Fully Dynamic Maintenance of k-Connectivity in Parallel

Weifa Liang[†] Richard P. Brent[‡] Hong Shen[§]

† Department of Computer Science

Australian National University, Canberra, ACT 0200, Australia

wliang@cs.anu.edu.au

‡ Oxford University Computing Laboratory

Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom

Richard.Brent@comlab.ox.ac.uk

§ School of Computing and Information Technology

Griffith University, Nathan, QLD 4111, Australia

hong@cit.gu.edu.au

Abstract

Given a graph $G = (V, E)$ with n vertices and m edges, the k-connectivity of G denotes either the k-edge connectivity or the k-vertex connectivity of G. In this paper we deal with the fully dynamic maintenance of k-connectivity of G in the parallel setting for $k = 2, 3$. We study the problem of maintaining k-edge/vertex connected components of a graph undergoing repeatedly dynamic updates such as edge insertions and deletions, and answering the query of whether two vertices are included the same k-edge/vertex connected component. Our major results are the following. (i) An NC algorithm for the 2-edge connectivity problem is proposed, which runs in $O(\log n \log(m/n))$ time using $O(n^{3/4})$ processors per update and query. (ii) It is shown that the biconnectivity problem can be solved in $O(\log^2 n)$ time using $O(n\alpha(2n, n)/\log n)$ processors per update and $O(1)$ time with a single processor per query, or in $O(\log n \log \frac{m}{n})$ time using $O(n\alpha(2n, n)/\log n)$ processors per update and $O(\log n)$ time using $O(n\alpha(2n, n)/\log n)$ processors per query, where $\alpha(.,.)$ is the inverse of Ackermann's function. (iii) An NC algorithm for the triconnectivity problem is also derived, which takes $O(\log n \log \frac{m}{n} + \log n \log\log n/\alpha(3n, n))$ time using $O(n\alpha(3n, n)/\log n)$ processors per update and $O(1)$ time with a single processor per query. (iv) An NC algorithm for the 3-edge connectivity problem is obtained, which has the same time and processor complexities as the algorithm for the triconnectivity problem. To the best of our knowledge, the proposed algorithms are the first NC algorithms for the problems using $O(n)$ processors in contrast to $\Omega(m)$ processors for solving them from scratch. In particular, the proposed NC algorithm for the 2-edge connectivity problem uses only $O(n^{3/4})$

processors. All the proposed algorithms run on a CRCW PRAM.

Keywords: *NC algorithms, 2-edge/vertex connectivity, 3-edge/vertex connectivity, dynamic data structures, parallel algorithm design and analysis, graph problems.*

1 Introduction

A *fully dynamic graph algorithm* is one that allows edge insertions and deletions, and recomputes a desired graph property quickly after each such update. A *partially dynamic graph algorithm* is one that allows edge insertion updates only. In this paper we focus on the design of fully dynamic NC algorithms for maintaining the k-connectivity of graphs. Given a graph $G = (V, E)$ with n vertices and m edges, the *k-connectivity* of G refers to either the *k-edge connectivity* or *k-vertex connectivity* [18], which will be formal defined in Section 2.1.

1.1 Previous related results

k-connectivity of G is a basic property in graph theory with wide applications in robust routing, distributed computing, reliable communications networks design, etc. There have been extensive studies of algorithms for dynamically maintenance of the k-connectivity of graphs with $k \leq 4$. For example, Even et. al. [12] studied the connected component problem ($k = 1$) in the early 1980s. Frederickson [15] studied the minimum spanning tree problem by giving a fully dynamic algorithm with $O(\sqrt{m})$ time per update and $O(1)$ time per query. Eppstein et. al. [10] improved Frederickson's algorithm by presenting an $O(\sqrt{n}\log(m/n))$ algorithm, using a sparsification technique. Later, Eppstein et. al. [11] improved their own algorithm further by an $O(\log(m/n))$ factor. (Rauch) Henzinger et. al. [25] gave an algorithm for fully dynamic maintenance of minimum spanning trees with $O(\sqrt[3]{n}\log n)$ time per update and constant time per query. Randomization has also been used to improve the time complexity of fully dynamic maintenance of connected components. Henzinger et. al. [24, 23] showed that a spanning forest can be maintained in $O(\log^3 n)$ expected amortized time per update, and the query can be answered in $O(\log n/\log\log n)$ time. The update time was further improved to $O(\log^2 n)$ by Henzinger et. al [27]. Since the initial submission of this paper in 1995, there has been further development in this topic. In particular, Holm et. al. [28] recently gave an improved algorithm for the fully dynamic maintenance of connected components with $O(\log^2 n)$ amortized time per update and $O(\log n/\log\log n)$ time per query, and an improved algorithm for the fully dynamic maintenance of minimum spanning forests with $O(\log^4 n)$ amortized time per update and $O(\log n/\log\log n)$ time per query, thus matching the previously best randomized bound and improving substantially the previously best deterministic bounds of $O(\sqrt[3]{n}\log n)$. Note that Holm et. al.'s algorithm assumes that the initial graph is empty, and the update time is the amortized time. Thus, a single insertion or deletion operation in their algorithm takes $\Omega(n)$ time in the worst case. In the parallel setting, Ferragina [13] studied fully dynamic maintenance of connected components on an EREW PRAM provided only a small number of processors are used, and the proposed parallel algorithm requires $O((\frac{n \log n}{p} + p^{1+\epsilon})(\frac{\log f}{\log \log n}))$ time using $O(p)$ processors per update, where $p \leq (n \log n)^{1/(2+\epsilon)}$, $f = \frac{m}{n \log n}$, and $0 \leq \epsilon < 1$. Das et. al. [8] and Ferragina [14] developed parallel algorithms for the fully dynamic maintenance of minimum spanning trees and connected components, which requires $O(\log n)$ time and the total of $O(n^{2/3} \log \frac{m}{n})$ work. Independently, Liang et. al. [35] also proposed parallel algorithms for the same problem which requires $O(\log n \log \frac{m}{n})$ time and $O(n^{2/3})$ processors.

As Galil et. al. [19] pointed out, the fully dynamic maintenance of 2-connectivity, however, is much harder than the fully dynamic maintenance of connected components because the maintenance of connected components only involves merging two connected components into one connected component or splitting a connected component into two connected components per update, whereas the maintenance of 2-edge/vertex connected components may need merging $O(n)$ 2-edge/vertex connected components into one or splitting a 2-edge/vertex connected component into $O(n)$ 2-edge/vertex connected components per update. Westbrook et. al. [45] presented the first partially dynamic algorithms for the maintenance of both 2-edge connected components and biconnected (2-vertex connected) components, which require $O(m\alpha(m,n))$ time for the maintenance of 2-connectivity of graphs when there are m queries and edge insertions. In other words, the algorithm takes $O(\alpha(m,n))$ amortized time per update or per query, where $\alpha(m,n)$ is the inverse of Ackermann's function. Westbrook et. al. left an open problem as to whether there exists a sublinear time, fully dynamic algorithm for the 2-connectivity problem. Since then, finding a sublinear time algorithm for the problem has been a challenging task. Galil et. al. [19] presented the first sublinear time algorithm for 2-edge connectivity, by using Frederickson's clustering technique and the other techniques. Their algorithm requires $O(m^{2/3})$ time. Later Frederickson [16] improved this algorithm to $O(\sqrt{m})$ using an *ambivalent* data structure. Eppstein et. al. [10, 11] further improved Frederickson's algorithm by developing $O(\sqrt{n}\log(m/n))$ and $O(\sqrt{n})$ time algorithms, based on their sparsification technique. They also derived algorithms for the fully dynamic maintenance of biconnectivity and triconnectivity, which take $O(\alpha(q,n))$ amortized time per insertion or per query, and $O(n)$ time per deletion, where q is the total number of queries made. Furthermore, Henzinger et. al. [24] generalized their randomization technique for connected components by giving an $O(\log^5 n)$ expected amortized time bound per update. Recently Holm et. al. [28] extended their results for connected components by presenting an algorithm for the fully dynamic maintenance of 2-edge connected components. Their algorithm requires $O(\log^4 n)$ amortized time per update.

The fully dynamic maintenance of biconnected components seems much harder than that of 2-edge connected components. There is a reduction technique [18] which can reduce a k-edge connectivity problem to a k-vertex connectivity problem, but the reverse reduction has not yet been found. In 1992 Rauch [40] presented the first fully dynamic algorithm for the maintenance of biconnected components. Her algorithm takes $O(m^{2/3})$ amortized time per update and $O(1)$ time per query. She later improved her algorithm to $O(\min\{\sqrt{m}\log n, n\})$, using new data structures and the sparse certificate technique [41]. In 1995 Henzinger et. al. [26] further improved her result to $O(\sqrt{n \log n} \log \frac{m}{n})$. Later Henzinger et. al. [23, 24] presented polylogarithmic time randomized algorithms for the fully dynamic maintenance of biconnectivity [23], using a novel decomposition of graphs and the randomization technique. Their algorithm requires $O(\Delta \log^4 n)$ expected amortized time per update, where Δ is the maximum degree of the graph. Holm et. al. [28] further improved the above results by giving a deterministic algorithm, which takes $O(\log^4 n)$ amortized time per update.

When $k = 3$, Galil et. al. [20] presented a partially dynamic algorithm for the maintenance of 3-edge connected components. Their algorithm takes $O((n + q)\alpha(q, n))$ time for a sequence of q queries and updates on an n-vertex graph. Independently, La Poutré et. al. [34] presented a partially dynamic algorithm which has the same time bound as Galil et. al. [20]. Later Galil et. al. [21] claimed that they developed a fully dynamic algorithm for the problem which requires $O(m^{2/3})$ time per update and per query by minor modifications to their 2-edge connectivity algorithm [19]. Moreover, La Poutré [33] also proposed a partially dynamic algorithm for the maintenance of triconnected components, which needs $O((n + q)\alpha(q, n))$ time for a sequence of q queries and updates. As for $k > 3$, there are only a few results available. Kanevsky et. al. [30] presented a partially dynamic algorithm for the maintenance of 4-vertex connected components which requires $O(\alpha(q, n))$ amortized time per update and per query provided that the graph is triconnected, where q is the number of operations performed. Dinitz [9] presented a partially dynamic algorithm for the maintenance of 4-edge connected components. Note that using Eppstein et. al's [10, 11] sparsification technique and the above algorithms, the following results then follow easily: (i) fully dynamic maintenance of 3-edge connected components can be done in $O(n^{2/3})$ time per update and per query; (ii) fully dynamic maintenance of triconnected components can be done in $O(\alpha(q, n))$ time per insertion or per query, and in $O(n)$ time per deletion; (iii) fully dynamic maintenance of 4-vertex connected components can be done in $O(\log n)$ time per insertion, in $O(n \log n)$ time per deletion, and in $O(1)$ time per query.

Surprisingly, we have not seen any NC algorithms for the fully dynamic maintenance of k-connectivity with $k \geq 2$ in the parallel environment. In particular, we have not seen any NC algorithms for the addressed problems using $O(n)$ processors only. In this paper we will focus on developing such NC algorithms for the problems for the first time. It must be mentioned that the previously known sequential algorithms except the algorithms by Henzinger et. al. [23, 24] and Holm et. al. [28] are hardly parallelizable since they use the restricted clustering decomposition and depth-first search technique, which are inherently sequential. The algorithms by Henzinger et. al. [23, 24] are Las-Vegas type of randomized algorithms, which are incomparable to the deterministic algorithms, because the deterministic simulation of the randomized algorithms may require $\Omega(m)$ amount of work. The algorithm due to Holm et. al. [28] does guarantee that each update can be done in $O(\log^4 n)$ amortized time, but the time complexity is amortized, which means that all the data structures must be rebuilt after undergoing $\Omega(m)$ updates. The rebuild takes $\Omega(n)$ time. In other words, it takes $\Omega(n)$ time per update in the worst case. Therefore, even if their algorithm can be parallelized, $\Omega(n)$ processors must be required in order to achieve a polylogarithmic time complexity per update in the worst case.

1.2 Our contributions

In this paper we deal with the fully dynamic maintenance of k-edge/vertex connectivity property of a graph in the parallel environment with $k = 2, 3$, after undergoing an intermixed sequence of insertion, deletion and query operations, which are described as follows.

- *SameConnComp(x,y):* return *true* if vertices x and y are in the same k-edge/vertex connected component; otherwise, return *false*.
- *InsertEdge(x,y):* insert a new edge between vertex x and vertex y.
- *DeleteEdge(x,y):* delete an existent edge between vertex x and vertex y.

We present NC algorithms for the fully dynamic maintenance of k-edge/vertex connected components in G for $k = 2, 3$ by using $O(n)$ processors in contrast to the usual $\Omega(m)$ processors needed for computing these properties from scratch. Our major results are as follows: (i) an NC algorithm for the 2-edge connectivity problem is presented, which requires $O(\log n \log(m/n))$ time using $O(n^{3/4})$ processors per update and per query; (ii) an NC algorithm for the biconnectivity problem is also given, which requires either $O(\log^2 n)$ time using $O(n\alpha(2n, n)/\log n)$ processors per update and $O(1)$ time with a single processor per query, or $O(\log n \log \frac{m}{n})$ time using $O(n\alpha(2n, n)/\log n)$ processors per update and $O(\log n)$ time with $O(n\alpha(2n, n)/\log n)$ processors per query; (iii) an NC algorithm for the triconnectivity problem is derived, which requires $O(\log n \log \frac{m}{n} + \log n \log\log n/\alpha(3n, n))$ time using $O(n\alpha(3n, n)/\log n)$ processors per update and $O(1)$ time with a single processor per query; (iv) an NC algorithm for the 3-edge connectivity problem is obtained, which has the same time and processor complexities as the algorithm for the fully dynamic maintenance of triconnectivity. To the best of our knowledge, the proposed algorithms for the above problems are the first NC algorithms that use only $O(n)$ processors in contrast to $\Omega(m+n)$ processors for solving them from scratch. At the same time, it should be mentioned that the basic idea of the proposed algorithm for the fully maintenance of 2-edge connectivity comes from Galil et. al's sequential algorithm, although their algorithm by itself is not easily parallelizable. Our NC algorithm for this problem is achieved by replacing the sequential part of their algorithm with a highly parallelizable approach.

In this paper the following three parallel computational models will be used: (i) an Exclusive Read and Exclusive Write (EREW for short) PRAM in which concurrent read and concurrent write are forbidden; (ii) a CREW PRAM in which Concurrent Read is allowed but only Exclusive Write is permitted; and (iii) a CRCW PRAM in which both Concurrent Read and Concurrent Write are allowed, but the write conflicts are resolved arbitrarily.

The remainder of this paper is organized as follows. In Section 2 we introduce some basic concepts with respect to the k-edge/vertex connectivity such as the sparse k-edge/vertex certificates, Galil et. al's reduction technique [18], and Eppstein et. al's sparsification technique [10]. In Section 3, for the sake of simplicity, we consider the fully dynamic maintenance of 2-edge connected components in a *connected* graph. The data structures used for the maintenance are described, and the algorithms for connectivity queries, edge insertions, and edge deletions are presented. In Section 4 we first extend the results in connected graphs to disconnected graphs, and then present an improved NC algorithm for fully dynamic maintenance of 2-edge connectivity in general graphs. We finally discuss the fully dynamic maintenance of 3-edge connected components. In Section 5 we deal with the fully dynamic maintenance of biconnected components and triconnected components, and we conclude our discussions in Section 6.

2 Preliminaries

2.1 Definitions

$G(V, E)$ is *k-edge connected* if it is still connected after deleting any $k-1$ edges from it. Assuming that G contains no less than $k+1$ vertices, G is *k-vertex connected* if it is still connected after deleting any $k-1$ vertices from it. If G is not k-edge/vertex connected, it contains k-edge/vertex connected components. Given a graph G and an integer k, a pair of vertices u and v in G is said to be *k-edge connected* if the removal of any $k-1$ edges from G leaves u and v connected. This is an equivalence relationship, and written as $\equiv_k$, i.e., if a pair of vertices x and y is k-edge connected, we write $x \equiv_k y$. The vertices in G are partitioned by this relationship into equivalence classes, and each equivalence class is a *k-edge connected component* in G.

It seems much more complicated to define the *k-vertex connected component* in G with $k > 3$. Here we only define this concept for $1 \leq k \leq 3$, which will be used in this paper. Given a graph $G(V, E)$, let $E_1, E_2, \ldots, E_h$ be an edge partition of E into equivalence classes such that two edges e' and e'' are in the same equivalence class if and only if either (i) $e' = e''$; or (ii) there is a simple cycle in G containing both e' and e''. For all i, $1 \leq i \leq h$, let V_i be the set of endpoints of the edges in E_i. Then, the subgraph $G_i = (V_i, E_i)$ is a *biconnected component* (or 2-vertex connected component) in G. An edge contained in no cycle is a biconnected component by itself and is referred to as a *bridge* in G. A graph G without including any bridge is called *2-edge connected.* If G contains bridges, then the removal of a bridge disconnects the graph. Usually, 2-vertex connectivity is called *biconnectivity*, and a biconnected component is called a *block.* If for every pair of vertices, there are two vertex disjoint paths in G between them, then G is called a *biconnected* graph. It is well known [1] that G is not biconnected if and only if there is a vertex $v \in V$ whose removal disconnects G. Such a v is called an *articulation point* of G.

Let $\{a,b\}$ be a pair of vertices in a biconnected graph G. Suppose the edges in G are partitioned into equivalence classes $E_1, E_2, \ldots, E_p$ such that two edges which lie on a common path not containing any vertex of $\{a,b\}$ except as an endpoint are in the same class. The classes E_i are called the *separation classes* of G with respect to $\{a,b\}$. If there are at least two separation classes, then $\{a,b\}$ is a *separation pair* of G unless (i) there are exactly two separation classes, one class consists of a single edge, or there are exactly three classes, each consisting of a single edge. If G is a biconnected graph that does not contain a separation pair, then G is *triconnected.* Let $\{a,b\}$ be a separation pair of G and $E_1, E_2, \ldots, E_p$ be the separation classes of G with respect to $\{a,b\}$. Let $E' = \bigcup_{i=1}^{k} E_i$ and $E'' = \bigcup_{i=k+1}^{p} E_i$ and $|E'| \geq 2$ and $|E''| \geq 2$. Let $G_1 = (V(E'), E' \cup \{(a,b)\})$ and $G_2 = (V(E''), E'' \cup \{(a,b)\})$ where $V(E')$ and $V(E'')$ are the sets of endpoints of edges in E' and E''. G_1 and G_2 are called the *split graphs* of G with respect to $\{a,b\}$. Replacing G by the two split graphs is called *splitting* G. The new edges (a,b) added to both G_1 and G_2 are called *virtual edges.* If G is biconnected, then any split graph of G is also biconnected. Suppose G is split, the split graphs are split, and so on, until no more splits are possible. The graphs constructed in this way are called *split components* of G. The split components of G are of three types: triple bonds of form $(\{a,b\}, \{(a,b),(a,b),(a,b)\})$, triangles of form $(\{a,b,c\}, \{(a,b),(a,c),(b,c)\})$, and triconnected graphs. The split components of G are called *triconnected components* in G. A triconnected component usually is also called *3-vertex connected component*, and 3-vertex connectivity is called *triconnectivity.*

An inverted tree $T(V, E_T)$ is a directed tree rooted at a distinguished vertex r, $r \in V$, such that for every other vertex v $(v \neq r)$ there is a parent pointer pointing to the parent $F_T(v)$ of v, an edge $\langle v, F_T(v)\rangle \in E_T$, and $F_T(r) = r$. Given an inverted tree T and any two vertices v and u in it, the *lowest common ancestor* $LCA(u,v)$ of u and v is the first common vertex on the paths from u (v) to the root. It is known that every LCA query in T can be answered in $O(1)$ time using one processor on an EREW PRAM provided that T is proper preprocessed. The preprocessing takes $O(\log n)$ time and uses $O(n)$ processors on an EREW PRAM if T contains n vertices [42]. Note that all the trees used in this paper are maintained as inverted trees.

2.2 Transformation technique of graphs

We now introduce a technique to transform a general graph with arbitrary degree into a graph with bounded degree. It is well known [22] that a graph $G(V,E)$ can be transformed to another graph $G'(V',E')$ such that every vertex in G' has no degree greater than three. The transformation proceeds as follows. For each vertex u of degree $d \geq 4$ in G, assume that $v_0, v_1, \ldots, v_{d-1}$ are the adjacent vertices of u in G. Then G' is generated by replacing u with new vertices $u_0, u_1, \ldots, u_{d-1}$, adding edges $(u_i, u_{(i+1)\ mod\ d})$ and edges (u_i, v_i) to G', $0 \leq i \leq d-1$. One important property of the transformation is that it preserves the 2-edge connectivity of G [19]. That is, if two vertices are in a 2-edge connected component in G, then they will also be in the same 2-edge connected component in G'. Otherwise, they are still not in the same 2-edge connected component in G'.

It is clear that G' contains $O(m+n)$ edges and vertices and each vertex has degree no greater than three. Without loss of generality, in Section 3 we assume $G(V,E)$ is a graph with $O(m+n)$ vertices, and each vertex has degree no greater than three when dealing with the 2-edge connectivity. Otherwise, the transformation can be applied, which takes $O(\log n)$ time using $O(m+n)$ processors on an EREW PRAM.

2.3 The sparse k-edge (k-vertex) certificate

Let $G(V,E)$ be an arbitrary graph. An *edge separator* in G is such a subset of edges that the removal of the edges in it will disconnect G. A *sparse k-edge certificate* of G is a subgraph H in G containing $O(kn)$ edges, and every edge separator in H of size less than k is also an edge separator in G. A *sparse k-vertex certificate* of G can be defined similarly. In the following we show how to find a sparse k-edge/k-vertex certificate of G [38, 4].

Let T_1 be a maximal spanning forest in G and T_i be a maximal spanning forest of graph $G_i = G - \cup_{j=1}^{i-1} T_j$ for $i > 1$. Denote by $U_i = \cup_{j=1}^{i} T_j$, the union of the i maximal spanning forests $T_1, T_2, \ldots, T_i$. The graph U_k is a *sparse k-edge certificate* of G, and has the following property.

Lemma 1 *[38, 39] The graph U_k is l-edge connected if and only if G is l-edge connected, for any integer l with $1 \leq l \leq k$.*

Lemma 1 always holds no matter whether G is a simple graph or not. We now introduce the *scan-first search technique* [4] on G which will be used to find a sparse k-vertex certificate of G.

Definition 1 *A scan-first search in a connected undirected graph $G(V,E)$ starting from a specified vertex r is a systematic way of visiting the vertices in G. To scan a vertex is to visit all previously unvisited adjacent vertices of that vertex. At the beginning of the search, only r is visited. Then, the search iteratively scans an already visited but unscanned vertex until all vertices are scanned.*

If G is disconnected, a scan-first search spanning forest of G is then obtained by applying the scan-first search on each connected component in G. Cheriyan et. al. [4] presented an efficient parallel algorithm for finding a scan-first search forest in G, which proceeds as follows. It first finds an arbitrary spanning tree T in G. It then traverses T by labeling the vertices with their preorder numberings in T. It finally constructs the following scan-first tree ST. Let $b(v)$ be the parent of v in ST and $N(v)$ be the set of adjacent vertices of v in G, $b(v)$ is such a vertex in $N(v)$ that has the smallest preorder numbering for every $v \in V$, $v \neq r$.

Lemma 2 *[4] For a graph with n vertices and m edges, a scan-first search spanning forest can be found in $O(\log n)$*

time using $C(n,m)$ processors on a CRCW PRAM, where $C(n,m)$ is the number of processors used to find spanning trees in all connected components in $O(\log n)$ time.

Note that, currently the best result of $C(n,m)$ is $O((m+n)\alpha(m,n)/\log n)$ on a CRCW PRAM [6]. With the above preparation, the sparse k-vertex certificate of G is obtained as follows. For $i = 1, 2, \ldots, k$, let E_i be the edge set of a scan-first search spanning forest in graph $G_{i-1} = (V, E - (E_1 \cup \ldots \cup E_{i-1}))$, and $G_0 = (V, E)$. Then, the graph $C_k = E_1 \cup \ldots \cup E_k$ is a sparse k-vertex certificate of G, which contains n vertices and no more than $k(n-1)$ edges.

Lemma 3 *[4] The graph C_k defined is l-vertex connected if and only if G is l-vertex connected, for any integer l with $1 \leq l \leq k$.*

2.4 The sparsification technique

Eppstein et. al.'s sparsification technique [10] is based on a *certificate* concept. Before introducing the technique, we reproduce some of the definitions which are fundamental to the technique.

Definition 2 *For any graph property $\mathcal{P}$ and graph $G(V,E)$, a certificate for G is a graph G' such that G has the property $\mathcal{P}$ if and only if G' has the property.*

Definition 3 *For any graph property $\mathcal{P}$ and a graph $G(V,E)$, a strong certificate for G is a graph G' on the same vertex set such that, for any H, $G \cup H$ has the property $\mathcal{P}$ if and only if $G' \cup H$ has the property.*

Definition 4 *Let $\mathcal{A}$ be a function mapping graphs to strong certificates. Then $\mathcal{A}$ is stable if it has the following two properties: (1) for any graphs G and H, $\mathcal{A}(G \cup H) = \mathcal{A}(\mathcal{A}(G) \cup H)$; (2) for any graph G and edge $e \in E$, $\mathcal{A}(G - \{e\})$ differs from $\mathcal{A}(G)$ by $O(1)$ edges.*

The sparsification technique is described as follows. We partition the edges in $G(V,E)$ into $\lceil m/n \rceil$ groups, all except one contain exactly n edges. The remaining group called the *small* group may contain between 1 and n edges. When inserting an edge into G, we place it into the small group. When deleting an edge from a group, we move another edge from the small group into the group, to keep the group size invariant (i.e., n edges exactly). If deleting the last edge in the small group, we remove the small group entirely, and if inserting an edge to the small group while there are already n edges in it, we start a new small group.

Having finished edge grouping, we now establish a complete binary tree $\mathcal{BT}$ with $\lceil m/n \rceil$ leaf nodes, which correspond to the $\lceil m/n \rceil$ groups. $\mathcal{BT}$ is called the *sparsification tree* in which every *node* corresponds to a subgraph formed by the edges in the graphs at the leaf nodes of $\mathcal{BT}$ that are the descendants of the node. For every node in $\mathcal{BT}$, we maintain a sparse certificate $\mathcal{P}$ for it. The sparse certificate at a leaf node is obtained by applying the procedure of finding a sparse certificate to the subgraph consisting of the edges in the corresponding group. The sparse certificate at a non-leaf node is the union of the sparse certificates of its two children nodes.

Each update in G will cause either a single leaf node of $\mathcal{BT}$ to split or two leaf nodes to merge in the worst case. Since only the edge insertions and deletions are allowed, the number of edges in every group except the small group will never be changed. When an edge is inserted or deleted, only $O(1)$ groups are involved. For each of these groups and its $O(\log \frac{m}{n})$ ancestor nodes in $\mathcal{BT}$, we maintain the sparse certificates of these nodes dynamically. As results, the sparse certificate of G is the sparse certificate at the root of $\mathcal{BT}$.

2.5 The reduction technique

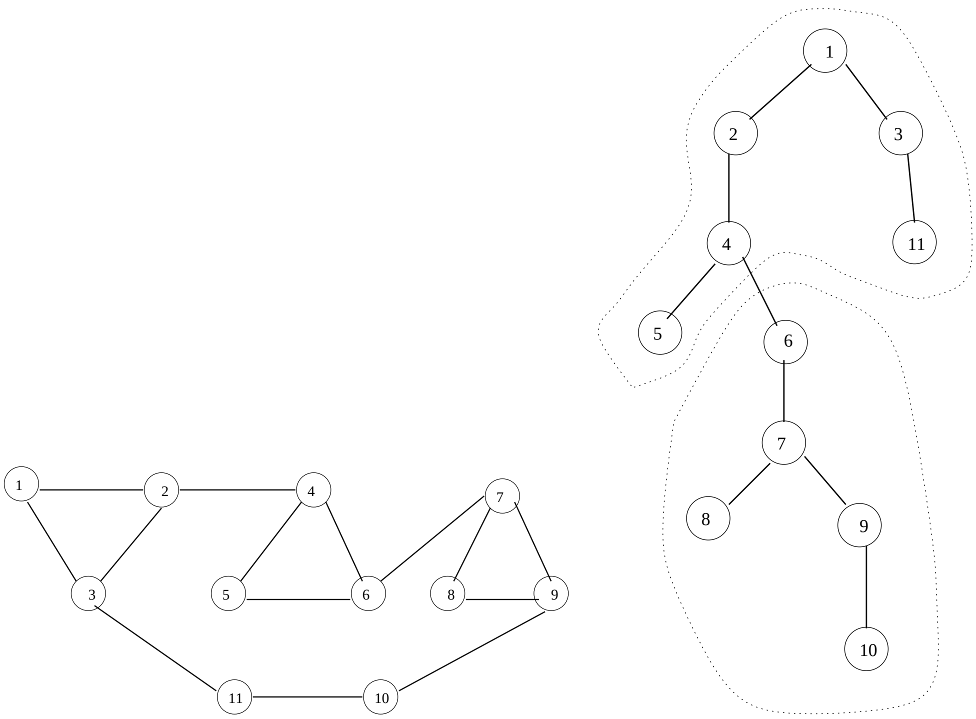
A reduction technique due to Galil et. al. [18], can reduce a k-edge connectivity problem in a graph $G = (V, E)$ to a k-vertex connectivity problem in another graph $\varphi_k(G) = (\varphi(V), \varphi(E))$, and $\varphi_k(G)$ is constructed as follows. For every vertex v in G, there are $k-2$ vertices $\varphi(v_1), \varphi(v_2), \ldots, \varphi(v_{k-2})$ in $\varphi_k(G)$, referred to as *node-vertices* of $\varphi_k(G)$. For every edge e in G, there is a vertex $\varphi(e)$ in $\varphi_k(G)$, referred to as *arc-vertex* of $\varphi_k(G)$. Let v be any vertex of G and $u_0, u_1, \ldots, u_{d-1}$ be the adjacent vertices of v in G, and $e_i = (v, u_i)$ be an edge in G, $0 \leq i \leq d-1$. Then, edges $(\varphi(e_i), \varphi(e_{(i+1) \ mod \ d}))$ and $(\varphi(e_i), \varphi(v_j))$ are in $\varphi_k(G)$, $0 \leq i \leq d-1$, $0 \leq j \leq k-2$. Notice that $\varphi_k(G)$ is obtained from G by replacing v and its incident edges $e_0, e_1, \ldots, e_{d-1}$ by a wheel with $k-2$ hubs $\varphi(v_1), \varphi(v_2), \ldots, \varphi(v_{k-2})$, and vertices $\varphi(e_0), \varphi(e_1), \ldots, \varphi(e_{d-1})$. This implies that $\varphi_k(G)$ has $O(m + kn)$ vertices and $O(km)$ edges. Clearly, $\varphi_k(G)$ can be constructed in time $O(\log n)$ time using $O(k(m+n))$ processors on an EREW PRAM if G contains $O(n)$ vertices and $O(m)$ edges. The graph $\varphi_k(G)$ has the following property.

Theorem 1 *[18] $G(V, E)$ is k-edge connected if and only if $\varphi_k(G)(\varphi(V), \varphi(E))$ is k-vertex connected.*

Following Theorem 1, to check whether two vertices x and y in G are k-edge connected can be done through checking whether their corresponding node-vertices $\varphi(x_1)$ and $\varphi(y_1)$ in $\varphi_k(G)$ are k-vertex connected.

3 Maintaining 2-Edge Connected Components in Connected Graphs

For the sake of simplicity, in this section we assume that $G(V, E)$ is connected, and contains $O(m+n)$ vertices and edges. Every vertex in G has degree no greater than 3. Otherwise, the transformation technique will be applied in order to obtain a graph with the degree no more than three. Figure 1(a) shows such a graph.



(a) A graph $G(V, E)$

(b) A spanning tree and a vertex cluster partition of G

Figure 1: A graph and its spanning tree and the vertex cluster partition.

3.1 Overview

We present our results in the following order. First, we decompose G into a number of disjoint connected subgraphs, and build data structures for each of them to keep the information on the subgraph. Also, we build a data structure to keep the information among the subgraphs, which is referred to as the *compact representation* $\hat{\mathcal{G}}$ of G. $\hat{\mathcal{G}}$ is the *super graph* of G, defined as follows. Each vertex in $\hat{\mathcal{G}}$ corresponds to a subgraph of G. There is an edge between two vertices in $\hat{\mathcal{G}}$ if there are edges in G between the vertices in the two corresponding subgraphs. For any two subgraphs in G, we use two data structures to keep the edge information between them. Then, we assume that all the data structures for 2-edge connected components in each subgraph have been already built. Now consider a query which asks whether two given vertices x and y are in the same 2-edge connected component in G. To answer this query, we proceed as follows. Let x be in $\mathcal{C}_x$ and y be in $\mathcal{C}_y$. We check whether $\mathcal{C}_x$ and $\mathcal{C}_y$ are in the same 2-edge connected component in $\hat{\mathcal{G}}$. If yes, then x and y are in the same 2-edge connected component in G. Otherwise, for every articulation point $\mathcal{C}$ in $\hat{\mathcal{G}}$ separating $\mathcal{C}_x$ from $\mathcal{C}_y$, we examine the induced subgraph of G by the vertices in $\mathcal{C}$ to see whether there is a potential bridge internal to the subgraph separating x from y in G. If yes, then x and y are not in the same 2-edge connected component in G. Finally, we deal with the fully dynamic maintenance of the data structures when there is an edge insertion or deletion. Note that each insertion/deletion only involves updating a constant number of subgraphs and $\hat{\mathcal{G}}$. Accordingly, we only update those involved subgraphs and $\hat{\mathcal{G}}$ instead of the entire graph G, which is much cheaper in terms of the maintenance cost because of the smaller sizes of these subgraphs and $\hat{\mathcal{G}}$ compared to G. Therefore, it is possible to reduce the maintenance cost to respond the updates and queries.

The rest of this section is organized as follows. Section 3.2 introduces the data structures that the proposed algorithm uses; and Section 3.3 presents algorithms for constructing the data structures. Section 3.4 proposes an algorithm for query processing, using the proposed data structures. Section 3.5 shows how to maintain the data structures efficiently when there are edge insertions and deletions.

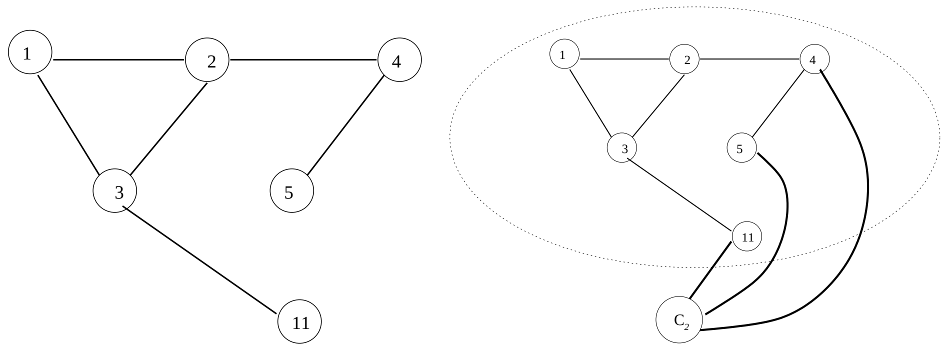
3.2 Data structures for 2-edge connected components

The proposed algorithm basically is a parallel implementation of Galil et. al.'s algorithm [19]. Therefore, most data structures used in their algorithm are adopted. Let $G(V, E)$ be a connected graph with the maximum degree three. The *vertex cluster partition* of G is described as follows. Following the vertex clustering idea by Frederickson [15], the vertex set V is partitioned into $V_1, V_2, \ldots, V_p$ disjoint subsets such that (i) $\lfloor K/2 \rfloor \leq |V_i| \leq \lceil 3K/2 \rceil$ for all i, $1 \leq i \leq p$; and (ii) the induced subgraph of G by the vertices in V_i is connected, where K is a parameter to be determined later. $V = \cup_{i=1}^{p} V_i$ and $p = O(m/K)$ because G contains $O(m+n)$ vertices and each vertex in it has a degree no greater than three. Each vertex subset V_i is a *vertex cluster* (or cluster for short) of G. The size of a cluster is the number of vertices in it. However, the size of each cluster is $O(K)$. For example, a spanning tree in G is shown in Figure 1(b). The vertex set of G is partitioned into two vertex clusters $\mathcal{C}_1 = \{1, 2, 3, 4, 5, 11\}$ and $\mathcal{C}_2 = \{6, 7, 8, 9, 10\}$ with $K = 4$.

3.2.1 Data structures pertinent to a vertex cluster $\mathcal{C}$

Induced subgraph $\mathcal{G}(\mathcal{C})$ of $\mathcal{C}$: Let $\mathcal{C}$ be a vertex cluster. An edge in G with its both endpoints in $\mathcal{C}$ is said to be *internal* to $\mathcal{C}$, while an edge in G with only one endpoint in $\mathcal{C}$ is said to be *incident* to $\mathcal{C}$. The induced subgraph of G by the vertices in $\mathcal{C}$ is denoted by $\mathcal{G}(\mathcal{C})$. For the above example, $\mathcal{G}(\mathcal{C}_1)$ is shown in Figure 2(a). Edge $(2, 4)$ in G is internal to $\mathcal{C}_1$, while edge $(4, 6)$ in G is incident to $\mathcal{C}_1$.

Full representation $\mathcal{F}(\mathcal{C})$ of $\mathcal{C}$: The *full representation* $\mathcal{F}(\mathcal{C})$ of a vertex cluster $\mathcal{C}$ is a simple graph, which consists of $\mathcal{G}(\mathcal{C})$ a vertex for each vertex cluster adjacent to $\mathcal{C}$, and the edges incident to $\mathcal{C}$. In other words, the vertex set of $\mathcal{F}(\mathcal{C})$ consists of the vertices internal to $\mathcal{C}$ and the corresponding vertices of the vertex clusters which are incident to $\mathcal{C}$. The edge set of $\mathcal{F}(\mathcal{C})$ is composed of all the edges in G internal and incident to $\mathcal{C}$. Figure 2(b) shows $\mathcal{F}(\mathcal{C}_1)$, where $\mathcal{C}_2$ is the cluster adjacent to $\mathcal{C}_1$.



(a) The induced subgraph $\mathcal{G}(\mathcal{C}_1)$ of G by the vertices in cluster $\mathcal{C}_1$

(b) The full presentation $\mathcal{F}(\mathcal{C}_1)$ of a vertex cluster $\mathcal{C}_1$ where $\mathcal{C}_1 = \{1, 2, 3, 4, 5, 11\}$

Figure 2: The induced subgraph $\mathcal{G}(\mathcal{C}_1)$ and the full representation $\mathcal{F}(\mathcal{C}_1)$ of $\mathcal{C}_1$.

Tree representation $\mathcal{T}(\mathcal{C})$ of $\mathcal{C}$: The *tree representation* $\mathcal{T}(\mathcal{C})$ of a vertex cluster $\mathcal{C}$ is a tree in which the edges are the bridges in $\mathcal{G}(\mathcal{C})$ and the vertices are the 2-edge connected components in $\mathcal{G}(\mathcal{C})$. For a given $\mathcal{G}(\mathcal{C}_1)$ in Figure 2(a), the tree representation $\mathcal{T}(\mathcal{C}_1)$ of $\mathcal{C}_1$ is shown in Figure 3(a), where vertices 1, 2 and 3 are in a 2-edge connected component called a, vertices 4, 5 and 11 are in the 2-edge connected components called b, c and d respectively.

3.2.2 Edge set presentation between two vertex clusters

Given two clusters $\mathcal{C}_i$ and $\mathcal{C}_j$, let $E_{i,j} = \{e_1, e_2, \ldots, e_h\}$ be a subset of E in which the two endpoints of every edge are in $\mathcal{C}_i$ and $\mathcal{C}_j$, respectively. The edges in $E_{i,j}$ are represented by two data structures: one is for $\mathcal{C}_i$, named $E(\mathcal{C}_i, \mathcal{C}_j)$; and the other is for $\mathcal{C}_j$, named $E(\mathcal{C}_j, \mathcal{C}_i)$. Let X be an endpoint set of the edges in $E_{i,j}$ which is a subset of $\mathcal{C}_j$, then $E(\mathcal{C}_i, \mathcal{C}_j) = \{y \mid y \text{ is a 2-edge connected component vertex in } \mathcal{T}(\mathcal{C}_j) \text{ including } x \text{ and } x \in X\}$. Therefore, $E(\mathcal{C}_i, \mathcal{C}_j)$ is a vertex subset of $\mathcal{T}(\mathcal{C}_j)$. In other words, $E(\mathcal{C}_i, \mathcal{C}_j)$ contains those vertices of $\mathcal{T}(\mathcal{C}_j)$ that, for each such vertex u, there is at least an edge $(x, y) \in E_{i,j}$ incident to u, where $x \in \mathcal{C}_i$, $y \in u$ and u is a 2-edge connected component in $\mathcal{G}(\mathcal{C}_j)$. $E(\mathcal{C}_i, \mathcal{C}_j)$ is represented by a balanced binary search tree, and the *key* to access a vertex in it is the preorder numbering of the vertex in $\mathcal{T}(\mathcal{C}_j)$. For the example in Figure 1(b), there are two vertex clusters $\mathcal{C}_1$ and $\mathcal{C}_2$. $E_{2,1} = \{(4, 6), (5, 6), (11, 10)\}$. $E(\mathcal{C}_2, \mathcal{C}_1) = \{b, c, d\}$. Figure 3(b) shows $E(\mathcal{C}_2, \mathcal{C}_1)$, where $preorder(x)$ is the preorder numbering of x in $\mathcal{T}(\mathcal{C}_1)$.

3.2.3 Super graphs and equivalent graphs

The super graph $\hat{\mathcal{G}}$ of G is a multigraph defined as follows. Each vertex in $\hat{\mathcal{G}}$ is a vertex cluster. There is an edge between two vertex clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ if $|E_{i,j}| = 1$, there are two edges between them if $|E_{i,j}| \geq 2$. For example, the graph in Figure 4 is a super graph of a graph G.

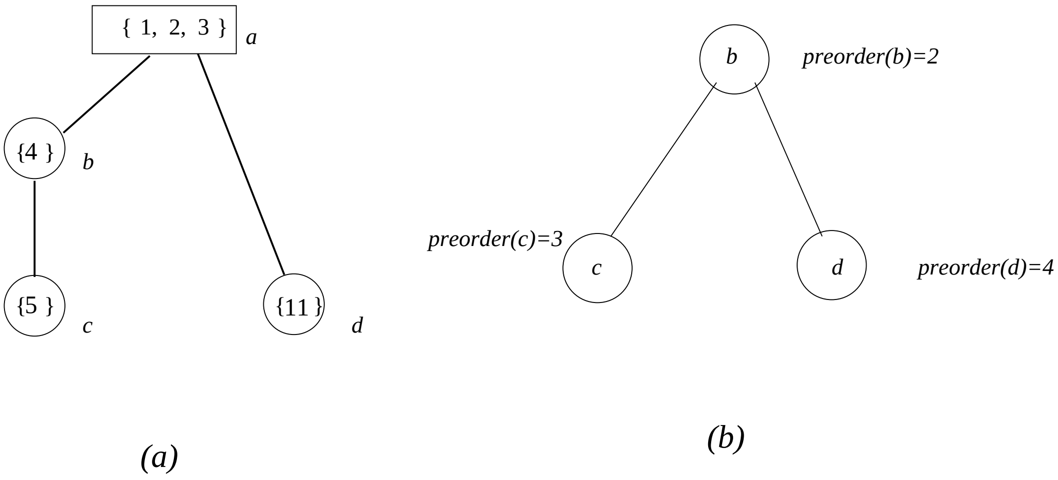


Figure 3: (a) The tree representation $\mathcal{T}(\mathcal{C}_1)$ of $\mathcal{C}_1$; (b) $E(\mathcal{C}_2, \mathcal{C}_1)$ and its binary search tree representation.

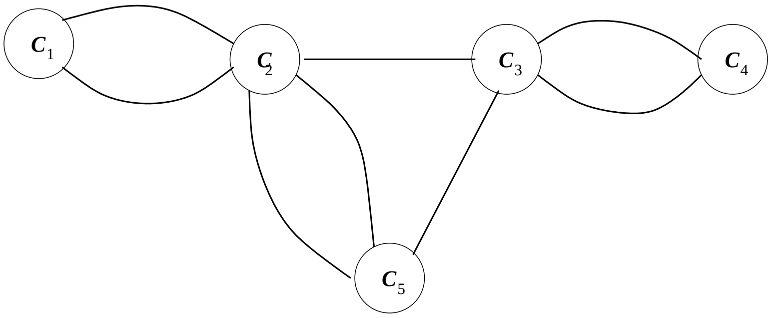
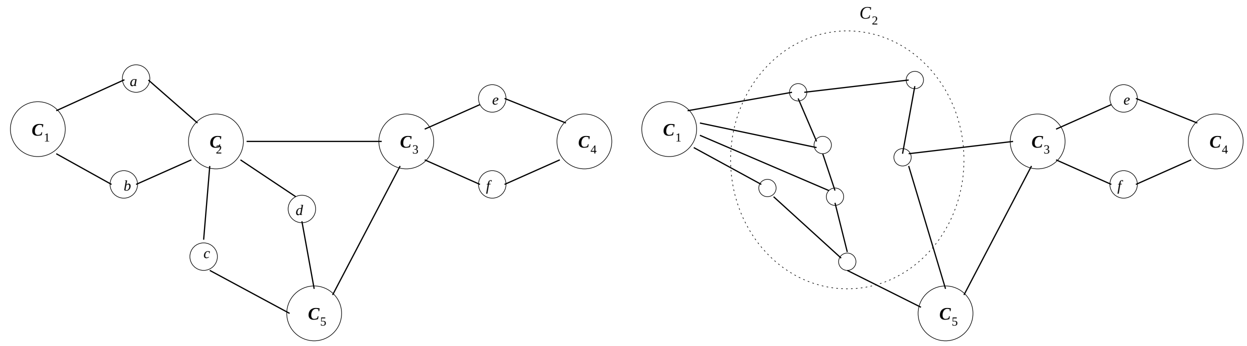


Figure 4: The super graph $\hat{\mathcal{G}}$ of a graph G.

A simple graph $\overline{\mathcal{G}}(\mathcal{V}, \mathcal{E})$ derived from $\hat{\mathcal{G}}$ is defined as follows. $\mathcal{V} = \{\mathcal{C}_i \mid \mathcal{C}_i$ is a vertex cluster$\} \cup \{v_{i,j,1}, v_{i,j,2} \mid$ there are two edges in $\mathcal{G}$ between $\mathcal{C}_i$ and $\mathcal{C}_j\}$, where $v_{i,j,k}$ is an *artificial vertex*, $k = 1, 2$. $\mathcal{E} = \{(\mathcal{C}_i, \mathcal{C}_j) \mid$ there is only one edge in $\hat{\mathcal{G}}$ between $\mathcal{C}_i$ and $\mathcal{C}_j\}$ $\cup\{(\mathcal{C}_i, v_{i,j,k}), (v_{i,j,k}, \mathcal{C}_j) \mid$ there are two edges between $\mathcal{C}_i$ and $\mathcal{C}_j$ in $\hat{\mathcal{G}}$ and $k = 1, 2\}$. Thus, the vertex set of $\hat{\mathcal{G}}$ is a subset of the vertex set of $\overline{\mathcal{G}}$. Graph $\overline{\mathcal{G}}(\mathcal{V}, \mathcal{E})$ is called the *equivalent* graph of $\hat{\mathcal{G}}$ in the sense that they both have the same sets of articulation points and bridges. Assume that $\hat{\mathcal{G}}$ contains m' edges and n' vertices. Then, $\overline{\mathcal{G}}(\mathcal{V}, \mathcal{E})$ contains no more than $2m'$ edges and $n' + m'$ vertices. For example, the graph in Figure 5(a) is the equivalent graph of a super graph shown in Figure 4. From now on, we assume that there is no big difference between $\hat{\mathcal{G}}$ and $\overline{\mathcal{G}}$ with respect to the articulation points and bridges. The purpose that we here introduce the equivalent graph $\overline{\mathcal{G}}$ for $\hat{\mathcal{G}}$ is to make Tarjan et. al's parallel algorithm for finding biconnected components can be applied, which is only applicable for simple graphs.

3.3 Construction of data structures

We now show how to construct the defined data structures in parallel. In order to obtain a vertex cluster partition of G, a spanning tree $T(V, E_T)$ of G is generated, using any efficient parallel algorithm such as the algorithm by



(a) The equivalent graph $\overline{\mathcal{G}}$ of a super graph $\hat{\mathcal{G}}$

(b) Replace $\mathcal{C}_2$ and its artificial adjacent vertices by $\mathcal{F}(\mathcal{C}_2)$ in $\overline{\mathcal{G}}$

Figure 5: The equivalent graph $\overline{\mathcal{G}}$ of $\hat{\mathcal{G}}$ and the resulting graph through replacing a vertex cluster $\mathcal{C}_2$ in $\overline{\mathcal{G}}$ by $\mathcal{F}(\mathcal{C}_2)$.

Awerbuch et. al [2]. Assume T has already been built and stored as an inverted tree. Our approach for the vertex cluster partition of G is based on the following lemma due to Lipton et. al.

Lemma 4 *[37] Given an n-vertex tree with the maximum degree three, there is a vertex partitioning by removing exactly one edge from the tree. As the result, the tree is separated into two subtrees, and the number of vertices in each subtree is between $\lfloor n/3 \rfloor$ and $\lceil 2n/3 \rceil$.*

The vertex cluster partition of G then can be obtained as follows.

Lemma 5 *Given an inverted tree T with m vertices and the maximum degree three, partition its vertices into cm/K vertex clusters such that every vertex cluster contains $O(K)$ vertices. This can be done in $O(\log(m/K))$ time using $O(m)$ processors on an EREW PRAM, where $m \geq 3K/2$, cm/K is the number of clusters, and c is constant.*

Proof Let $\mathcal{F}$ be a forest of inverted trees. Initially $\mathcal{F} = \{T\}$. For each tree $T' \in \mathcal{F}$, we proceed as follows. If the number of vertices n_1 in T' is between $K/2$ and $3K/2$, we do nothing about it; otherwise, traverse T' by a Eulerian tour and make it into a linked list L; delete such an edge in L that the two resulting subtrees have almost the same number of vertices (one has $n_1/3$ vertices and the other has $2n_1/3$ vertices in the worst case). The existence of such a partition is guaranteed by Lemma 4. Remove T' from $\mathcal{F}$ and add the two new trees to $\mathcal{F}$. Repeat the above procedure on $\mathcal{F}$ until the size of every tree in $\mathcal{F}$ is between $K/2$ and $3K/2$. There are $O(\log(m/K))$ calls for the procedure. As a result, every tree in $\mathcal{F}$ is a vertex cluster of G because its size is between $K/2$ and $3K/2$ and the tree is a connected subgraph in G. Therefore, the vertex cluster partition can be done in $O(\log m)$ time using $O(m)$ processors on an EREW PRAM [31], because the size of the resulting tree is reduced by one third at least for each procedure call, compared with the size of its immediately predecessor. $\square$

Note that the restricted vertex cluster partition by Frederickson can not be applied here because it uses the depth-first-search technique which seems highly sequential. Furthermore, although the algorithms for finding a spanning tree and a vertex cluster partition take $O(\log n)$ time using $O(m+n)$ processors and $O(\log(m/K))$ time using $O(m)$ processors, respectively, they are only done once. In the design of dynamic algorithms, the time spent for this part is called *preprocessing time*, which is not taken into account as part of the dynamic maintenance cost for the operations including insertions, deletions and queries.

Given a vertex cluster partition of G, the defined data structures are constructed as follows. For a vertex cluster $\mathcal{C}$, the graph $\mathcal{G}(\mathcal{C})$ can be constructed in $O(1)$ time using $O(K)$ processors on a CRCW PRAM because $\mathcal{G}(\mathcal{C})$ contains $O(K)$ vertices and edges and the degree of each vertex in it is constant. Since each vertex cluster $\mathcal{C}$ contains $O(K)$ vertices and each vertex in G has a degree no greater than three, then, the full representation $\mathcal{F}(\mathcal{C})$ of $\mathcal{C}$ contains $O(K)$ vertices and edges. $\mathcal{F}(\mathcal{C})$ can be constructed in $O(\log K)$ time with $O(K)$ processors, using the adjacency lists of those vertices in G that are also in $\mathcal{C}$ and the vertex cluster partition of G. Note that, the maximum degree of the vertices in $\mathcal{F}(\mathcal{C})$ is $O(K)$, not a constant. The tree representation $\mathcal{T}(\mathcal{C})$ of $\mathcal{C}$ can be obtained by the following lemma.

Lemma 6 *The full tree representation $\mathcal{T}(\mathcal{C})$ of a vertex cluster $\mathcal{C}$ can be constructed in $O(\log K)$ time using $O(K)$ processors on a CRCW PRAM.*

Proof We here provide a constructive proof. We know that $\mathcal{G}(\mathcal{C})$ is a simple graph. First, find all biconnected components in $\mathcal{G}(\mathcal{C})$ by applying Tarjan et. al's algorithm [44]. This takes $O(\log K)$ time with $O(K)$ processors on a CRCW PRAM because $\mathcal{G}(\mathcal{C})$ contains $O(K)$ vertices and edges. Then, find all bridges in $\mathcal{G}(\mathcal{C})$. The edge in a biconnected component consisting of one edge only is a bridge in $\mathcal{G}(\mathcal{C})$, which can be found in $O(1)$ time with $O(K)$ processors after all biconnected components in $\mathcal{G}(\mathcal{C})$ have been found. Let B be the set of bridges in $\mathcal{G}(\mathcal{C})$. Let $\mathcal{G}'(\mathcal{C})$ be the resulting graph after removing the edges in B from $\mathcal{G}(\mathcal{C})$, which can be done in $O(1)$ time with $O(K)$ processors because of the constant degree of each vertex in $\mathcal{G}(\mathcal{C})$. Finding connected components in $\mathcal{G}'(\mathcal{C})$ which in fact are the 2-edge connected components (BCs) in $\mathcal{G}(\mathcal{C})$ takes $O(\log K)$ time using $O(K)$ processors on a CRCW PRAM [43]. Finally, use the vertex in BC with the smallest index to label all the vertices in it, the BC is then represented by that vertex. Thus, all the vertices in the BC have a unique representative. Let $D()$ be such a labeling function, i.e., $D(u) = v$ for all $u \in BC$, where v is the vertex in BC with the smallest index.

Let T'' be a spanning tree of $\mathcal{G}(\mathcal{C})$. we now construct the tree $\mathcal{T}(\mathcal{C})$. Let $F_{\mathcal{T}(\mathcal{C})}(v)$ be the parent of v in $\mathcal{T}(\mathcal{C})$. Then, the vertex set of $\mathcal{T}(\mathcal{C})$ consists of all 2-edge connected components in $\mathcal{G}(\mathcal{C})$. The edges in $\mathcal{T}(\mathcal{C})$ are defined as follows. Initially, set $F_{\mathcal{T}(\mathcal{C})}(D(v)) := v$ for all $v \in BC$. Then, for every edge (x, y) in B, assign $F_{\mathcal{T}(\mathcal{C})}(D(x)) := D(y)$ if y is the parent of x in T''; $F_{\mathcal{T}(\mathcal{C})}(D(y)) := D(x)$ otherwise. This can be done in $O(1)$ time with $O(K)$ processors because B contains $O(K)$ edges. □

Given an inverted tree $\mathcal{T}(\mathcal{C}_i)$ of $\mathcal{C}_i$, assigning every vertex in $\mathcal{T}(\mathcal{C}_i)$ a preorder numbering takes $O(\log K)$ time using $O(K)$ processors on an EREW PRAM [31]. Then, all $E(\mathcal{C}_j, \mathcal{C}_i)$s can be constructed in $O(\log K)$ time using $O(K)$ processors, by sorting the preorder numbering of vertices in increasing order, because the degree of each vertex in $\mathcal{G}(\mathcal{C}_i)$ is no more than three, assuming that $\mathcal{C}_j$ is a vertex cluster adjacent to $\mathcal{C}_i$.

The super graph $\hat{\mathcal{G}}$ can be constructed in $O(1)$ time using $O((m/K)^2)$ processors because $\hat{\mathcal{G}}$ contains $O(m/K)$ vertices. Thus, the equivalent graph $\overline{\mathcal{G}}$ of $\hat{\mathcal{G}}$ can be constructed in the same time and processor bounds because $\overline{\mathcal{G}}$ contains $O((m/K)^2)$ vertices and edges.

Consequently, the data structures pertinent to a vertex cluster $\mathcal{C}$ can be maintained in $O(\log K)$ time using $O(K)$ processors; and the data structures for $\hat{\mathcal{G}}$ and $\overline{\mathcal{G}}$ can be maintained in $O(\log(m/K) + \log K)$ time using $O((m/K)^2)$ processors on a CRCW PRAM, which will be proven in Section 3.4.

3.4 The algorithm for queries

Given a pair of vertices x and y, the query is about whether x and y are in the same 2-edge connected component in G. Instead of using G, an auxiliary graph $\mathcal{G}_{x,y}$ will be used to answer the query. Let $\mathcal{C}_x$ and $\mathcal{C}_y$ be the vertex clusters containing x and y respectively. The simple graph $\mathcal{G}_{x,y}$ is obtained from $\overline{\mathcal{G}}$ through replacing vertices $\mathcal{C}_x$, $\mathcal{C}_y$ and their adjacent artificial vertices, and the edges incident to them by the full representations $\mathcal{F}(\mathcal{C}_x)$ and $\mathcal{F}(\mathcal{C}_y)$ of $\mathcal{C}_x$ and $\mathcal{C}_y$. For example, Figure 5(b) shows the resulting graph after replacing $\mathcal{C}_2$ and its adjacent artificial vertices in $\overline{\mathcal{G}}$ in Figure 5(a) by its full representation $\mathcal{F}(\mathcal{C}_2)$. Now, assume that x and y are not in the same 2-edge connected component in G. Then, there must be at least one bridge $e \in E$ in G separating x from y and any such bridge can be one of the following two types: (1) type 1 if e is either internal to $\mathcal{C}_x$, or $\mathcal{C}_y$, or an edge between two different clusters; (2) type 2 if e is internal to a vertex cluster $\mathcal{C}$ with $\mathcal{C} \neq \mathcal{C}_x$ and $\mathcal{C} \neq \mathcal{C}_y$. To check these two types of bridges, we have the following theorem.

Theorem 2 *[19] Let x and y be any two vertices in G. Then the following is true: (i) e is a type 1 bridge in G if and only if the corresponding edge is a bridge in $\mathcal{G}_{x,y}$; (ii) if e is a type 2 bridge in G separating x from y, and e is internal to a cluster $\mathcal{C}$ ($\mathcal{C} \neq \mathcal{C}_x$ and $\mathcal{C} \neq \mathcal{C}_y$), then $\mathcal{C}$ is an articulation point in $\mathcal{G}_{x,y}$.*

Following Theorem 2, the checking algorithm proceeds as follows. It checks type 1 bridges in $\mathcal{G}_{x,y}$ first. If there is no type 1 bridge, then it checks type 2 bridges. If no type 2 bridge exists, then, x and y are in the same 2-edge connected component in G; otherwise, x and y are in the different 2-edge connected components in G. The rest of this section, therefore, focuses on checking the two types of bridges. However, type 1 bridge can be easily detected by the following lemma.

Lemma 7 *Given a simple connected graph with n_1 vertices and m_1 edges, to determine whether there exist bridges in any path between two given vertices takes $O(\log n_1)$ time using $O(n_1 + m_1)$ processors on a CRCW PRAM.*

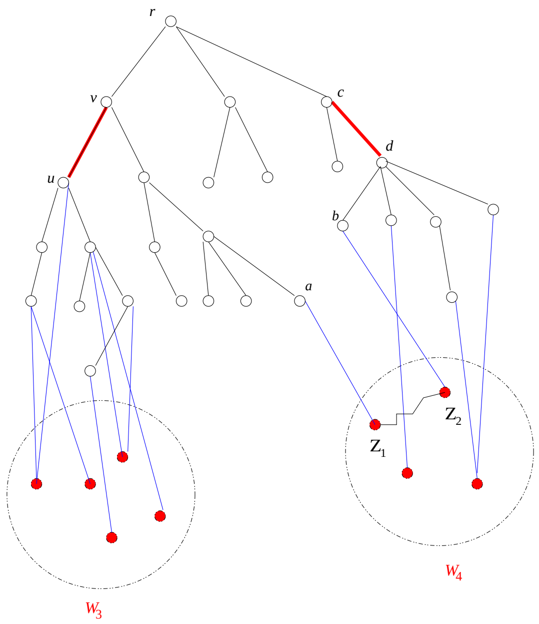
Proof First, find all biconnected components in the graph by applying Tarjan et. al's algorithm, and identify those biconnected components consisting of one edge only (the edges are the bridges of the graph). Let B be the set of such edges. Then, find a spanning tree T in the graph by applying any efficient spanning tree algorithm, and identify the edges in path π in T between the two given vertices. Finally, if there is an edge in both B and π, then it is a bridge of the graph separating the two vertices. The time spent for finding biconnected components is $O(\log n_1)$ and the number of processors used is $O(m_1 + n_1)$. Constructing B takes $O(1)$ time using $O(n_1)$ processors. T can be found in $O(\log n_1)$ time using $O(m_1 + n_1)$ processors. All the edges in π can be identified in $O(\log n_1)$ time using $O(n_1)$ processors. Therefore, all the bridges in the graph separating x from y can be found in $O(\log n_1)$ time using $O(m_1 + n_1)$ processors on a CRCW PRAM. □

As results, type 1 bridges in G between x and y can be found in $O(\log K + \log(m/K))$ time using $O(K + (m/K)^2)$ processors on a CRCW PRAM, because $\mathcal{G}_{x,y}$ contains $O(K + (m/K)^2)$ vertices and edges by Lemma 7.

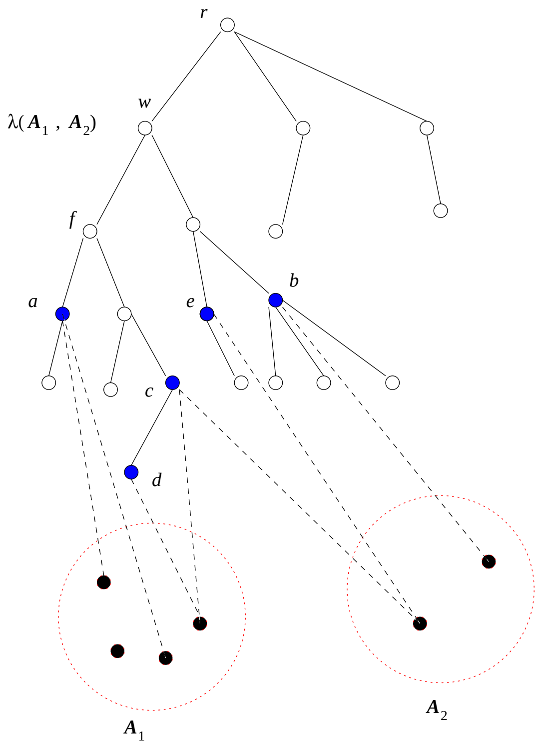
3.4.1 Type 2 bridge checking

We now deal with type 2 bridges. We first introduce the following notions and notations for later use.

Let a vertex cluster $\mathcal{C}$ be an articulation point in $\mathcal{G}_{x,y}$ separating x and y. Then, $\mathcal{C}$ is also an articulation point in both $\overline{\mathcal{G}}$ and $\hat{\mathcal{G}}$ separating $\mathcal{C}_x$ from $\mathcal{C}_y$, by the definition of equivalent graphs. Let $AC = \{\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_t\}$ be a set of vertex clusters in which each vertex cluster $\mathcal{Q}_j$ in $\hat{\mathcal{G}}$ is adjacent to $\mathcal{C}$, $1 \leq j \leq t$ and $t \geq 2$. Let $\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_s$ be the s connected components in the resulting graph $\hat{\mathcal{G}} - \{\mathcal{C}\}$ after the removal of $\mathcal{C}$ and the edges incident to it from $\hat{\mathcal{G}}$, where each vertex in $\mathcal{W}_i$ represents a vertex cluster, $1 \leq i \leq s$. Without loss of generality, let $\mathcal{W}_1 = \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_p\}$ be the connected component containing $\mathcal{C}_x$, $\mathcal{W}_2 = \{\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_q\}$ be the connected component containing $\mathcal{C}_y$, and $\mathcal{W}_i = \{\mathcal{Z}_{i,1}, \mathcal{Z}_{i,2}, \ldots, \mathcal{Z}_{i,r(i)}\}$ be a connected component containing the vertex clusters in $AC - (\mathcal{W}_1 \cup \mathcal{W}_2)$, $3 \leq i \leq s$, where $\mathcal{X}_{l_1}$, $\mathcal{Y}_{l_2}$, and $\mathcal{Z}_{i,j}$ are the vertex clusters, $1 \leq l_1 \leq p$, $1 \leq l_2 \leq q$, $1 \leq j \leq r(i)$, and $3 \leq i \leq s$. Clearly, $p + q + \sum_{i=3}^{s} r(i) = t$. Given the tree representation $\mathcal{T}(\mathcal{C})$ of $\mathcal{C}$ and an edge e in it, let $\mathcal{T}_1(\mathcal{C})$ and $\mathcal{T}_2(\mathcal{C})$ be the resulting subtrees after the removal of e from $\mathcal{T}(\mathcal{C})$. The connected component $\mathcal{W}_i$ ($= \{\mathcal{Z}_{i,1}, \mathcal{Z}_{i,2}, \ldots, \mathcal{Z}_{i,r(i)}\}$) defined is said to be *compatible with* e if all the edges between $\mathcal{Z}_{i,j}$ and $\mathcal{C}$, $1 \leq j \leq r(i)$, are incident to either $\mathcal{T}_1(\mathcal{C})$ or $\mathcal{T}_2(\mathcal{C})$, but not to both, $3 \leq i \leq s$. For example, the connected component $\mathcal{W}_3$ shown in Figure 6(a) is compatible with edge (u, v), but the connected component $\mathcal{W}_4$ is incompatible with edge (c, d) which is explained as follows. There are two edges (v_1, v_a) and (v_2, v_b) in G such that both $v_1 \in Z_1$ and $v_2 \in Z_2$, while Z_i is a vertex cluster in $\mathcal{W}_4$, $i = 1, 2$. Let a and b be the 2-connected components in $\mathcal{G}(\mathcal{C})$ containing another endpoints v_a and v_b of the two edges, and $\mathcal{T}_1(\mathcal{C})$ and $\mathcal{T}_2(\mathcal{C})$ be the resulting trees including c and d after the removal of (c, d) from $\mathcal{T}(\mathcal{C})$. We then have $a \in \mathcal{T}_1(\mathcal{C})$ but $b \in \mathcal{T}_2(\mathcal{C})$, while both $v_1 \in Z_1$ in $\mathcal{W}_4$ and $v_2 \in Z_2$ in $\mathcal{W}_4$. Following the definition, $\mathcal{W}_4$ is incompatible with edge (c, d).



(a) $\mathcal{W}_3$ is compatible with (u, v) while $\mathcal{W}_4$ is incompatible with (c, d) in $\mathcal{T}(\mathcal{C})$



(b) An illustration of $\lambda(\mathcal{A}_1, \mathcal{A}_2)$

Figure 6: The notions on $\mathcal{T}(\mathcal{C})$

Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_l$ be any l clusters in $\hat{\mathcal{G}}$ adjacent to cluster $\mathcal{C}$, $l > 0$. Then, $E(\mathcal{A}_i, \mathcal{C}) \neq \emptyset$ for all i, and color each vertex in $E(\mathcal{A}_i, \mathcal{C})$ with color *blue*, $1 \leq i \leq l$. Denote by $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_l)$ a vertex in $\mathcal{T}(\mathcal{C})$ which is the root of a subtree containing all the blue vertices but no other subtree rooted at a proper descendant of $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_l)$ also contains all the blue vertices. In other words, $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_l)$ is such a vertex in $\mathcal{T}(\mathcal{C})$ that all the edges between $\mathcal{A}_i$ and $\mathcal{C}$ are incident *below* it, $1 \leq i \leq l$. Here "below" means that a vertex is included in a subtree of $\mathcal{T}(\mathcal{C})$ rooted at $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_l)$. For example, given two clusters $\mathcal{A}_1$ and $\mathcal{A}_2$ adjacent to $\mathcal{C}$, shown in Figure 6(b), vertex w is $\lambda(\mathcal{A}_1, \mathcal{A}_2)$, but vertex f is not, because the subtree rooted at f does not include the blue vertices b and e in $E(\mathcal{A}_2, \mathcal{C})$, where a, b, c, d, and e are blue vertices in $\mathcal{T}(\mathcal{C})$.

Recall that $\mathcal{C}_x$ and $\mathcal{C}_y$ are the clusters including vertices x and y which are adjacent to $\mathcal{C}$, $\mathcal{X}_i \in \mathcal{W}_1$, and $\mathcal{Y}_j \in \mathcal{W}_2$, $1 \leq i \leq p$ and $1 \leq j \leq q$. A vertex v in $\mathcal{T}(\mathcal{C})$ is colored *red* if there is an edge between $\mathcal{X}_i$ and $\mathcal{C}$ incident to v, $1 \leq i \leq p$. Similarly, a vertex u in $\mathcal{T}(\mathcal{C})$ is colored *black* if there is an edge between $\mathcal{Y}_j$ and $\mathcal{C}$ incident to u, $1 \leq j \leq q$. Define the *top-red* vertex ρ as $\lambda(\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_p)$ and the *top-black* vertex β as $\lambda(\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_q)$. We now check type 2 bridges. Let $\mathcal{C}$ be an articulation point in $\mathcal{G}_{x,y}$ separating x from y. Then a type 2 bridge internal to $\mathcal{C}$ separating x from y in G may exist. Our objective is to find the bridge if it exists. The intuition behind the algorithm is as follows.

Assume that there is an edge $e = (u, v)$ in $\mathcal{T}(\mathcal{C})$ and the corresponding edge of e in G is a bridge separating x from y. Let $\mathcal{T}_1(\mathcal{C})$ and $\mathcal{T}_2(\mathcal{C})$ be the two subtrees including u and v after the removal of e from $\mathcal{T}(\mathcal{C})$. Following the definition of $\mathcal{C}_x$ and $\mathcal{C}_y$, every path in G between x and y must have the following structure. It starts with a path outside of $\mathcal{C}$ ending at a vertex of $\mathcal{C}$ that corresponds to a red vertex in $\mathcal{T}(\mathcal{C})$. It ends with a path outside $\mathcal{C}$ starting at a vertex of $\mathcal{C}$ that corresponds to a black vertex in $\mathcal{T}(\mathcal{C})$. Since the corresponding edge of e internal to $\mathcal{C}$ is the bridge of G separating x from y, then there is a path in G from x to y containing the corresponding edge of e. Thus, e satisfies the following two conditions: (i) e must be in a path in $\mathcal{T}(\mathcal{C})$ between a red vertex and a black vertex; and (ii) each $\mathcal{W}_i$ is *compatible* with e, $3 \leq i \leq s$. In other words, for a given $\mathcal{W}_i$, all the edges between $\mathcal{W}_i$ and $\mathcal{C}$ are incident to either $\mathcal{T}_1(\mathcal{C})$ or $\mathcal{T}_2(\mathcal{C})$, but not to both, $3 \leq i \leq s$. The query algorithm thus is to check whether there is such an edge e in $\mathcal{T}(\mathcal{C})$ in order to determine whether there is a bridge in G separating x from y. We have the following lemma.

Lemma 8 *[19] Let $\mathcal{C}$ be a vertex cluster that is an articulation point in $\mathcal{G}_{x,y}$ separating x from y. There is a bridge inside $\mathcal{C}$ separating x from y in G if and only if there is an edge e in $\mathcal{T}(\mathcal{C})$ that satisfies the following two conditions: (i) the removal of e separates red vertices from black vertices; and (ii) each connected component $\mathcal{W}_i$ is compatible with e, $3 \leq i \leq s$.*

Given ρ and β in $\mathcal{T}(\mathcal{C})$. Let $v = LCA(\beta, \rho)$ be the lowest common ancestor of ρ and β in $\mathcal{T}(\mathcal{C})$. Condition (i) of Lemma 8 can be further decomposed into four cases which are expressed in the following lemma, depending on the relative positions of the three vertices ρ, β and v in $\mathcal{T}(\mathcal{C})$.

Lemma 9 *[19] Let ρ be the top-red, β the top-black vertices, and $v = LCA(\beta, \rho)$. Let $\pi_{\rho,\beta}$ be the path in $\mathcal{T}(\mathcal{C})$ between ρ and β. Then, (1) if $v = \rho = \beta$, no edge in $\mathcal{T}(\mathcal{C})$ can separate black and red vertices; (2) if $v \neq \rho$ and $v \neq \beta$, then all the edges in $\pi_{\rho,\beta}$ separate black and red vertices; (3) if $v = \beta \neq \rho$, an edge e separates black and red vertices if and only if e is in $\pi_{\rho,\beta}$ and there are no black vertices below $child(e)$, where $child(e)$ is an endpoint of e and another endpoint of e is the parent of $child(e)$ in $\mathcal{T}(\mathcal{C})$; (4) if $v = \rho \neq \beta$, an edge e separates black from red vertices if and only if e is in $\pi_{\rho,\beta}$ and there are no red vertices below $child(e)$.*

Lemmas 8 and 9 imply that we only need to check whether there is an edge e in $\pi_{\rho,\beta}$ of $\mathcal{T}(\mathcal{C})$ separating red vertices from black vertices and every $\mathcal{W}_i$ is compatible with e, $3 \leq i \leq s$, in order to check whether there is a bridge inside $\mathcal{C}$ separating x from y in G. The algorithm proceeds as follows. It first considers all the edges between $\mathcal{C}$ and the clusters in $\mathcal{W}_1$ and $\mathcal{W}_2$. If there is no edge e in $\pi_{\rho,\beta}$ separating red vertices from black vertices in $\mathcal{T}(\mathcal{C})$

(Case (1)), then there is no bridge inside $\mathcal{C}$ separating x from y in G, the query is answered. Otherwise, it examines Cases (2)-(4) of Lemma 9. We here only deal with Case (3). Case (4) is analogous, while Case (2) can be dealt by checking the edges in two paths $\pi_{\rho,v}$ and $\pi_{\beta,v}$ in $\mathcal{T}(\mathcal{C})$ respectively, using the very similar fashion. However, the algorithm for Case (3) due to Galil et. al. is highly sequential, and therefore hard to parallelize. A parallel algorithm for Case (3) is proposed below.

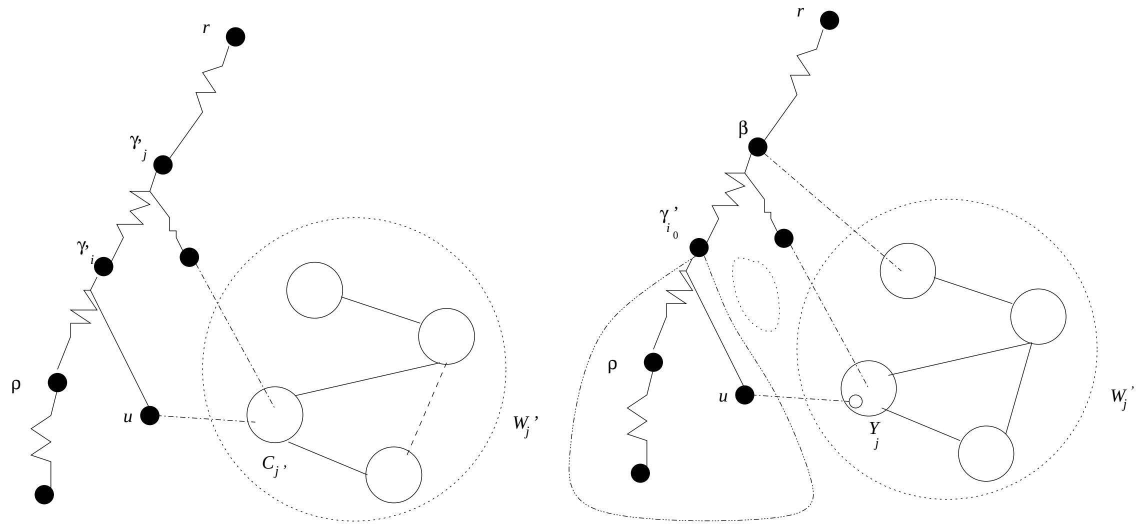
Assume that r is the root of $\mathcal{T}(\mathcal{C})$ and each vertex v in $\mathcal{T}(\mathcal{C})$ has a preorder numbering $preorder(v)$. Consider the connected components $\mathcal{W}_3$, $\mathcal{W}_4$, ..., $\mathcal{W}_s$ defined. Let $\gamma_i = \lambda(\mathcal{Z}_{i,1}, \mathcal{Z}_{i,2}, \ldots, \mathcal{Z}_{i,r(i)})$, where cluster $\mathcal{Z}_{i,j} \in \mathcal{W}_i$. If some γ_i is not in path $\pi_{\rho,r}$ of $\mathcal{T}(\mathcal{C})$ between ρ and r, then all the edges between $\mathcal{Z}_{i,j}$ and $\mathcal{C}$ are incident to one of the two subtrees $\mathcal{T}_1(\mathcal{C})$ and $\mathcal{T}_2(\mathcal{C})$ after the removal of any edge e in $\pi_{\rho,\beta}$ from $\mathcal{T}(\mathcal{C})$ for all j, $1 \leq j \leq r(i)$. This means that $\mathcal{W}_i$ is always compatible with e in $\mathcal{T}(\mathcal{C})$ for any edge e in $\pi_{\rho,\beta}$, and all the edges between $\mathcal{Z}_{i,j}$ in $\mathcal{W}_i$ and $\mathcal{C}$ do not help determine whether e separates red vertices from black vertices in $\mathcal{T}(\mathcal{C})$. So, discard all such γ_is that are not in path $\pi_{\rho,r}$ for further consideration, $3 \leq i \leq s$. Sort the remaining γ_is in $\mathcal{T}(\mathcal{C})$ by the preorder numbering in decreasing order, and let $\gamma'_0, \gamma'_1, \gamma'_2, \ldots, \gamma'_{s'}$ be the sorted vertex sequence and $\mathcal{W}'_0$, $\mathcal{W}'_1$, ..., $\mathcal{W}'_{s'}$ be the corresponding connected components, then $preorder(\gamma'_l) > preorder(\gamma'_{l+1})$, $0 \leq l < s'$, $0 \leq s' \leq s$, and $\gamma'_0 = \rho$. Label each vertex cluster in $\mathcal{W}'_i$ with γ'_i. Thus, each cluster $\mathcal{C}_{i'} \in \mathcal{W}'_i$ has a label γ'_i, $0 \leq i \leq s'$.

The checking of Case (3) is implemented through constructing two auxiliary graphs G_i first, $i = 1, 2$, and then using the graphs to check whether there is an edge in $\pi_{\rho,\beta}$ in $\mathcal{T}(\mathcal{C})$ satisfying the two conditions in Lemma 8. The objective of constructing G_1 and G_2 is to find a maximal subpath $\pi_{\rho,\gamma'_{i_0}}$ of $\pi_{\rho,\beta}$ including ρ such that no edge in it can satisfy the two conditions in Lemma 8.

The auxiliary graph $G_1(V_1, E_1)$ is constructed as follows. $V_1 = \{\gamma'_0, \gamma'_1, \ldots, \gamma'_{s'}\}$. There is an edge $(\gamma'_i, \gamma'_j) \in E_1$ if and only if (i) there is a cluster $\mathcal{C}_{i'} \in \mathcal{W}'_i$ and a cluster $\mathcal{C}_{j'} \in \mathcal{W}'_j$ and $E_{i',j'} \neq \emptyset$; or (ii) there is a cluster $\mathcal{C}_{j'} \in \mathcal{W}'_j$ and there is an edge between $\mathcal{C}_{j'}$ and $\mathcal{C}$ incident below γ'_i in $\mathcal{T}(\mathcal{C})$. Figure 7(a) illustrates condition (ii) in the construction of G_1, where u represents a 2-edge connected component in $\mathcal{G}(\mathcal{C})$. There is an edge between $\mathcal{C}_{j'}$ and $\mathcal{C}$ incident to u which is below γ'_i, and $u \in \mathcal{C}$. It is obvious that for a given pair of vertices γ'_i and γ'_j in G_1, if either condition (i) or condition (ii) holds, then there is an edge in G_1 and a corresponding path $\pi_{\gamma'_i,\gamma'_j}$ in $\mathcal{T}(\mathcal{C})$ in which no edge can satisfy the two conditions in Lemma 8, because there are some $\mathcal{W}'_l$s which are incompatible with e for every edge e in $\pi_{\gamma'_i,\gamma'_j}$.

Let CC be a connected component in $G_1(V_1, E_1)$ in which γ'_l and γ'_h are the two vertices with the smallest and the largest indexes l and h. In case CC contains only one vertex, then $l = h$. Clearly $0 \leq l, h \leq s'$. We assign every vertex $v \in CC$ a pair of labels, l and h. Let $[l, h]$ represent a closed interval with $l \leq h$. Denote by $[l_i, h_i]$ the pair of labels assigned to γ'_i. Then, we have the following lemma.

Lemma 10 *Let CC be a connected component in $G_1(V_1, E_1)$ and every vertex in it is labelled with a pair of labels*



(a) The condition (ii) in the construction of G_1

(b) An illustration of checking whether an edge in $\pi_{\gamma'_{i_0},\beta}$ satisfies the two conditions of Lemma 8.

Figure 7: Checking whether an edge in $\pi_{\rho,\beta}$ satisfying the two conditions of Lemma 8.

l_i and h_i as above. Then there are $\mathcal{W}'_j$s which are incompatible with e for every edge e in path $\pi_{\gamma'_{l_i},\gamma'_{h_i}}$ in $\mathcal{T}(\mathcal{C})$, $1 \le j \le s'$.

Proof Let CC be a connected component in G_1 in which every vertex has been labelled with l_i and h_i. Then, the vertices γ'_{l_i} and γ'_{h_i} are included in CC by the definition. Following the above discussion, we know that there is no edge in $\pi_{\gamma'_a,\gamma'_b}$ in $\mathcal{T}(\mathcal{C})$ satisfying the two conditions in Lemma 8 for every edge $(\gamma'_a,\gamma'_b) \in E_1$ in G_1, by the construction of G_1 and $l_i \le a, b \le h_i$. Thus, every edge in G_1 corresponds to a subpath of $\pi_{\rho,r}$ in $\mathcal{T}(\mathcal{C})$ in which no edge satisfies the two conditions in Lemma 8. Now, consider two edges (γ'_a,γ'_b) and (γ'_b,γ'_c) sharing a common vertex γ'_b in a CC, then no edge in $\pi_{\gamma'_a,\gamma'_c}$ satisfies the two conditions in Lemma 8, because both $\pi_{\gamma'_a,\gamma'_b}$ and $\pi_{\gamma'_b,\gamma'_c}$ do not contain any edge satisfying the two conditions. Therefore, no edge in path $\pi_{\gamma'_{l_i},\gamma'_{h_i}}$ in $\mathcal{T}(\mathcal{C})$ satisfies the two conditions in Lemma 8, because $\pi_{\gamma'_{l_i},\gamma'_{h_i}}$ is the union of the subpaths generated by the corresponding edges of G_1 in the CC. $\square$

Given that each vertex $\gamma'_i \in V_1$ has a pair of labels l_i and h_i, $0 \le i < s'$, another graph $G_2(V_2, E_2)$ is constructed as follows: $V_2 = V_1$ and $(\gamma'_i,\gamma'_j) \in E_2$ if and only if (i) either $(\gamma'_i,\gamma'_j) \in E_1$; or (ii) the intersection of the two intervals is non-empty, i.e., $[l_i, h_i] \cap [l_j, h_j] \ne \emptyset$, which means that two paths $\pi_{\gamma'_{l_i},\gamma'_{h_i}}$ and $\pi_{\gamma'_{l_j},\gamma'_{h_j}}$ are overlapping. Since no edge in both $\pi_{\gamma'_{l_i},\gamma'_{h_i}}$ and $\pi_{\gamma'_{l_j},\gamma'_{h_j}}$ satisfies the two conditions of Lemma 8, no edge in the union of the two paths satisfies the two conditions in Lemma 8. We then have the following lemma.

Lemma 11 *Let CC be a connected component in $G_2(V_2, E_2)$ including vertex $\rho = \gamma'_0$ such that i_0 is the maximum* *index of the vertices in it. Then, there is no edge e in $\pi_{\rho,\gamma'_{i_0}}$ in $\mathcal{T}(\mathcal{C})$ satisfying (i) the removal of e separates red vertices from black vertices; and (ii) each connected component $\mathcal{W}'_i$ is compatible with e, $1 \leq i \leq s'$.*

Proof Let CC' be a connected component in G_1 consisting of vertices $\gamma'_{i_1}, \gamma'_{i_2}, \ldots, \gamma'_{i_k}$. Assume that these vertices are sorted by their preorder numberings in $\mathcal{T}(\mathcal{C})$ in decreasing order, i.e., $preorder(\gamma'_{i_j}) > preorder(\gamma'_{i_{j+1}})$ and $1 \leq j \leq k-1$. Then, there is no edge in path $\pi_{\gamma'_{i_1},\gamma'_{i_k}}$ in $\mathcal{T}(\mathcal{C})$ satisfying the two conditions in Lemma 8, which is derived by Lemma 10 directly.

Let CC_i and CC_j be the two connected components in G_1 and $[l_i, h_i]$ and $[l_j, h_j]$ the pairs of labels for them. If $[l_i, h_i] \cap [l_j, h_j] \neq \emptyset$, we can distinguish four cases. (i) $l_i > l_j$ but $h_i < h_j$, i.e., $[l_i, h_i] \subset [l_j, h_j]$. Path $\pi_{\gamma'_{l_i},\gamma'_{h_i}}$ is a subpath in $\mathcal{T}(\mathcal{C})$ of $\pi_{\gamma'_{l_j},\gamma'_{h_j}}$, while no edge in $\pi_{\gamma'_{l_j},\gamma'_{h_j}}$ in $\mathcal{T}(\mathcal{C})$ satisfies the two conditions in Lemma 8. (ii) $l_i < l_j$ and $h_i < h_j$. There is no edge in $\pi_{\gamma'_{l_i},\gamma'_{h_j}}$ in $\mathcal{T}(\mathcal{C})$ satisfying the two conditions in Lemma 8, because both $\pi_{\gamma'_{l_i},\gamma'_{h_i}}$ and $\pi_{\gamma'_{l_j},\gamma'_{h_j}}$ are the subpaths of $\pi_{\gamma'_{l_i},\gamma'_{h_j}}$. (iii) $l_i < l_j$ but $h_i > h_j$. This case is similar to case (i), omitted. (iv) $l_i > l_j$ and $h_i > h_j$. This case is similar to case (ii), and omitted. Thus, each edge (γ'_a, γ'_b) in a connected component including ρ corresponds to a path $\pi_{\gamma'_a,\gamma'_b}$ in $\mathcal{T}(\mathcal{C})$. So, we conclude that there is no edge in $\pi_{\rho,\gamma'_{i_0}}$ in $\mathcal{T}(\mathcal{C})$ satisfying the two conditions in Lemma 8, because$\pi_{\rho,\gamma'_{i_0}}$ is the union of the subpaths derived from the corresponding edges in the connected component including ρ in G_2. □

The example shown in Figure 8 illustrates the idea used in the proof of Lemma 11. Assume that $\gamma'_{j_1}, \gamma'_{j_2}, \gamma'_{j_3}$, and γ'_{j_4} are the vertices in a connected component in G_1, $\gamma'_{i_1}, \gamma'_{i_2}, \gamma'_{i_3}$, and γ'_{i_4} are the vertices in another connected component in G_1. We have shown that there is no edge in either $\pi_{\gamma'_{j_1},\gamma'_{j_4}}$ or $\pi_{\gamma'_{i_1},\gamma'_{i_4}}$ of $\mathcal{T}(\mathcal{C})$ satisfying the two conditions of Lemma 8, by Lemma 10. Suppose that $[j_1, j_4] \cap [i_1, i_4] \neq \emptyset$, then, the vertices γ'_{j_1}, γ'_{j_2}, γ'_{j_3}, γ'_{j_4}, γ'_{i_1}, γ'_{i_2}, γ'_{i_3}, and γ'_{i_4} are in the same connected component in G_2, by the definition of G_2. Clearly, there is no edge in path $\pi_{\gamma'_{j_1},\gamma'_{i_4}}$ in $\mathcal{T}(\mathcal{C})$ satisfying the two conditions in Lemma 8 due to that $\pi_{\gamma'_{j_1},\gamma'_{i_4}}$ is the union of $\pi_{\gamma'_{j_1},\gamma'_{j_4}}$ and $\pi_{\gamma'_{i_1},\gamma'_{i_4}}$.

Having built G_1 and G_2, to decide whether there is an edge in $\pi_{\rho,\beta}$ in $\mathcal{T}(\mathcal{C})$ satisfying the two conditions in Lemma 8 (Case (3)), it proceeds as follows. If $preorder(\beta) > preorder(\gamma'_{i_0})$, then there is no edge in $\pi_{\rho,\beta}$ satisfying the two conditions in Lemma 8, because $\pi_{\rho,\beta}$ is a subpath of $\pi_{\rho,\gamma'_{i_0}}$ and there is no edge in $\pi_{\rho,\gamma'_{i_0}}$ satisfying the two conditions in Lemma 8 by Lemma 11, where vertices γ'_{i_0} and ρ are in the same connected component in G_2 and γ'_{i_0} has the maximum index i_0 in the connected component. Otherwise, although there is no edge in path $\pi_{\rho,\gamma'_{i_0}}$ satisfying the two conditions of Lemma 8, we cannot guarantee that this claim also holds in path $\pi_{\gamma'_{i_0},\beta}$. Path $\pi_{\rho,\beta}$ is the union of $\pi_{\rho,\gamma'_{i_0}}$ and $\pi_{\gamma'_{i_0},\beta}$. To check whether there is an edge in path $\pi_{\gamma'_{i_0},\beta}$ satisfying the two conditions of Lemma 8, we proceed by checking whether there is a vertex cluster $\mathcal{Y}_j \in \mathcal{W}_2$ including $\mathcal{C}_y$ such that $u \in E(\mathcal{Y}_j, \mathcal{C})$ and

$$preorder(\gamma'_{i_0}) \quad \leq \quad preorder(u) < preorder(\gamma'_{i_0}) + nd(\gamma'_{i_0}) - 1, \tag{1}$$

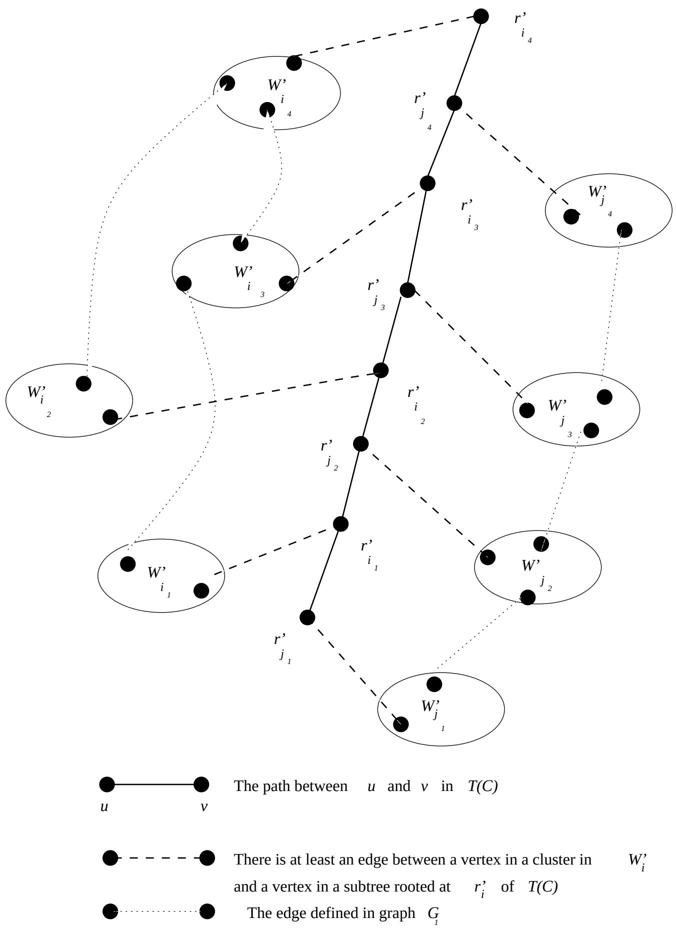


Figure 8: An illustration of the proof of Lemma 11.

where $nd(z)$ is the number of vertices in a subtree of $\mathcal{T}(\mathcal{C})$ rooted at z. In other words, we need to check whether there is an edge between $\mathcal{Y}_j$ and $\mathcal{C}$ incident to u, and u is below γ'_{i_0} in $\mathcal{T}(\mathcal{C})$, $1 \leq j \leq q$. Figure 7(b) illustrates this case. If inequality (1) holds, then, no edge in $\pi_{\gamma'_{i_0},\beta}$ satisfies the two conditions of Lemma 8, which means there is no bridge internal to $\mathcal{C}$ separating x from y in G. We then examine the other articulation points $\mathcal{C}'$ if existed in $\mathcal{G}_{x,y}$ separating x from y to determine whether x and y are in the same 2-edge connected component in G. If none of the articulation points contains a bridge separating x from y in G, then x and y are in the same 2-edge connected component in G.

3.4.2 Implementing the operations

Now we discuss the implementation details of the proposed algorithm for query processing. Due to space limitations, some unimportant details are omitted. Assume that the data structures defined are available.

$\mathcal{G}_{x,y}$ is constructed as follows. First, a graph $\mathcal{G}'$ is generated from $\overline{\mathcal{G}}$ by removing the vertices $\mathcal{C}_x$ and $\mathcal{C}_y$ and their adjacent artificial vertices (i.e., the new vertices added in the generation of $\overline{\mathcal{G}}$). Then, $\mathcal{G}_{x,y}$ is generated by merging the adjacency lists of $\mathcal{G}'$, $\mathcal{F}(\mathcal{C}_x)$, and $\mathcal{F}(\mathcal{C}_y)$, which takes $O(\log K + \log(m/K))$ time using $O(K + (m/K)^2)$ processors on a CRCW PRAM, because $\mathcal{G}_{x,y}$ contains $O(K + (m/K)^2)$ vertices and edges.

Given the graph $\mathcal{G}_{x,y}$, the next objective is to find all articulation points $\mathcal{C}$s in $\mathcal{G}_{x,y}$ separating x from y. Obviously, each such $\mathcal{C}$ is also an articulation point in both $\overline{\mathcal{G}}$ and $\hat{\mathcal{G}}$ separating $\mathcal{C}_x$ from $\mathcal{C}_y$, which can be found by the following lemma.

Lemma 12 *Finding all articulation points in $\overline{\mathcal{G}}$ separating $\mathcal{C}_x$ from $\mathcal{C}_y$ takes $O(\log(m/K) + \log K)$ time using $O((m/K)^3)$ processors on a CRCW PRAM.*

Proof Since $\hat{\mathcal{G}}$ and $\overline{\mathcal{G}}$ are the equivalent graphs, an articulation point in $\hat{\mathcal{G}}$ is also an articulation point in $\overline{\mathcal{G}}$. We proceed the following in parallel. For each vertex $\mathcal{C}$ in $\hat{\mathcal{G}}$, make a copy $\mathcal{G}_\mathcal{C}$ of $\overline{\mathcal{G}}$ for $\mathcal{C}$, delete $\mathcal{C}$ and its incident edges from $\mathcal{G}_\mathcal{C}$, compute the connected components of the resulting graph $\mathcal{G}_\mathcal{C} - \{\mathcal{C}\}$. If vertices $\mathcal{C}_x$ and $\mathcal{C}_y$ are not in the same connected component in $\mathcal{G}_\mathcal{C} - \{\mathcal{C}\}$, then $\mathcal{C}$ is an articulation point in $\overline{\mathcal{G}}$ separating $\mathcal{C}_x$ from $\mathcal{C}_y$. Since the connected component computation in graph $\mathcal{G}_\mathcal{C} - \{\mathcal{C}\}$ for each $\mathcal{C}$ requires $O(\log(m/K))$ time and $O((m/K)^2)$ processors on a CRCW PRAM [43], finding all articulation points in $\overline{\mathcal{G}}$ separating $\mathcal{C}_x$ from $\mathcal{C}_y$ requires $O(\log(m/K) + \log K)$ time and $O((m/K)^3)$ processors on a CRCW PRAM because $\overline{\mathcal{G}}$ contains $O(m/K)$ vertex clusters (all the other vertices in $\overline{\mathcal{G}}$ are artificial vertices) and $O((m/K)^2)$ edges. $\Box$

Now assume that every vertex v in $\mathcal{T}(\mathcal{C})$ has been assigned a preorder numbering $preorder(v)$. Otherwise, such an assignment can be done in $O(\log K)$ time with $O(K)$ processors using Euler traversal technique [31] because $\mathcal{T}(\mathcal{C})$ is an inverted tree containing $O(K)$ vertices and edges. Also, answering a LCA query in $\mathcal{T}(\mathcal{C})$ can be answered in $O(1)$ time using one processor on an EREW PRAM [42].

Lemma 13 *Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_l$ be l vertex clusters adjacent to a vertex cluster $\mathcal{C}$. $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_l)$ can be found in $O(\log K + \log l)$ time with $O(l)$ processors on a CRCW PRAM, where $1 \leq l \leq \lceil cm/K \rceil$ and c is constant.*

Proof For every $\mathcal{A}_i$, the two vertices in $E(\mathcal{A}_i, \mathcal{C})$ with the smallest and the largest preorder numberings in $\mathcal{T}(\mathcal{C})$ can be found in $O(\log K)$ time using one processor due to the fact that $E(\mathcal{A}_i, \mathcal{C})$ contains $O(K)$ vertices and is maintained as a balanced binary search tree. Let u_i and v_i be the two vertices found from $E(\mathcal{A}_i, \mathcal{C})$. Then, sort the sequence of $2l$ vertices in $\mathcal{T}(\mathcal{C})$ by their preorder numberings in increasing order, which can be done in $O(\log l)$ time with $O(l)$ processors. $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_l)$ is the LCA of the two vertices with the smallest and the largest preorder numberings in the sorted sequence. So, the entire computation takes $O(\log l + \log K)$ time and uses $O(l)$ processors on a CRCW PRAM. $\Box$

Corollary 1 *ρ and β can be found in $O(\log(m/K) + \log K)$ time using $O(m/K)$ processors on a CRCW PRAM.*

Proof It can be derived from Lemma 13 with $l \leq cm/K$, because $1 \leq p, q \leq cm/K$, c is constant. $\Box$

Recall that $\gamma_i = \lambda(\mathcal{Z}_{i,1}, \mathcal{Z}_{i,2}, \ldots, \mathcal{Z}_{i,r(i)})$, where $\mathcal{Z}_{i,j} \in \mathcal{W}_i$, $1 \leq j \leq r(i)$, and $3 \leq i \leq s$. Finding the sequence $\gamma_3, \gamma_4, \ldots, \gamma_s$ takes $O(\log K + \log(m/K))$ time using $O(\sum_{i=3}^{s} r(i)) = O(m/K)$ processors on a CRCW PRAM. Let $\gamma_1 = \rho$ and $\gamma_2 = \beta$, which can be found in the same time and processor bounds. Then, discard those γ_is that are not in $\pi_{\rho,r}$. Let $\gamma'_1, \gamma'_2, \ldots, \gamma'_{s'}$ be the sorted resulting sequence by the preorder numbering of vertices in $\mathcal{T}(\mathcal{C})$ in decreasing order. This sequence can be obtained in $O(\log s) = O(\log(m/K))$ time using $O(s) = O(m/K)$ processors by checking if $\gamma_i = LCA(\rho, \gamma_i)$ which takes $O(1)$ time using one processor, $1 \leq i \leq s \leq cm/K$. Furthermore, let $\mathcal{W}'_0, \mathcal{W}'_1, \ldots, \mathcal{W}'_{s'}$ be the corresponding connected components of $\gamma'_0, \gamma'_1, \gamma'_2, \ldots, \gamma'_{s'}$, where $0 \leq s' \leq s$ and $\gamma'_0 = \rho$.

The complexity for the construction of G_1 is as follows. Checking condition (i) takes $O(1)$ time using $O((m/K)^2)$ processors because there are $O(m/K)$ vertex clusters, and checking condition (ii) takes $O(\log K)$ time using $O(m/K)$ processors which has been proved by the following lemma.

Lemma 14 *[19] Let $\mathcal{C}'$ be any cluster adjacent to $\mathcal{C}$, and let v be any vertex of $\mathcal{T}(\mathcal{C})$. Then, to check whether there are edges in G between $\mathcal{C}'$ and $\mathcal{C}$ incident below v in $O(\log K)$ time with one processor on a CRCW PRAM.*

Notice that we use the same data structure as Galil et. al. did in this part, the proof of Lemma 14 is omitted.

The construction of G_1 thus requires $O(\log K)$ time and $O((m/K)^2)$ processors because $1 \leq s' \leq cm/K$. Finding connected components in G_1 requires $O(\log(m/K))$ time and $O((m/K)^2)$ processors on a CRCW PRAM because G_1 contains $O(m/K)$ vertices and $O((m/K)^2)$ edges [43]. Broadcasting the smallest and the largest indices of the vertices in a connected component to all the vertices in it requires $O(\log(m/K))$ time and $O(m/K)$ processors on an EREW PRAM. Similarly, the construction of G_2 and finding connected components in G_2 takes $O(\log(m/K))$ time using $O((m/K)^2)$ processors on a CRCW PRAM. As a result, the maximum index i_0 of a vertex in the connected component in G_2 containing ρ can be found in the above time and processor bounds. Thus, the path $\pi_{\rho,\gamma'_{i_0}}$ in $\mathcal{T}(\mathcal{C})$ can be found, which does not contain any edge satisfying the two conditions in Lemma 8. Checking whether there is any edge in $\pi_{\gamma'_{i_0},\beta}$ satisfying the two conditions of Lemma 8 can be done in $O(\log K)$ time using $O(m/K)$ processors by Lemma 14, because there are $O(m/K)$ vertices in $\mathcal{T}(\mathcal{C})$. Thus, we have the following theorem.

Theorem 3 *The query about whether x and y are in the same 2-edge connected component in a connected graph $G(V, E)$ can be answered in $O(\log m)$ time using $O(m^{3/4})$ processors on a CRCW PRAM.*

Proof The construction of $\mathcal{G}_{x,y}$ takes $O(\log K + \log(m/K))$ time using $O(K + (m/K)^2)$ processors. Following Theorem 2, type 1 bridge checking takes $O(\log K + \log(m/K))$ time using $O(K + (m/K)^2)$ processors; type 2 bridge checking needs finding all articulation points in $\mathcal{G}_{x,y}$ separating x from y, which requires $O(\log K + \log(m/K))$

time using $O((m/K)^3)$ processors. Checking all the cases in Lemma 9 requires $O(\log(m/K) + \log K)$ time using $O((m/K)^2)$ processors. The theorem then follows, by setting $K = m^{3/4}$. $\square$

3.5 Inserting and deleting edges

In this section we consider the *InsertEdge*(x, y) operation. Let $\mathcal{C}_x$ and $\mathcal{C}_y$ be the vertex clusters including x and y. When a new edge (x, y) is inserted, the degrees of vertices x and y in the original graph G increase by one. If none of them becomes four, only the data structures pertinent to $\mathcal{C}_x$ and $\mathcal{C}_y$ are required to be updated, i.e., $\mathcal{F}(\mathcal{C}_x)$, $\mathcal{T}(\mathcal{C}_x)$, $\mathcal{F}(\mathcal{C}_y)$, $\mathcal{T}(\mathcal{C}_y)$, $E(\mathcal{C}_x, \mathcal{C}_y)$, $E(\mathcal{C}_y, \mathcal{C}_x)$, and $\hat{\mathcal{G}}$ are needed to be updated. Otherwise, if either the degree of x or the degree of y becomes four, then the transformation given in Section 2 will be applied, this leads to a constant number of extra vertices and edges in the resulting graph. Assume that the degree of x in G now is four. We apply the transformation to the cluster $\mathcal{C}_x$. As a result, all data structures pertinent to $\mathcal{C}_x$ must be updated to reflect the change. However, to keep the vertex degree of a graph no greater than three, the insertion of extra vertices may lead to the size of $\mathcal{C}_x$ becoming greater than $3K/2$. We deal with this latter case by splitting $\mathcal{C}_x$ into two vertex clusters $\mathcal{C}'$ and $\mathcal{C}''$. Since the original size of $\mathcal{C}_x$ is between $K/2$ and $3K/2$ and the degree of each vertex in it is no greater than three, the two new clusters $\mathcal{C}'$ and $\mathcal{C}''$ as well as the data structures pertinent to them can be constructed in $O(\log K)$ time using $O(K + (m/K)^2)$ processors on a CRCW PRAM, which is explained as follows. The full representations $\mathcal{F}(\mathcal{C}')$ and $\mathcal{F}(\mathcal{C}'')$, and the tree representations $\mathcal{T}(\mathcal{C}')$ and $\mathcal{T}(\mathcal{C}'')$ of $\mathcal{C}'$ and $\mathcal{C}''$ can be obtained in $O(\log K)$ time with $O(K)$ processors by the discussion in Section 3.3. The balanced binary search tree for the other cluster $\mathcal{C}_j$ adjacent to $\mathcal{C}_x$, $E(\mathcal{C}', \mathcal{C}_j)$, $E(\mathcal{C}_j, \mathcal{C}')$, $E(\mathcal{C}'', \mathcal{C}_j)$ and $E(\mathcal{C}_j, \mathcal{C}'')$ can be constructed in $O(\log K)$ time using $O(K)$ processors because the degree of every vertex in $\mathcal{C}_x$ is at most three. The handling of $\mathcal{C}_y$ can be done similarly. The super graph $\hat{\mathcal{G}}$ and its equivalent graph $\overline{\mathcal{G}}$ can be updated in $O(\log(m/K) + \log K)$ time using $O((m/K)^2)$ processors because they contain $O((m/K)^2)$ vertices.

To perform an insertion of (x, y), we distinguish two cases depending on whether (x, y) is an edge internal to a cluster or an inter-cluster edge. (i) If $\mathcal{C}_x = \mathcal{C}_y = \mathcal{C}$, then (x, y) is an internal edge. As a result, only the data structures related to $\mathcal{C}$, $\mathcal{F}(\mathcal{C})$ and $\mathcal{T}(\mathcal{C})$, are updated, which takes $O(\log K)$ time using $O(K)$ processors on a CRCW PRAM. (ii) If $\mathcal{C}_x \neq \mathcal{C}_y$, then (x, y) is an inter-cluster edge. The data structures needed to be updated are $\mathcal{F}(\mathcal{C}_x)$, $\mathcal{T}(\mathcal{C}_x)$, $\mathcal{F}(\mathcal{C}_y)$, $\mathcal{T}(\mathcal{C}_y)$, $E(\mathcal{C}_x, \mathcal{C}_y)$, $E(\mathcal{C}_y, \mathcal{C}_x)$, and $\hat{\mathcal{G}}$. This takes $O(\log K + \log(m/K))$ time using $O((m/K)^2)$ processors on a CRCW PRAM. In summary, inserting an edge (x, y) causes at most two clusters to be split and at most constant of clusters' data structures to be updated. Following the above discussion, each update requires $O(\log K + \log(m/K))$ time using $O(K + (m/K)^2)$ processors on a CRCW PRAM.

To keep $O(\log m)$ time complexity per update and per query in the worst case, we need to be cautious when dealing with the data structure maintenance. Let m_t be the number of edges in the graph at time t. We show an update at time t can be implemented in $O(\log m_t)$ time with $O(m_t^{3/4})$ processors on a CRCW PRAM. Let $K_t = \lceil m_t^{3/4} \rceil$. When the value of K changes due to an *InsertEdge* or *DeleteEdge* operation, there will be at least $\lceil m_t^{3/4}/2 \rceil$ more updates before K becomes twice as large or one half as small as it was before. Following the idea of Galil et. al. [19] that adjusts a constant number of clusters each time there is an update, this gives a total of $O(m_t^{3/4})$ cluster adjustments. Since there are no more than $O(m_t/K_t) = O(m_t^{1/3})$ clusters that need to be adjusted, and the adjustments can be accomplished before a new round of adjustment starts. Thus, whenever there is an insertion, the clusters can be scanned to find any cluster that is too small, and a constant number of these clusters can be merged with a neighbor if needed.

The processing for the deletion case is similar to that for the insertion case, which is omitted here. Therefore, we have the following theorem.

Theorem 4 *Full dynamic maintenance of 2-edge connected components in a connected graph $G(V, E)$ with m edges can be done in $O(\log m)$ time using $O(m^{3/4})$ processors on a CRCW PRAM.*

Proof From the discussion above and Theorem 3, the theorem follows. □

4 Maintaining 2-Edge and 3-Edge Connected Components in General Graphs

4.1 Maintaining 2-edge connected components in disconnected graphs

We deal with the fully dynamic maintenance of 2-edge connected components in disconnected graphs by extending the results of connected graphs. Let G be a disconnected graph consisting of l connected components. We augment G by adding $l - 1$ *dummy* edges such that the augmented graph is connected and no cycle in the augmented graph includes two dummy edges. Every edge in G is assigned weight 1 and every dummy edge is assigned weight 2. Such augmentation does not change the property of 2-edge connected components in G, i.e., two vertices x and y are in a 2-edge connected component in the augmented graph if and only if they are in the same 2-edge connected component in G. The key here is to maintain a vertex cluster partition of a minimum spanning tree T, instead of maintaining a vertex cluster partition of an arbitrary spanning tree, in the augmented graph.

When inserting an edge $e = (x, y)$, we check whether there is already a dummy edge in T between them. If yes, we decrease the weight of e from 2 to 1; otherwise, we insert this edge with weight 1 to T and update T, which can be implemented in $O(\log m)$ time with $O(m^{2/3})$ processors on a CREW PRAM, using either Ferragina's algorithm [14] or Liang et. al.'s algorithm [35]. If e becomes an edge in the new minimum spanning tree through deleting a dummy edge e', then e' is deleted. When deleting an edge $e = (x, y)$, we check whether e is in T. If yes, T is updated; otherwise, e is deleted. Therefore, we have the following theorem.

Theorem 5 *Full dynamic maintenance of 2-edge-connected components in a disconnected graph $G(V, E)$ can be done in $O(\log m)$ time using $O(m^{3/4})$ processors on a CRCW PRAM.*

Proof It is straightforward from Theorem 4. □

The above proposed algorithm can be further improved if the sparsification technique of Eppstein et. al. [10] is employed. In the following we present an improved algorithm for the problem, using the sparsification technique.

If G is k-edge connected, we can use a sparse k-edge certificate U_k $(= \cup_{i=1}^{k} T_i)$ of G instead of G itself when dealing with the k-edge connectivity of G. Eppstein et. al. [10] have shown that U_k is a strong k-edge certificate of G. Obviously the sparse 2-edge certificate of G is a special case with $k = 2$. For any fixed k, U_k can be maintained in time $O(\log n)$ with $O(n^{2/3})$ processors per update, using the k copies of the algorithm for fully dynamic maintenance of minimum spanning forests. where copy i is used to maintain forest T_i, $i = 1, \ldots, k$. Also, U_k is stable since at most a single edge flips in one of the k minimum spanning forests when there is an update.

Theorem 6 *Full dynamic maintenance of 2-edge connected components in a graph $G(V, E)$ can be done in $O(\log n \log \frac{m}{n})$ time using $O(n^{3/4})$ processors on a CRCW PRAM.*

Proof By applying the sparsification technique, we first build a tree $\mathcal{BT}$ and maintain the sparse 2-edge certificate at every node in $\mathcal{BT}$ by two dynamic data structures, one for 1-edge connectivity which requires $O(\log n)$ time and $O(n^{2/3})$ processors on a CREW PRAM [14, 35], and another for 2-edge connectivity $U_2 = T_1 \cup T_2$, i.e., T_1 and T_2. We then run the parallel algorithm for the maintenance of 2-edge connected components in U_2 described in the beginning of this section at the root node of $\mathcal{BT}$. So, for every edge insertion and deletion, the maintenance time spent for all the data structures involved is $O(\log n \log \frac{m}{n})$ and the number of processors used is $O(n^{3/4})$ because the height of $\mathcal{BT}$ is $O(\log \frac{m}{n})$. Similarly, the query can be answered at the root of $\mathcal{BT}$ in the same time and processor bounds as above. □

4.2 Maintaining 3-edge connected components

Theorem 7 *Fully dynamic maintenance of 3-edge connected components in a graph $G(V, E)$ requires $O(\log n \log \frac{m}{n} + \log n \log \log n / \alpha(3n, n))$ time using $O(n\alpha(3n, n)/ \log n)$ processors per update and $O(1)$ time with a single processor per query on a CRCW PRAM.*

Proof Let U_3 be a sparse 3-edge certificate of G. By applying the reduction technique of Galil et. al. that reduces the k-edge connectivity of G into the k-vertex connectivity of another graph $\varphi_k(G)$, we first construct $\varphi_3(U_3)$ from

U_3, which takes $O(\log n)$ time using $O(n/\log n)$ processors on a CRCW PRAM, because $\varphi_3(U_3)$ contains $O(n)$ vertices and edges. We then maintain a tree $\mathcal{BT}$ in which each node contains a sparse 3-vertex certificate of the subgraph induced by the edges at the leaf nodes in the subtree rooted at the node, using Eppstein et. al's sparsification technique. The computational complexity of the fully dynamic maintenance of 3-vertex connected components is given by Theorem 9 in Section 5.3, the theorem then follows.□

5 Maintaining Biconnected and Triconnected Components in General Graphs

5.1 Finding a strong and sparse k-vertex certificate of G

Eppstein et. al's sparsification technique [10] is applicable only if the certificate $\mathcal{P}$ is both sparse and strong. For k-vertex connectivity of a graph G, Eppstein et. al. [10] have shown that C_k $(= \cup_{j=1}^{k} B_j)$ is a strong, sparse certificate of G, where B_j is a breadth-first search forest in graph $G - \cup_{i=1}^{j-1} B_i$. However, currently the number of processors used in any NC algorithm for finding a breadth-first search forest is no better than the number of scalar operations for matrix multiplication, which is $O(n^{2.376})$ [7]. To reduce the number of processors used, we adopt an alternative approach to find a sparse k-vertex certificate, and show the found certificate is a strong certificate.

To find a sparse k-vertex certificate of $G(V,E)$, we use the *scan-first search* technique [4]. Let $C_k = G(V, \cup_{i=1}^{k} E_i)$ where E_i is the edge set of the spanning forest F_i obtained by applying the scan-first search in graph $G_i = G - \bigcup_{j=1}^{i-1} F_j$. It is obvious that C_k is sparse when k is fixed. In the following we show that C_k is strong too.

Lemma 15 *C_k is a strong k-vertex certificate of $G(V,E)$.*

Proof Let S be a $(k-1)$-vertex cut in any graph $C_k \cup H$ which partitions the vertices in $V-S$ into two subsets A and B. Obviously, H does not contain any edge between a vertex in A and a vertex in B. Let $s_i \in S - \{s_1, s_2, \ldots, s_{i-1}\}$ be the first vertex in the ith scan-first search of $G_i = G - F_1 - F_2 - \ldots - F_{i-1}$ where F_i is a scan-first search spanning forest of G_i.

Let T be a tree rooted at r containing s_1 and $T \in F_1$. We claim either $r \in A$ or $r \in B$, or $r \in S$. If $r \in S$, then $r = s_1$ by the following argument: the index of r is smaller than the index of s_1 because r is visited before s_1 by the definition of scan-first search. This contradicts that s_1 has the smallest index in S. So, either $r \in A$ or $r \in B$. Without loss of generality, assume that $r \in A$. In the following we show that all the edges between s_1 and the vertices in B are included in T, therefore, in F_1, because $T \in F_1$. We proceed as follows.

If $r = s_1$, then all such edges incident to s_1 are in T obviously. Now, consider $r \in A$. Assume that there is an edge $(s_1, v) \in E$ and $v \in B$ but $(s_1, v) \notin T$. We show this is impossible. Since s_1 and v are in the same connected component in G, they must be in the same spanning tree in G. Therefore, both s_1 and v are included in T. Let $p(v)$ be the parent of v in T. We then have either $p(v) \in A$ or $p(v) \in B$, or $p(v) \in S$, and only one of these holds. If $p(v) \in A$, then there is an edge between $p(v)$ $(\in A)$ and v $(\in B)$ after the deletion of all the vertices in S and the edges incident to them. This contradicts that S is a $(k-1)$-vertex cut between A and B. Thus, either $p(v) \in B$ or $p(v) \in S$. Obviously, it is also impossible that $p(v) \in B$ because the index of s_1 is smaller than that of any other vertex in B by the definition of scan-first search. Thus, s_1 must be the parent of v in T because $(s_1, v) \in E$. Therefore, $p(v)$ must be in S and $p(v) = s_1$, i.e., the edge (s_1, v) must be included in T. In other words, all the edges between s_1 and the vertices in B are included in F_1 instead in $C_k - F_1$, and $S - \{s_1\}$ forms a $(k-2)$-vertex cut in $C_k - F_1$. We proceed in the same way on the resulting graph. As a result, each application of scan-first search will eliminate a vertex from G, and in the end F_k must be disconnected. But this can only happen if S is also a vertex cut in G, and hence in $G \cup H$.

We have already shown that any vertex cut in $C_k \cup H$ is also a vertex cut in $G \cup H$. The converse follows immediately from the fact that C_k is a subgraph of G. Hence, C_k is a strong certificate of G. □

5.2 Maintaining biconnected components

Assume that G is connected. Otherwise, two types of data structures are maintained, one for 1-edge connectivity (connected components) and another for biconnectivity (2-vertex connected components). As results, if there is a query to ask whether vertices u and v are in the same biconnected component in G, we first check the data structures for 1-edge connectivity to see whether u and v are connected. If yes, we then run the query algorithm on the data structures for biconnectivity. Otherwise, the response to the query is *false*. It is known that the fully dynamic maintenance of 1-edge connectivity can be done in $O(\log n \log \frac{m}{n})$ time using $O(n^{2/3})$ processors on a CREW PRAM [8, 35].

Theorem 8 *Fully dynamic maintenance of biconnected components of a graph $G(V, E)$ requires $O(\log^2 n)$ time using $O(n\alpha(2n, n)/\log n)$ processors per update and $O(1)$ time with one processor per query, or requires $O(\log n \log \frac{m}{n})$ time using $O(n\alpha(2n, n)/\log n)$ processors per update and $O(\log n)$ time using $O(n\alpha(2n, n)/\log n)$ processors per query on a CRCW PRAM.*

Proof When an edge is inserted or deleted, it results in only $O(1)$ leaf node updates in $\mathcal{BT}$. Meanwhile, the sparse 2-vertex certificates of the involved leaf nodes and their ancestor nodes in the paths in $\mathcal{BT}$ from the leaf nodes to the root node need to be updated. Finding the sparse 2-vertex certificate at a node requires $O(\log n)$ time and $O(n\alpha(2n, n)/\log n)$ processors by Cheriyan et. al.'s algorithm [4]. At the root node, we run Tarjan et.

al's algorithm [44] to maintain biconnected components in the sparse 2-vertex certificate at the root node, while finding biconnected components in a graph with n vertices and m edges requires $O(\log n)$ time and $O(m+n)$ processors on a CRCW PRAM, which can also be implemented in $O(\log^2 n)$ with $O((m+n)/\log n)$ processors by Brent's theorem [3]. Therefore, updating all data structures requires $O(\log n \log \frac{m}{n} + \log^2 n) = O(\log^2 n)$ time and $O(n\alpha(2n,n)/\log n)$ processors when there is an insertion or a deletion. For a query, we just check the root node of $\mathcal{BT}$ to see whether the two vertices are in the same biconnected component, which takes $O(1)$ time using one processor. If we want to reduce the update time further, there is another approach in which, for each update, we recompute the sparse 2-vertex certificates for all involved nodes in $\mathcal{BT}$, and for each query about whether there exist k vertex disjoint paths, we run the algorithm by Khuller et. al. [32] at the root of $\mathcal{BT}$. For the case where $k=2$, their algorithm requires $O(\log n)$ time and $O(n\alpha(2n,n)/\log n)$ processors. So, it requires $O(\log n \log \frac{m}{n})$ time using $O(n\alpha(2n,n)/\log n)$ processors per update and $O(\log n)$ time using $O(n\alpha(2n,n)/\log n)$ processors per query on a CRCW PRAM. □

Notice that there is a faster algorithm for testing k vertex disjoint paths between a pair of vertices [29], which requires $O(k \log\log k \log n)$ time and $O((n^2+k)k(C(n,m)+kn))$ processors on an arbitrary CRCW PRAM. This algorithm is superior to the algorithm of Khuller et. al. [32] only when k is not constant.

5.3 Maintaining triconnected components

Assume that G is biconnected. Otherwise, two types of data structures are maintained, one for biconnectivity and another for triconnectivity of G. That is, if there is a query to ask whether vertices u and v are in the same triconnected component, we first check the data structures for biconnectivity to see whether u and v are biconnected. If yes, we then run the query procedure on the data structures for triconnectivity. Otherwise, the response to the query is *false*. By Theorem 8, the fully dynamic maintenance of biconnectivity can be done in $O(\log n \log \frac{m}{n})$ time using $O(n\alpha(2n,n)/\log n)$ processors on a CRCW PRAM.

Theorem 9 *Fully dynamic maintenance of triconnected components in a graph $G(V,E)$ requires $O(\log n \log \frac{m}{n} + \log n \log\log n/\alpha(3n,n))$ time using $O(n\alpha(3n,n)/\log n)$ processors per update and $O(1)$ time with one processor per query. The algorithm runs in a CRCW PRAM.*

Proof The approach employed is the same as the one used for Theorem 8 except that we replace Tarjan et. al's [44] algorithm for biconnectivity by the algorithm for triconnectivity due to Fussel et. al [17]. Their algorithm for the fully dynamic maintenance of triconnectivity of a graph with n vertices and m edges requires $O(\log n)$ time and $O((m+n)\log\log n/\log n)$ processors on a CRCW PRAM, which also can be implemented in $O(\log n \log\log n/\alpha(m,n))$ time using $O(m+n)\alpha(m,n)/\log n)$ processors by Brent's theorem. □

6 Conclusions

In this paper we have studied the problem of fully dynamic maintenance of k-connectivity of graphs, and presented the first NC algorithms with $\tilde{O}(n)$ work for it where $k = 2, 3$. In particular, our NC algorithm for 2-edge connectivity uses $o(n)$ processors only. For the biconnectivity problem, we have obtained an NC algorithm with $O(n)$ processors. From this algorithm, we have also derived NC algorithms with $O(n)$ processors for the fully dynamic maintenance of 3-edge connectivity and triconnectivity. However, for the addressed problems there is still a gap between the best sequential time complexity for them and the amount of work needed in parallel. It is interesting and challenging to find better NC algorithms for the them to reduce or eliminate the gap.

Acknowledgments We appreciate the three anonymous referees for their invaluable suggestions and comments which helped improve the quality and presentation of the paper. Also, we would like to thank the referees for bringing the paper [28] to our attention. Partial contents of this paper also appeared in [36].

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange networks and PRAM. *IEEE Trans. Computers*, Vol.C-36, 1258–1263, 1987.

[3] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, Vol.21, 201–206, 1974.

[4] J. Cheriyan, M. Y. Kao and R. Thurimella. Scan-first search and sparse certificates—an improved parallel algorithm for k-vertex connectivity. *SIAM J. Comput.*, Vol.22, 157–174, 1993.

[5] F. Y. Chin, J. Lam, and I -N Chen. Efficient parallel algorithms for some graph problems. *Commun. of ACM.*, Vol.25, 659–665, 1982.

[6] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. *Proc. 27th Annual Symp. of Foundations of Computer Sci.*, IEEE, 1986, 478–491.

[7] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Proc. 19th ACM Symp. Theory of Computing*, ACM, 1987, 1–6.

[8] S. K. Das and P. Ferragina. An $o(n)$ work EREW parallel algorithm for updating MST. *Proc. of European Symp. on Algorithms*, LNCS 855, 1994, 331–342.

[9] E. A. Dinitz. Maintaining the 4-edge-connected components of a graph on-line. *Proc. 2nd Israel Symp. on Theory of Computing and Systems*, 1993, 88-99.

[10] D. Eppstein, Z. Galil, G. F. Italiano and A. Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. *J. ACM*, Vol.44, 669–696, 1997.

[11] D. Eppstein, Z. Galil and G.F. Italiano. Improved Sparsification. *Tech. Report*, TR93-20, Department of Information and Computer Science, University of California, Irvine, 1993.

[12] S. Even and Y. Shiloach. An on-line edge deletion problem. *J. ACM*, Vol.28, 1–4, 1981.

[13] P. Ferragina. Static and dynamic parallel computation of connected components. *Inform. Proc. Lett.*, Vol.50, 63–68, 1994.

[14] P. Ferragina. An EREW PRAM fully-dynamic algorithm for MST. *Proc. 9th of Int'l. Conf. of Parallel Processing Symp.*, IEEE, 1995, 93–100.

[15] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, Vol.14, 781–798, 1985.

[16] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *Proc. 32nd Annual Symp. on Foundations of Computer Science*, IEEE, 1991, 632–641.

[17] D. Fussel, V. Ramachandran and R. Thurimella. Finding triconnected components by local replacement. *SIAM J. Comput.*, Vol.22, 587–615, 1993.

[18] Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, Vol.22, 57–61, 1991.

[19] Z. Galil and G. F. Italiano. Fully dynamic algorithms for 2-edge-connectivity. *SIAM J. Comput.*, Vol.21, 1047–1069, 1992.

[20] Z. Galil and G. F. Italiano. Maintaining the 3-edge-connected components of a graph on-line. *SIAM J. Comput.*, Vol.22, 11–28, 1993.

[21] Z. Galil and G. F. Italiano. Fully dynamic algorithms for 3-edge-connectivity. *Manuscript*, 1992.

[22] F. Harary. *Graph Theory.* Addison-Wesley, Reading, MA, 1969.

[23] M. R. Henzinger and V. King. Full dynamic biconnectivity and transitive closure. *Proc. 36th Symp. Foundations of Computer Science*, IEEE, 1995, 664–672.

[24] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. *Proc. 27th Symp. on Theory of Computing*, ACM, 1995, 519–527.

[25] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. *Proc. 24th Int'l. Colloquium on Automata, Languages, and Programming*, LNCS, Springer, 1997, 594–604.

[26] M. R. Henzinger and H La Poutré. Certificates and fast algorithms for biconnectivity in fully-dynamic graphs. *Proc. 3rd European Symp. Algorithms*, LNCS 979, Springer, 1995, 171–184.

[27] M. R. Henzinger and M. Thorup. Sampling to provide or to bound: with applications to fully dynamic graph algorithms. *Random Structures and Algorithms*, Vol.11, 369–379, 1997.

[28] J. Holm, K. de Lichtenberg and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Proc. 30th Symp. on Theory of Computing*, ACM, 1998, 79–89.

[29] K. Iwama, C. Iwamoto and T. Ohsawa. A faster parallel algorithm for k-connectivity. *Inform. Proc. Lett.*, Vol.61, 265–269, 1997.

[30] A. Kanevsky, R. Tamassia, G. Di Battista and J. Chen. On-line maintenance of the four-connected components of a graph. *Proc. 32nd Annual Symp. on Foundations of Computer Science*, IEEE, 1991, 793–801.

[31] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science– Volume A: Algorithms and Complexity*, MIT Press, Cambridge, Mass., 1990.

[32] S. Khuller and B. Schieber. Efficient parallel algorithms for testing connectivity and finding disjoint s–t paths in graphs. *SIAM J. Comput.*, Vol.20, 352–375, 1991.

[33] J. A. La Poutré. Maintenance of triconnected components of graphs. *Proc. 19th Int'l. Colloquium on Automata, Languages and Programming*, LNCS 623, 1992, 354–365.

[34] J. A. La Poutré, J. van Leeuwen and M. H. Overmars. Maintenance of 2- and 3-connected components of graphs, Part I: 2- and 3-edge-connected components. *Discrete Mathematics*, Vol. 114, 329–359, 1993.

[35] W. Liang and B. D. McKay. Fully dynamic maintenance of minimum spanning trees by using a sublinear number of processors. *Manuscript*, (`http://cs.anu.edu.au/~Weifa.Liang/Unpublished_manuscripts`), DCS, Australian National Univ., Canberra, Dec, 1994.

[36] W. Liang and H. Shen. Fully dynamic maintaining 2-edge connectivity in parallel. *Proc. 7th Symp. on Parallel and Distributed Processing*, IEEE, 1995, 216–223.

[37] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, Vol.3, 177–189, 1979.

[38] H. Nagamochi and T. Ibaraki. A linear time algorithm for finding a sparse k-connected subgraph of a k-connected graph. *Algorithmica*, Vol. 7, 583–596, 1992.

[39] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Discrete Math.*, Vol.5, 54–66, 1992.

[40] M. Rauch. Fully dynamic biconnectivity in graphs. *Proc. 33rd Symp. Foundations of Computer Science*, IEEE, 1992, 50–59.

[41] M. Rauch. Improved data structures for fully dynamic biconnectivity. *Proc. 26th Symp. on Theory of Computing*, ACM, 1994, 686–695.

[42] B. S. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, Vol.17, 1253–1262, 1988.

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

[44] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, Vol.14, 862–864, 1985.

[45] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, Vol.7, 433–464, 1992.

Weifa Liang received the Ph.D degree from the Australian National University in 1997, the M.E. degree from University of Science and Technology of China in 1989, and the B.Sc. degree from Wuhan University, China in 1984, all in computer science. He is currently a Lecturer in the Department of Computer Science at Australian National University. His research interests include routing protocol design for high speed networks, design and analysis of parallel and distributed algorithms, data warehousing and OLAP, query optimization, and graph theory. Dr Liang is a member of the IEEE.

Richard P. Brent (M'72-SM'83-F'91) received the B.Sc. (hons) degree in mathematics from Monash University, Australia, in 1968, and the Ph.D. degree in computer science from Stanford University, California, in 1971. Since 1998 he has been Professor of Computing Science at Oxford University, England. His research interests include parallel computer architectures, analysis of algorithms (especially parallel algorithms), numerical analysis and computational number theory. Dr Brent is a member of AMS, SIAM, Sigma Xi, and a fellow of the ACM and the IEEE.

Dr. **Hong Shen** is Professor of Computer Science and Research Director of Parallel Computing Unit in the School of Computing and Information Technology, Griffith University, Australia. He has published extensively in the areas of parallel and distributed computing, high-performance networking, algorithms and data mining. Prof. Shen has served as editor, associate editor and editorial board member of five international journals and chaired several international conferences.