 3, \ldots, m^2$, we set $M[i] \leftarrow 1$ if the elements $B[i-1] = (a, b, c)$ and $B[i] = (a', b', c')$ have the property $a = a'$, $b = b'$, $c = 1$, and $c' = 2$, otherwise, we set $M[i] \leftarrow 0$. The computation of the array $M[\,]$ can be completed in $O(1)$ time with $O(m^2)$ processors. Then the input graph G is not a weakly triangulated graph if and only if there exists an entry of the array $M[\,]$ equal to 1; this test can be done by computing the maximum entry of $M[\,]$ in $O(\log n)$ time using $O(m^2/\log n)$ processors.

The above description implies that the entire Step 5 is executed in $O(\log n)$ time using $O(m^2)$ processors on the EREW PRAM model of computation.

Step 6. This step takes $O(1)$ time using one processor.

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 EREW PRAM model. Thus, we have the following result.

Lemma 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 the EREW PRAM model.*

It is worth noting that all steps of algorithm WT_REC except for Step 3 can be executed in $O(\log n)$ parallel time; Step 3 necessitates $O(\log^2 n)$ time.

If the input graph is not connected, then we compute its connected components by using Chong et al.'s algorithm [8], and then apply the above algorithm on each of the components; we note that working on each component necessitates re-indexing. Since, for a graph on n vertices and m edges, both Chong et al.'s algorithm as well as the re-indexing take $O(\log n)$ time using $O(n + m)$ processors on the EREW PRAM, the following result is established.

Theorem 5.2. *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 the EREW PRAM model.*

Given that the currently fastest sequential algorithms for recognizing weakly triangulated graphs run in $O(m^2)$ time [4], [15], our parallel algorithm is cost-efficient.

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 optimal $O(\log n)$-time and $O((n+m)/\log n)$-processor EREW PRAM parallel algorithm. The same approach can be used to yield efficient sequential and parallel algorithms for biconnected components and strongly connected components of the complement of undirected and directed graphs, respectively. We also 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 EREW PRAM model of computation.

Due to the work of Chong et al. [8], the connected 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 EREW PRAM model. Thus, since our co-connectivity EREW PRAM algorithm computes the co-connected components of a graph for an optimal cost $O(n+m)$, it is reasonable to ask whether the time-processor complexity of the parallel connectivity algorithm of [8] can be improved to achieve an optimal cost $O(n+m)$, with preservation of the EREW PRAM model. We pose this as an open problem.

Our parallel algorithm for recognizing weakly triangulated graphs runs in $O(\log^2 n)$ time on the EREW PRAM model, for a cost of $O((n+m^2)\log n)$ and, thus, it is cost-efficient due to the work of Hayward et al. [15] and Berry et al. [4]. It is interesting to investigate whether there exist $O(\log n)$-time or $O(\log^2 n)$-time cost-optimal EREW PRAM algorithms for recognizing weakly triangulated graphs.

References

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

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

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

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

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

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

[7] K.W. Chong, Y. Han, Y. Igarashi, and T.W. Lam, Improving the efficiency of parallel minimum spanning tree algorithms, *Discrete Appl. Math.* **126**, 33–54, 2003.

[8] K.W. Chong, Y. Han, and T.W. Lam, Concurrent threads and optimal parallel minimum spanning trees algorithm, *J. Assoc. Comput. Mach.* **48**(2), 297–323, 2001.

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd edition, MIT Press, Cambridge, MA, 2001.
[10] E. Dahlhaus, J. Gustedt, and R.M. McConnell, Efficient and practical algorithms for sequential modular decomposition, *J. Algorithms* **41**, 360–387, 2001.
[11] G.A. Dirac, On rigid circuit graphs, *Abh. Math. Sem. Univ. Hamburg* **25**, 71–76, 1961.
[12] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.
[13] R.B. Hayward, Weakly triangulated graphs, *J. Combin. Theory Ser. B* **39**, 200–208, 1985.
[14] R.B. Hayward, Meyniel weakly triangulated graphs—I: co-perfect orderability, *Discrete Appl. Math.* **73**, 199–210, 1997.
[15] R.B. Hayward, J. Spinrad, and R. Sritharan, Weakly chordal graph algorithms via handles, *Proc.* 11*th ACM–SIAM Symp. on Discrete Algorithms* (*SODA* '00), pp. 42–49, 2000.
[16] D.S. Hirschberg, Parallel algorithms for the transitive closure and the connected components problems, *Proc.* 8*th ACM Symp. on Theory of Computing* (*STOC* '76), pp. 55–57, 1976.
[17] D.S. Hirschberg, A.K. Chandra, and D.V. Sarwate, Computing connected components on parallel computers, *Comm. ACM* **22**, 461–464, 1979.
[18] C.T. Hoàng, On the complexity of recognizing a class of perfectly orderable graphs, *Discrete Appl. Math.* **66**, 219–226, 1996.
[19] H. Ito and M. Yokoyama, Linear time algorithms for graph search and connectivity determination on complement graphs, *Inform. Process. Lett.* **66**, 209–213, 1998.
[20] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.
[21] 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.
[22] D. Nath and S.N. Maheshwari, Parallel algorithms for the connected components and minimal spanning trees, *Inform. Process. Lett.* **14**(1), 7–11, 1982.
[23] S.D. Nikolopoulos and L. Palios, Hole and antihole detection in graphs, *Proc.* 15*th ACM–SIAM Symp. on Discrete Algorithms* (*SODA* '04), pp. 843–852, 2004.
[24] J. Reif (ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann, San Mateo, CA, 1993.
[25] C. Savage and J. JáJá, Fast, efficient parallel algorithms for some graph problems, *SIAM J. Comput.* **10**, 682–691, 1981.
[26] Y. Shiloach and U. Vishkin, An $O(\log n)$ parallel connectivity algorithm, *J. Algorithms* **3**, 57–67, 1982.
[27] J.P. Spinrad and R. Sritharan, Algorithms for weakly triangulated graphs, *Discrete Appl. Math.* **59**, 181–191, 1995.

Received March 6, 2002, *and in revised form November* 19, 2002, *and in final form July* 3, 2003.
Online publication March 1, 2004.