Parallel Fully Dynamic Maintenance of 2-Connected Components

***Abstract*—Finding the biconnected components of a graph has a large number of applications in many other graph problems including planarity testing, computing the centrality metrics, finding the (weighted) vertex cover, coloring, and the like. Recent years saw the design of efficient algorithms for this problem across sequential and parallel computational models. However, current algorithms do not work in the setting where the underlying graph changes over time in a dynamic manner via the insertion or deletion of edges.**

Dynamic algorithms in the sequential setting that obtain the biconnected components of a graph upon insertion or deletion of a single edge are known from over two decades ago. Parallel algorithms for this problem are not heavily studied. In this paper, we design parallel algorithms that obtain the biconnected components of a graph subsequent to the insertion or deletion of a batch of edges. Our algorithms hence will be capable of exploiting the parallelism adduced due to a batch of updates.

We implement our algorithms on an AMD EPYC 7742 CPU having 128 cores. Our experiments on a collection of 10 real-world graphs from multiple classes indicate that our algorithms outperform parallel state-of-the-art static algorithms. [1]

I. Introduction

Analysis of large graphs is a computationally expensive task. With the growing importance of online social networks, research targeted towards fast computations of vital graph metrics is gaining widespread importance [11, 19, 9]. One of the challenges is to update vital graph metrics of massive graphs as the graph changes over time. For instance, consider a social network such as Facebook, which is on average, adding about 500,000 new users every day (about 6 new profiles every second). Such new users will result in adding more edges to the network in terms of "friends". Similarly, the actions of users in "unfriending" results in the network losing some edges every day. Similar settings apply to graphs from other domains such as transportation networks, biological networks, and collaboration networks. We, therefore, note that several classes of real-world networks face *churn*.

With current graph sizes of the order of several million to billions of nodes and edges, even simple computations such as connectivity testing can take a tremendous amount of time. Hence, repeated processing of the entire graph is not feasible. Therefore, we need to employ *dynamic algorithms* that treat the graphs not as a static entity but as a dynamic entity and process updates to the graph without warranting a re-computation of the graph analytic of interest.

Typical computations on such networks such as community detection, clustering, recommendation systems, and obtaining centrality metrics need, therefore, to take the churn in the network into account. Computations listed above, and also other computations such as finding the (weighted) vertex cover, coloring, and the maximal clique [7] use primitives such as finding the 2-connected components of the underlying graph [8, 13] to gain practical efficiency.

In this paper, we study parallel algorithms for maintaining the biconnected components of a graph in a dynamic setting. We first look at an *incremental* algorithm that can process a batch of edge insertions to an existing graph. Next, we study the *fully* dynamic algorithms that support inserting (deleting) a batch of edges to (*resp.* from) the current graph. Usually, and in our algorithms too, processing an insert batch of edges is easier and efficient compared to processing a delete batch. This separation of updates into insert-only batches and delete-only batches allows us to prepare algorithms that cater to insertions and deletions separately without losing any of the practicality of applying the solutions.

Despite such separation into insert-only batches and delete-only batches, the problem is still challenging. As Galil and Italiano [4] point out, the fully dynamic maintenance of bi-connectivity is much more challenging than the fully dynamic connectivity maintenance. The difference stems from the observation that in fully dynamic maintenance of biconnectivity, the number of biconnected components of an n vertex graph may change by $O(n)$ per every edge inserted or deleted.

In the sequential computing model, an effective algorithmic technique called *sparsification* proposed by Eppstein et al. [3] shows how to design efficient dynamic algorithms for several graph problems. The sparsification approach suggests transforming the work associated with an update to a sequence of updates on subgraphs of successively increasing size. The subgraphs themselves are called *sparse certificates*.

However, the sparsification approach has some limitations in the context of our problem. Firstly, the sparsification approach is designed to process a single update in a sequential manner. The current scale of the networks and the rate of churn do necessitate using parallelization and processing a batch of updates to gain efficiency. Secondly, we observe from Figure 1 that the practical advantage of sparsification in a parallel setting is limited. Figure 1 plots the time taken by a parallel adoption of sparsification for maintaining the biconnected components under a single edge insertion and the time taken by the parallel static algorithm of Slota and Madduri [16]. As the sparsification approach includes multiple BFS traversals on graphs of n vertices and $O(n)$ edges, it is not competitive in practice to parallel static algorithms such as [16].

This is an extended version of the paper with the same title.

[1] The implementation and an extended version of this paper is at [1].

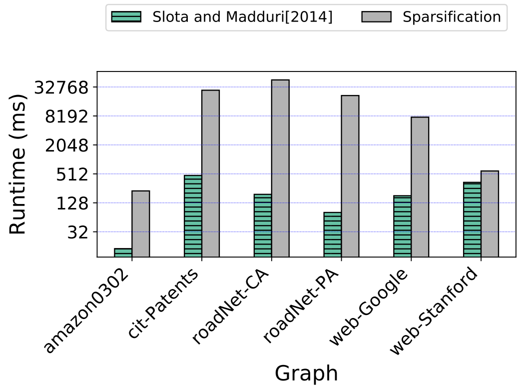

Fig. 1. Performance of the sparsification approach compared to the algorithm of [16] on a single edge insertion.

TABLE I
THROUGHPUT MEASURED AS UPDATES/SECOND ACHIEVED BY OUR ALGORITHMS.

Algorithm/Operation	Insert	Delete	
		Non-Tree	Tree
Incremental	12 M	-	-
Fully Dynamic	68 K	54 K	1 K

We therefore note that existing approaches with respect to dynamic parallel maintenance of bi-connected components offer scope for the design of efficient and practical algorithms.

A. Our Contributions and Results

Our study is motivated by the following factors: (1) the practical possibility that in real-world networks with high churn, one can prepare insert-only and delete-only batches, (2) the possibility of exploiting parallelism that one can adduce via a batch of updates, and (3) the increasing size of the current generation networks (order of billion edges) and the availability of massive parallelism through modern generation hardware.

In this paper, we design novel parallel algorithms for maintaining the biconnected components of an undirected graph in a dynamic setting. We design a parallel algorithm (**cf. Algorithm 1**) that maintains the biconnected components of a graph under a batch of insertions. According to the nomenclature of dynamic graph algorithms, this algorithm is an incremental algorithm. We then extend this algorithm (**cf. Algorithm 2, Algorithm 3**) to fully dynamic algorithms for maintaining the biconnected components of a graph under a batch of insertions and deletions.

Experiments on a set of real-world (from [10] and [2]) graphs suggest that our algorithms offer a significant advantage compared to static algorithms. The throughput that our algorithms achieve on a multi-core CPU is summarized in Table I. Proofs that are omitted from this version, and the source code of the implementation are available at [1].

B. Key Idea

As the algorithm of Slota and Madduri [16] shows, a non-root vertex w in a rooted BFS spanning tree T of G is a cut vertex if the following two properties hold. Property-1) All the children of w in T can reach each other in $G - w$, and Property-2) At least one child of w can reach a vertex v in $G - w$, such that the level of v is at most level of w in T. In our algorithm, we declare that a non-root vertex w in an arbitrary rooted spanning tree T of G as a cut vertex if and only if the children of w can be partitioned into groups in which there exists a group C of vertices such that the following hold. Property-3) All vertices in C can reach each other in $G[V(T_w)] - w$, and Property-4) None of them can reach the parent of w in $G - w$. For any two children x and y of w, we maintain an edge between x and y in an LCA graph G_w (cf. Section II) so that Property-3) can be quickly verified in our dynamic algorithm. Further, we identify the vertices that satisfy Property-4 using the notion of *safe* vertices. The central idea of our fully dynamic algorithm is that when a batch of edges are inserted/deleted, we traverse through necessary fundamental cycles independently in parallel and update our data-structure. In particular, we update the associated LCA graphs, repartition the children of the affected vertices in the associated LCA graphs, and update safe vertices to find the biconnected components of the graph.

C. Related Work

Sequential algorithms for maintaining the biconnected components of a graph under a single edge insertion or deletion are studied in several works, viz. Galil and Italiano [4], Henzinger and King [6]. The sparsification approach of Eppstein et al. [3] results in algorithms that can perform one update in $O(n)$ time. The idea of the sparsification-based approach is to perform the update on a sparse certificate of the original graph. The sparse certificate has only $O(n)$ edges of the original graph.

Ramalingam [12] argues that the time taken to process an update be measured in terms of the impact the new edge has on the analytic. Ramalingam refers to this as the incremental complexity and shows that for many graph problems, one can design sequential algorithms that work in time proportional to the incremental complexity of the problem.

Liang et al. [20] show a PRAM algorithm for maintaining 2-edge connectivity that can process a single update in $O(\log n \log(m/n))$ time using $O(n^{3/4})$ processors. Further, they also show that 2-vertex connectivity can be maintained in time $O(\log^2 n)$ using $O(n\alpha(2n; n) \cdot \log n)$ processors for a single update where $\alpha(;)$ is the inverse Ackermann's function. They also show how to extend their results to the case of maintaining 3-edge and 3-vertex connectivity for a single update. The run time shown in the algorithm of Liang et al. [20] relies on using sparsification on their algorithms and we note from Figure 1 that the algorithm of Liang et al. [20] faces challenges similar to that of [3] in practice.

The sparsification approach [3] has been used in several recent parallel dynamic graph algorithms. Bhowmick and Das [18] use sparsification to maintain graph connectivity in parallel. Khanda et al. [9] study the dynamic maintenance of single-source shortest paths in an evolving graph. The work in [9] presents the batch parallel case for their single update case shown in [19].

McColl et al. [11] show how to handle graph connectivity in a dynamic setting on real-world graphs. They argue that all but a small minority of queries require very little processing time in updating the connectivity as the graph changes. Simsiri et al. [15] also study the connectivity problem on dynamic graphs but introduce a model for handling updates in bulk. On multicore CPUs, they show how to process a stream of edges to be added to the graph in small batches in logarithmic time in parallel using work that is nearly linear in the size of the batch. Their work, however, addresses only the incremental version of the problem.

Jamour et al. [8] use the idea of Green et al. [5] that identifies *affected* vertices due to a single edge insertion or deletion and run BFS from the affected vertices to update the betweenness-centrality values of nodes in a graph. This approach is extended by Regunta et al. [14] to handle the case of a batch update and also to the case of updating the closeness-centrality values of vertices in a graph.

II. Cut Vertices via Unsafe Components

We start this section with standard graph-theoretic terminology. Later, we introduce the notion of safe and unsafe components in LCA-graphs. Finally, we present the necessary and sufficient conditions for a vertex to be a cut vertex using the notion of unsafe components.

We use G to denote an unweighted and undirected graph. A vertex v in G is a *cut vertex* if $G - v$ is disconnected. Similarly, an edge e in G is a *cut edge* if $G-e$ is disconnected. A graph with no cut vertex is called *biconnected*. A maximal biconnected subgraph of G is referred to as a *biconnected component* of G. The number of neighbours of a vertex v in G is called the *degree* of v, denoted by $d(v)$. Let T be a rooted spanning tree of G, and r be the root of T. An edge in G is a *non-tree edge* if it does not appear in T. A non-tree edge $e = (u, v)$ along with the tree path between u and v forms a cycle, C_e, known as a *fundamental cycle*. For a non-tree edge $e = (u, v)$ of G, we use $C_{(u,v)}$ or C_e to denote the fundamental cycle formed with the tree path between u and v and the non-tree edge (u, v). A vertex in T is an LCA vertex if it is the least common ancestor for the end points of some non-tree edge e; for simplicity, we say that v is the LCA for edge e as well as fundamental cycle C_e. We define the notion of base vertices and base edges for a fundamental cycle C_e whose LCA is w as follows: The set $B = \{b \mid b$ is a child of w in T and $b \in V(C_e)\}$ is the set of *base vertices*. Note that $|B| \leq 2$. In case $|B| = 2$, a dummy edge between the two vertices in B is the *base edge* of C_e. For a vertex w in T, the set of base vertices and base edges of all the fundamental cycles whose LCA is w forms a graph G_w, referred to as an LCA-graph. In case w is not an LCA, we can observe that G_w is an empty graph. A vertex u in T is called a *safe vertex* if u is part of a fundamental cycle whose LCA vertex is not equal to the parent of u; otherwise u is *unsafe*. For any two vertices x and y in an LCA-graph G_w, we observe that there exists a path between x and y in G_w if and only if there exists path between x and y in $G[V(T_w)] - w$, where $G[V(T_w)]$ is the subgraph of G induced by the vertices in the subtree rooted at w in T. This observation leads us to define the notion of safe and unsafe components in G_w. A component H in G_w is *safe* if it has at least one safe vertex; otherwise H is *unsafe*. The importance of the presence of a safe vertex in a component of an LCA-graph is described in the following lemma.

Lemma 2.1: Let T be a rooted spanning tree of G, and w be an LCA vertex in T. Let x be a vertex in G_w and y be the parent of w. There is a path between x and y in $G - w$ if and only if x is reachable to a safe vertex in G_w.

Proof: We consider the case that x is reachable to a safe vertex z in G_w without going through w and let us denote such a path by P_1. Then, z is in a fundamental cycle C_e whose LCA is not equal to w. Consequently there is a path P_2 between z and y in $C_e - w$, and the same path lies in $G - w$. The concatenation of P_1 and P_2 leads to a path between x and y in $G - w$.

Now we examine the other direction in which there is a path $P = (x = x_1, \ldots, x_k = y)$ between x and y in $G - w$. Let H be the component in G_w containing x. Let x_i be the last vertex in P that appears in H, where $i \geq 1$. Let x_j be the last vertex in P that is a descendent of x_i, where $j \geq i$. There is a tree path between x_{j+1} and y in $T - w$ due to the existence of the sub-path in P between x_{j+1} and y. Consequently, $e = (x_j, x_{j+1})$ is a non-tree edge. The tree path from y to x_j which contains x_i, another tree path from y to x_{j+1}, along with e forms a fundamental cycle c_e, whose LCA is not equal to w. Thus x_i is safe and the claim holds true. ■

The notion of unsafe components in LCA-graphs helps to derive the necessary and sufficient condition for a vertex to be a cut vertex.

Theorem 2.2: Let T be a rooted spanning tree of an unweighted graph G, r be the root of T, and w be an arbitrary vertex in T. w is a cut vertex in G if and only if one of the following holds true. a) $w \neq r$ and G_w has an unsafe component, b) $w = r$ and G_w has at least two components, and c) w is incident to a cut edge and $d(w) \geq 2$.

Proof: We first prove the sufficient condition for a vertex to be cut vertex. We first consider the case in which $w \neq r$ and G_w has an unsafe component H. Let x be an arbitrary vertex in H, which is unsafe. The vertex x can not reach any safe vertex, because all the vertices in H are unsafe. Therefore, there is no path between x and the parent of w in $G - w$ due to Lemma 2.1, and thus w is a cut vertex. We now consider the second case in which $w = r$, and G_w has at least two components. Let H_1 and H_2 be two components in G_w. Due to the construction of G_w, there are no non-tree edges crossing H_1 and H_2. Hence, there is no path in $G-w$ between a vertex in H_1 and a vertex in H_2, which implies that w is a cut vertex. In the last case, w is incident to a cut edge (w, x), and let y be a neighbour of w such that $x \neq y$. The removal of w separates x from y, and thus w is a cut vertex.

We now prove the contrapositive of the necessary condition. In other words, we assume that the three conditions a), b), and c) are false, and show that w is not a cut vertex in G. If $d(w)$ is 1, then w is not a cut-vertex. Accordingly, in the rest of the proof, we assume that $d(w) \geq 2$ and w is not incident to a cut edge, because the condition c) is false. We first consider the case that $w \neq r$. Let H be a safe component in G_w and x be a safe vertex in H. Every vertex in H can reach x without going through w, as H is connected. Since every component in G_w is safe, from Lemma 2.1 the children of w who appear in G_w can reach the parent of w in $G-w$. Let y be a child of w such that y is not in G_w. Since w is not incident to a cut edge and y is not in G_w, (w, y) belong to a fundamental cycle whose LCA is not equal to w. Therefore y is a safe vertex. Thus, the children of w in T do not disconnect from the parent of w in $G-w$. The neighbours of w that are not children of w are any way reachable from the parent of w in $G-w$. Thus, w is not a cut vertex. We now consider the case that $w = r$. As the number of components in G_w is one, for every two vertices in G_w, there is a path in G without going through w. Thus, the removal of w does not disconnect the graph, and hence w is not a cut vertex. ■

III. A PARALLEL INCREMENTAL ALGORITHM

In this section, we describe a parallel algorithm for inserting a batch $\mathcal{B}$ of edges, to a current graph G and update the underlying data-structure to retrieve the cut vertices and cut edges in $G + \mathcal{B}$. Algorithm 1 shows a high level view of our incremental algorithm. Interestingly, it turns out that we can also obtain an algorithm for obtaining the biconnected components of a graph using Algorithm 1. This is explained in Section III-A.

Key Idea: Based on Theorem 2.2, we mark a vertex v in a graph G as a cut-vertex if the following holds. I) v is incident to a cut-edge (u, v) such that $d(v) \geq 2$. II) v is an LCA for the end vertices of a non-tree edge and G_v contains at least one unsafe component. The key idea in our dynamic algorithms is to update whether an edge is a cut-edge or not and update the components along with their safeness value in LCA graphs associated with LCA vertices, by traversing through the affected fundamental cycles.

For any two vertices x and y in G, the corresponding LCA-graphs G_x and G_y are vertex disjoint. This observation triggers us to introduce the notion of *global* LCA-graph G_*, where G_* is the disjoint union of G_x for every vertex x in G. The global LCA-graph allows to perform edge level parallelism on all edges in G_*.

Data-Structure: We now present various components of the incremental data-structure $\mathcal{D} = (r, p, safe, cutVertex, cutEdge, rep, hasUcomp, numCcomp)$. A spanning tree T of G rooted at r is stored using an integer array $p[\,]$, where $p[v]$ stores the parent of v in T. $safe[\,]$ is a boolean array on the vertices, and we set $safe[v]$ to true if and only if v is a safe vertex. Similarly, $cutVertex[\,]$ and $cutEdge[\,]$ are the boolean arrays on vertices and edges to indicate whether a given vertex or an edge is a cut vertex or a cut-edge, respectively. We ensure that all the vertices in a component of G_* have the same representative. To this end, we use an integer array $rep[\,]$ and $rep[v]$ stores the representative of v in F_*, where F_* is a set of rooted trees, and each tree corresponds to a component in G_*. $hasUcomp[w]$ is set to true if G_w has an unsafe component; otherwise it is set to false. The number of connected components in G_w is stored in $numCcomp[w]$.

Algorithm 1 INCREMENTALBATCH

Input: The incremental data-structure $\mathcal{D}$ of G, a batch $\mathcal{B}$ of edges to be inserted in G.
Task: Update $\mathcal{D}$ and find the cut vertices and cut edges in $G + \mathcal{B}$.

1: **for** each edge $e = (u, v)$ in $\mathcal{B}$ **do in parallel**
2: $\quad (V_*, E_*)$ = RETRIEVE-BASE-VERTICES-EDGES(T, e)
3: F_* = UPDATECONNECTCOMP(F_*, V_*, E_*)
4: **for** each vertex v in F_* **do in parallel**
5: $\quad safe[rep[v]]$ = $safe[rep[v]]$ **or** $safe[v]$
6: **for** each vertex v in G, s.t $rep[v] = v$ **do in parallel**
7: $\quad hasUcomp[p[v]]$ = $hasUcomp[p[v]]$ **or** $\neg safe[v]$
8: $\quad numCcomp[p[v]] = numCcomp[p[v]] + 1$
9: **for** each vertex w in G **do in parallel**
10: $\quad cutVertex[w]$ = ($w \neq r$ and $hasUcomp[w]$ is $true$) **or** ($w = r$ and $numCcomp[r] > 1$) **or** (w is incident to a cut-edge and $d(w) > 1$)

Description of the INCREMENTALBATCH Algorithm. Our incremental algorithm allows us to insert a batch of edges and compute the cut vertices and cut edges in the updated graph. Algorithm 1 details the major steps of our approach.

When a batch $\mathcal{B}$ of edges are inserted, we traverse each fundamental cycle formed with an edge e in $\mathcal{B}$ in parallel, using RETRIEVE-BASE-VERTICES-EDGES(T, e). This function identifies the set V_* of base vertices and E_* of base edges of C_e, which are to be added to G_*. Also, the applicable vertices are marked as safe while traversing a fundamental cycle. Then, all the connected components F_* are updated by inserting V_* and E_* to G_* using UPDATECONNECTCOMP(F_*, V_*, E_*). Later, all the vertices in F_* propagate their safeness to their representatives. A representative or a component is marked as safe if it has at least one safe vertex in Line 5. Further, each representative x in F_w (F_*) propagates whether it is a safe component or not to w (parent of x in T). The existence of an unsafe component in G_w is marked in $hasUcomp[w]$ in Line 7. The number of connected components in G_w is maintained in $numCcomp[w]$ in Line 8. Finally, the three sufficient conditions from Theorem 2.2 are applied in Line 10, to mark the cut vertices in the updated graph.

RETRIEVE-BASE-VERTICES-EDGES(T, e): We first identify the LCA vertex w of e. Later, we go through all the tree edges $e' = (x, y)$ in the fundamental cycle C_e, where x is the parent of y and execute the following: $cutEdge[e']$ is updated to $false$ as e' belongs to a cycle b) If $x \neq w$, $safe[y]$ is updated to $true$. Further, the base vertices and base edge of C_e are added to V_* and E_*, respectively.

UPDATECONNECTCOMP(F_*, V_*, E_*): The purpose of this function is to update the connected components F_* of G_* by including V_* and E_*. We achieve this by using the parallel algorithm to identify connected components [17]. This algorithm maintains the set-disjoint-union data structure in parallel. The

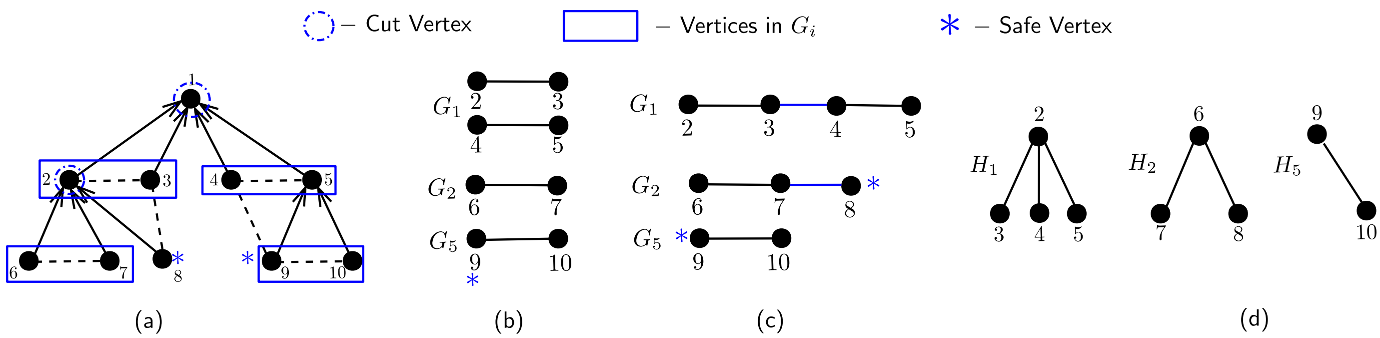

Fig. 2. (a) A graph G and a rooted spanning tree T of G are shown. Solid edges are edges of T, and dashed edges are non-tree edges of T. (b) Non-empty LCA-graphs are shown. (c) LCA-graphs are updated after the batch $\{(3,4),(7,8)\}$ of edges are inserted. (d) The components of LCA-graphs are represented using star trees.

two key operations involved to find the connected components of G_* are namely *grafting* and *short-cutting*. The set of vertices of a connected component in F_* are represented using a rooted tree. For each vertex v in G, we perform the following in parallel to include new bases vertices in F_* as singleton sets: if $v \in V_*$ and $v \notin F_*$, $rep[v] = v$. When an edge in E_* is added across two components, the grafting operation joins the corresponding rooted trees. The other operation short-cutting converts every rooted tree to a star graph. The edges in E_* across the trees are treated as active edges for the next iteration. The above two operations grafting and short-cutting, happens on active edges in every iteration until there are no more active edges. The grafting happens on all active edges in parallel, whereas the short-cutting works on each vertex in F_* in parallel. The time complexity of Algorithm 1 is dominated by Lines 2 and 3. The running time is $O(\ell + \log n)$, and the total work is $O(\ell|\mathcal{B}| + n\log n)$, where ℓ is the average length of a fundamental cycle in G, and $n = |V(G)|$.

Now, we illustrate Algorithm 1 using Figure 2. The vertices 8 and 9 are safe due to the fundamental cycles $C_{(3,8)}$ and $C_{(4,9)}$, respectively. As G_1 has two components $\{2,3\}$ and $\{4,5\}$, and 1 is a root vertex, 1 is a cut vertex due to condition-(b) in Theorem 2.2. By applying condition-(a) in Theorem 2.2, we can deduce that 2 is a cut vertex, because G_2 has an unsafe component $\{6,9\}$ and 2 is not a root vertex. G_5 has only one safe component $\{9,10\}$, and no cut edge is incident to 5, and hence 5 is not a cut vertex. When a batch of two edges $\{(3,4),(7,8)\}$ is inserted, the fundamental cycles $C_{(3,4)}$ and $C_{(7,8)}$ are traversed independently in parallel, and edges $(3,4)$ and $(7,8)$ are added to G_1 and G_2, respectively. Now G_1 has only one component $\{1,2,3,4\}$ and hence 1 is not a cut vertex. Since G_2 has only one safe component, we can declare that 2 is not a cut vertex.

A. Parallel Biconnected Components Algorithm

An edge e in T is a cut edge if $cutEdge[e]$ is true, whereas all the non-tree edges are known to be non-bridges. All the cut edges are trivial biconnected components, which can be computed using $cutEdge[\ ]$. We observe that each non-trivial biconnected component in G is corresponding to an unsafe component in G_*. For each representative u of an unsafe component in G_*, we perform restricted variant of the BFS algorithm starting at u in parallel. In the restricted BFS algorithm, we avoid inserting cut vertices into the queue. Each restricted BFS precisely traverses through the edges of a biconnected component of G.

B. A Static Algorithm

We show how to use Algorithm 1 to derive a static algorithm to find the cut vertices, cut edges and the biconnected components of a graph G. We first obtain a spanning tree T of G and initialize various components in our data-structure $\mathcal{D}$ as follows. Every vertex in T expect the leaves are marked as cut vertices. All the edges in T are marked as cut edges. Every vertex in T is marked as unsafe. We then run Algorithm 1 on the batch $\mathcal{B}$ formed with the edges in $G - T$.

IV. A Fully Dynamic Algorithm

In this section, we first introduce the notion of partially and completely affected LCA vertices, which helps to avoid redundant work. Later, we describe individual components of our dynamic data-structure. Finally, we present fully dynamic algorithms to support inserting/deleting a batch of edges.

In our work, we assume that the underlying graph remains connected when edges are inserted or deleted. A vertex w is called as *completely affected* if the topology of G_w is changed due to insertion/deletion of edges to/from G. Similarly, a vertex w is termed as *partially affected* if the $safeness(v)$ becomes zero from non-zero or vice versa. The completely and partially affected vertices help to avoid recomputing the connected components and recounting the number of unsafe components in G_w.

Data-Structure: We now present various components of the fully dynamic data-structure $\mathcal{D} = (r, p, safe, cutVertex, cutEdge, rep, hasUcomp, numCcomp, Safeness, sup,$ LCA-$strength, weight)$ that are not part of the incremental data-structure. The number of fundamental cycles that support a vertex v as safe is called *safeness* of v and is stored in $safeness[v]$. For a tree edge e, $sup[e]$ is a set of fundamental cycles containing e. LCA-$strength[v]$ stores the number of non-tree edges whose LCA is v. The $weight$ array helps to

Algorithm 2 FULLY-DYNAMIC-INSERT-BATCH

Input: The fully dynamic data-structure $\mathcal{D}$ of an unweighted graph G, a batch $\mathcal{B}$ of edges to be inserted to G.
Task: Update $\mathcal{D}$ and find the cut vertices and cut edges in $G + \mathcal{B}$.

Phase-I

1: **for** each edge (u, v) in $\mathcal{B}$ **do in parallel**
2: INSERT-FUND-CYCLE$(e, pList, cList)$

Phase-II

3: Remove duplicates in $pList$ and $cList$ in parallel
4: **for** each vertex w in $cList$ **do in parallel**
5: F_w = CONNECTCOMP(G_w)
6: **for** each vertex w in $cList$ and $pList$ **do in parallel**
7: $hasUcomp[w]$ = HASUCOMP(G_w)
8: **for** each vertex w in G **do in parallel**
9: $cutVertex[w]$ = ($w \neq r$ and $hasUcomp[w]$ is true) **or** ($w = r$ and NUMCONNECTCOMP$(G_r) > 1$) **or** (w is incident to a cut-edge and $d(w) > 1$)

store the weights for base vertices and base edges. We use the notion of $pList$ and $cList$ to store sequences of partially and completely affected LCA vertices, respectively.

Description of FULLY-DYNAMIC-INSERT-BATCH Algorithm. The pseudo-code of our fully dynamic parallel algorithm, which allows us to insert a batch of edges and maintain the cut vertices and cut edges is shown in Algorithm 2. When a batch $\mathcal{B}$ of edges are inserted, we traverse each fundamental cycle formed with an edge in $\mathcal{B}$ in parallel, using INSERT-FUND-CYCLE$(e, pList, cList)$. This function updates the topology of the necessary LCA graph, updates the safeness of applicable vertices in C_e, and finally all the partially affected and completely affected vertices are appended to $pList$ and $cList$, respectively. All the duplicates from $pList$ and $cList$ are removed in the next step. We then recompute the connected components in G_w using CONNECTCOMP(G_w) for each w in $cList$ in parallel. Similarly, we recount the number of unsafe-components in G_w using NUMUCOMP(G_w) for each vertex w in $pList \cup cList$ in parallel. Finally, we apply Theorem 2.2 to find the cut vertices in the updated graph. The time complexity of Algorithm 3 is dominated by Lines 2 and 5. Hence, the total work is $O(\ell|\mathcal{B}| + m)$, and the running time is $O(\ell + m/|\mathcal{B}|)$, where $m = |E(G)|$.

INSERT-FUND-CYCLE$(e, pList, cList)$: The purpose of this function is to insert the non-tree edge e, traverse the fundamental cycle C_e, which is formed with T and e, and append the partially and completely affected LCA vertices to $pList$ and $cList$, respectively. We first identify the LCA w for the end points of e in T and increment LCA-*strength*$[w]$ by one. For each tree edge e' in C_e, we add e to $sup[e']$. For each vertex v in C_e, such that $p[v] \neq w$, we perform the following: increment $safeness[v]$ by one; If $p[v]$ is an LCA, and $safeness[v]$ is updated from zero to non-zero, then append $p[v]$ to $plist$. Later, we identify the base vertices and base edges of C_e. For each base vertex x of C_e, if x is not in G_w, then we add x to G_w and initialize $weight(x)$ with one; otherwise increment $weight(x)$ by one. Similarly, if the base edge e' of C_e is not in G_w, then e' is added to G_w and set $weight(e') = 1$; otherwise $weight(e')$ is incremented by one. Finally, if any base vertex or base edge is added to G_w, then w is append to $cList$ to note down that G_w is structurally updated.

CONNECTCOMP(G_w): We run the BFS algorithm on G_w and finds a forest F_w of rooted trees, where each tree corresponds to a spanning tree of a component in G_w. During the BFS traversal in G_w starting at a vertex, say x, every newly visited vertex updates its parent to x. This way, all the vertices in a component are stored in the rooted star-graph representation and root vertex is called as a representative..

HASUCOMP(G_w): Every safe vertex x in G_w propagates true to the representative of x. In case the x does not receive true from the children and x is not safe, such a component is treated as unsafe, and x propagates the existence of an unsafe component to w.

NUMCONNECTCOMP(G_w): This function finds the number of connected components in G_w by running the BFS algorithm.

Description of FULLY-DYNAMIC-DELETE-BATCH Algorithm. Algorithm 3 supports to delete a batch $\mathcal{B}$ of edges from the current graph and updates the cut vertices and cut edges dynamically. The batch $\mathcal{B}$ can have both tree and non-tree edges. This algorithm primary consists of three phases. Phase-I is responsible for deleting non-tree edges and deletes the corresponding base vertices and base edges from the affected LCA-graphs, using DELETE-FUND-CYCLE$(e, pList, cList)$. Similarly, Phase-II deletes tree edges in $\mathcal{B}$ from T, connect the disconnected trees of T with non-tree edges, and updates the base vertices and base edges accordingly from the corresponding LCA-graphs. Phase-III computes the connected components in LCA-graphs, identifies the existence of unsafe components, and finally updates the cut vertices.

We now elaborate the steps of Phase-II. Every vertex in a fundamental cycle C_e except the base vertices of C_e and the LCA of C_e is a safe vertex. Accordingly, we can observe that the addition or deletion of a fundamental cycle affects the safe and unsafe components in LCA-graphs. If we delete a tree edge e', LCA-graph of $G_{lca(e)}$ is affected, where $e \in sup(e')$. To track all these changes for all the tree edges to be deleted, we collect the $sup(e)$ for each tree edge e in $\mathcal{B}$ in $globalSup$ in Line 3. Later all the fundamental cycles formed with the non-tree edges in $globalSup$ are deleted using DELETE-FUND-CYCLE$(e, pList, cList)$ in Line 5. Now, we delete each tree edge (x, y) in $\mathcal{B}$ from T in parallel, by updating $p[y]$ to y, where x was the old parent of y. A few non-tree edges in $globalSup$ become tree edges to reconnect the components in T using RECONNECT($T, globalSup$) (Line 7) and each such non-tree edge is called *co-edge*. For all the rest of the non-tree edges in $globalSup$, we add new fundamental cycles and update the necessary data structures using INSERT-FUND-CYCLE$(e, pList, cList)$ in Line 9.

DELETE-FUND-CYCLE$(e, pList, cList)$: The aim of this function is to delete the non-tree edge e, traverse the fundamental cycle C_e, and append the partially and completely affected LCA vertices to $pList$ and $cList$, respectively. The steps of this function are similar to that of INSERT-FUND-CYCLE. We decrement LCA-$strenghth$ and $safeness$ by one rather than incrementing by one. Similarly, we delete edges from sup rather than adding. Also, decrement the weights of base vertices and base edges if their weight is at least 2; otherwise, delete them accordingly. Finally, $p[v]$ is added to $pList$ if $safness[v]$ becomes zero. Likewise, w is added to $cList$ if G_w is structurally updated.

RECONNECT($T, globalSup$): We first construct a simple super graph $\mathbb{G}$ in parallel, where $V(\mathbb{G}) = \{\text{root of a tree } t \mid t \text{ is a component in } T\}$ and $E(\mathbb{G}) = globalSup$. For every edge (t_1, t_2) in $\mathbb{G}$, we associate a *co-edge* (x, y) in $globalSup$ such that x and y are contained in trees rooted at t_1 and t_2, respectively. Then a spanning tree $\mathbb{T}$ of $\mathbb{G}$ is obtained using the parallel BFS algorithm. Further, for each edge (u, v) in $\mathbb{T}$ in parallel, we perform the following to reconnect the components in T: Obtain the co-edge (x, y) of (u, v), where x is the parent of y. Reverse the path between z and y and update the necessary LCA-graphs using REVERSEPATH(z, y), where z is the root of the tree containing y. Finally update the parent of y to x to insert the co-edge (x, y).

REVERSEPATH(z, y): This function aims to reverse the path between z and y and do the necessary updates on certain LCA-graphs due to affected non-tree edges. A non-tree edge e' that belongs to $sup(e)$ is called *affected* if e appears in the reversing path. We first reverse the path between z and y by updating the parents of the vertices involved in the path. We then walk from both the endpoints of an affected non-tree edge towards the root until we encounter a special vertex. The two special vertices found share an ancestor-descendant relationship. The ancestor vertex is the old LCA, and the descendant is the new LCA vertex. The neighbors of the identified LCA vertices in the walk and the special path are the corresponding base vertices. We then update LCA-graphs and LCA-strengths of old and new LCA vertices, along with the safeness values of old and new base vertices. The time complexity of Algorithm 3 is dominated by Line 7. Hence, the total work is $O(nk\ell)$, and the running time is $O(nk\ell/|\mathcal{B}|)$, where k denotes the size of an average cut in G.

V. IMPLEMENTATION DETAILS AND OPTIMIZATIONS

In this section, we discuss the technique we use to obtain LCA in our proposed algorithm. Further, we explain the details about the maintenance of the LCA graphs.

A. Finding LCA

A key operation in both of our proposed algorithms is to find the LCA vertex of a given non-tree edge. In Algorithm 1, we use the level numbers obtained from T to find the LCA vertex. As the spanning tree changes in the Algorithm 3, we do not use the level numbers. Instead, we perform a walk towards the root alternatively from the endpoints of the non-tree edge. We mark the visited ancestors during the walk, and the first ancestor visited by both walks is the LCA vertex. We maintain an integer visited array and mark the visited vertex with a unique non-tree edge number. This allows us to reuse the same visited array for all the non-tree edges, without initializing multiple times. However, each thread maintains a local visited array to avoid race conditions. The former method is more efficient as it only traverses through the tree edges that are part of the fundamental cycle.

Algorithm 3 FULLY-DYNAMIC-DELETE-BATCH

Input: The fully dynamic data-structure $\mathcal{D}$ of an unweighted graph G, a batch $\mathcal{B}$ of edges to be deleted from G.
Task: Update $\mathcal{D}$ and find the cut vertices and cut edges in $G - \mathcal{B}$.

Phase-I

1: **for** each non-tree edge e in B **do in parallel**
2:     DELETE-FUND-CYCLE$(e, pList, cList)$

Phase-II

3: $globalSup = \cup_{e \in \text{ Tree Edges in } B}\ sup(e)$ in parallel
4: **for** each non-tree edge e in $globalSup$ **do in parallel**
5:     DELETE-FUND-CYCLE$(e, pList, cList)$
6: Remove all the tree edges in $\mathcal{B}$ from T in parallel
7: RECONNECT$(T, globalSup)$
8: **for** each non-tree edge e in $globalSup$ such that e is not a co-edge **do in parallel**
9:     INSERT-FUND-CYCLE$(e, pList, cList)$.

Phase-III

10: Call **Phase-II** in Algorithm 2

B. LCA Graphs

In Algorithm 1, we maintain the global LCA graph as an edge list. Each thread maintains a local edge-list to avoid serial append to a global list. For UPDATECONNECTCOMP, each thread processes its own list, thus avoiding the need to merge the lists. On the contrary, we maintain LCA-graphs explicitly in Algorithms 2 and 3.

VI. EXPERIMENTAL RESULTS

In this section, we discuss the datasets and the configuration of the experimental platform. We then study the performance of Algorithm 1 over the static algorithms. Later, we discuss the performance of our Algorithms 2–3. Finally, we conclude by addressing the scalability of our algorithms.

A. Experimental Platform and Dataset

For our experiments, we use a shared memory platform containing two 64 core AMD EPYC 7742 processors spread across two I/O hubs for a total of 128 cores. With simultaneous multi-threading, each core supports two threads of execution hence effectively providing 256 execution lanes. The EPYC processor is based on the Zen 2 micro-architecture from AMD and is built on the 7nm process. It has a base frequency of 2.24 GHz and a working frequency of 3.34 GHz. The processor has a thermal design power (TDP) consumption of 225 watts and contains an L3 cache of 256 MB. The processor is connected to 1 TB DDR4 main memory spread across 8 NUMA nodes. The server is running CentOS 7. For the implementation, we

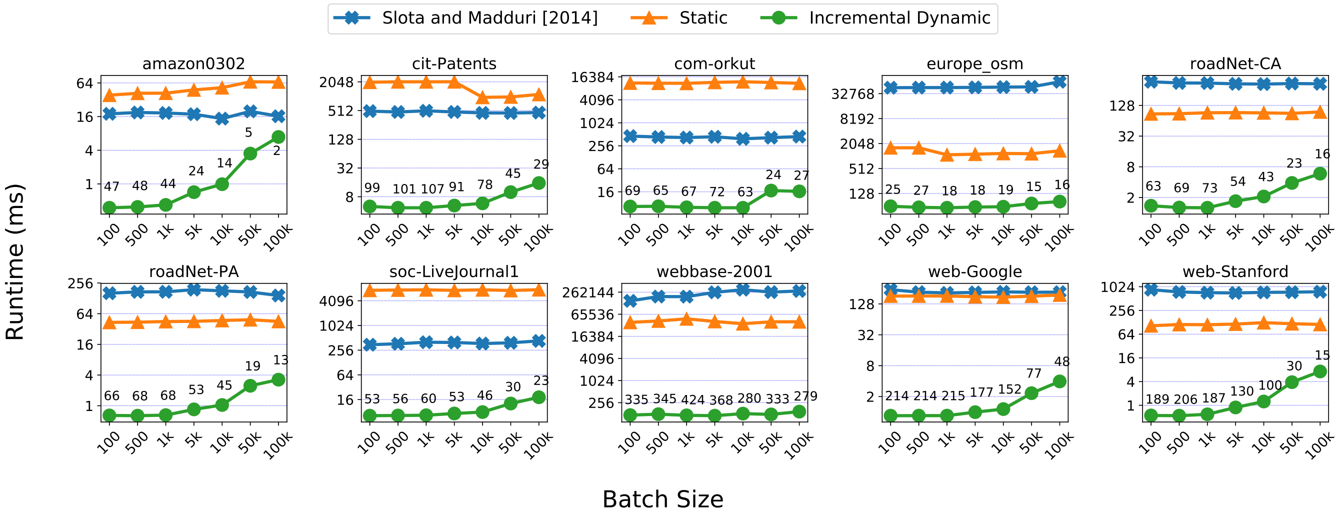

Fig. 3. Performance of Algorithm 1, Algorithm of Slota and Madduri [16], and our static algorithm over varying batch sizes for inserting edges. The numbers on the line corresponding to "Incremental Dynamic" refer to the (rounded) speedup of Algorithm 1 over the best of the other two approaches.

TABLE II
DATASETS OBTAINED FROM [10] AND [2].

Graph name	$\|V\|$	$\|E\|$
Road Networks		
roadNet-CA	1,965,206	2,766,607
roadNet-PA	1,088,092	1,541,898
europe_osm	50,912,018	108,109,320
Web Graph		
web-Stanford	281,903	2,312,497
web-Google	875,713	5,105,039
Product co-purchasing Networks		
amazon0302	262,111	1,234,877
Citation networks		
cit-Patents	3,774,768	16,518,948
Networks with ground-truth communities		
com-Orkut	3,072,441	117,185,083
Social networks		
soc-LiveJournal1	4,847,571	68,993,773
LAW		
webbase-2001	118,142,155	1,019,903,190

compile our code with gcc version 7.5 with OpenMP version 4.5. We compile all programs with the $-O3$ flag.

In Table II, we provide the details of the datasets used in our experiments. We choose a healthy mix of datasets from different publicly available sources in road networks, web graphs, co-purchasing networks, and social networks. We use datasets with edge sizes varying from 1 Million to 1 Billion edges in order to emphasize the scalability of our implementations. To generate a batch of edges to be inserted/deleted, we choose distinct end points uniformly at random. We ensure that edges already present in the graph are not included in an insert batch, and similarly edges not in the current graph are not part of a delete batch. Our experiments, unless mentioned otherwise, use 64 threads on the platform mentioned above. The input to our experiments on the static algorithm from [16] is the graph $G+\mathcal{B}$, where G is the original input graph and $\mathcal{B}$ is the batch of edges to be processed.

B. Performance of the Incremental Dynamic Algorithm (Algorithm 1

We now study the overall performance of Algorithm 1 compared to static algorithms. We compare a multi-core implementations of Algorithm 1 against the multicore implementations of the static algorithm by Slota and Madduri [16] [2], and our static algorithm *(Static)* described in Section III-B. For each of the graphs listed in Table II, we consider inserting edges in batches of size varying from 100 edges to 100,000 edges. Figure 3 shows the time taken by Algorithm 1, the static implementations of Slota and Madduri [16], and our static algorithm. The numbers on top of the line show the speedup (rounded to the nearest integer) of Algorithm 1 over the best of both static algorithms.

We notice from Figure 3 that Algorithm 1 performs 93$\times$ better on average in comparison to the static implementations of Slota and Madduri [16], and our static algorithm. We attribute this speedup to the fact that our incremental algorithm processes only *affected* fundamental cycles in parallel.

C. Performance of the Fully Dynamic Algorithm

From Section IV, we understand that deleting a tree edge is more expensive than deleting a non-tree edge. The deletion of a tree edge involves the deletion and insertion of multiple non-tree edges. Thus, we segregate the queries into three kinds: Insert edges, Delete Non-tree edges, and Delete Tree edges. This separation of queries also allows us to study the performance of Algorithms 2 and 3 independently. Algorithm 3 however works when the batch of edges to delete has a mix of tree and non-tree edges.

Figures 4, 5, and 6 show the performance of Algorithms 2, 3, and the algorithm of Slota and Madduri [16], and our

[2]Based on code from the authors at https://www.cs.rpi.edu/~slotag/publications.html

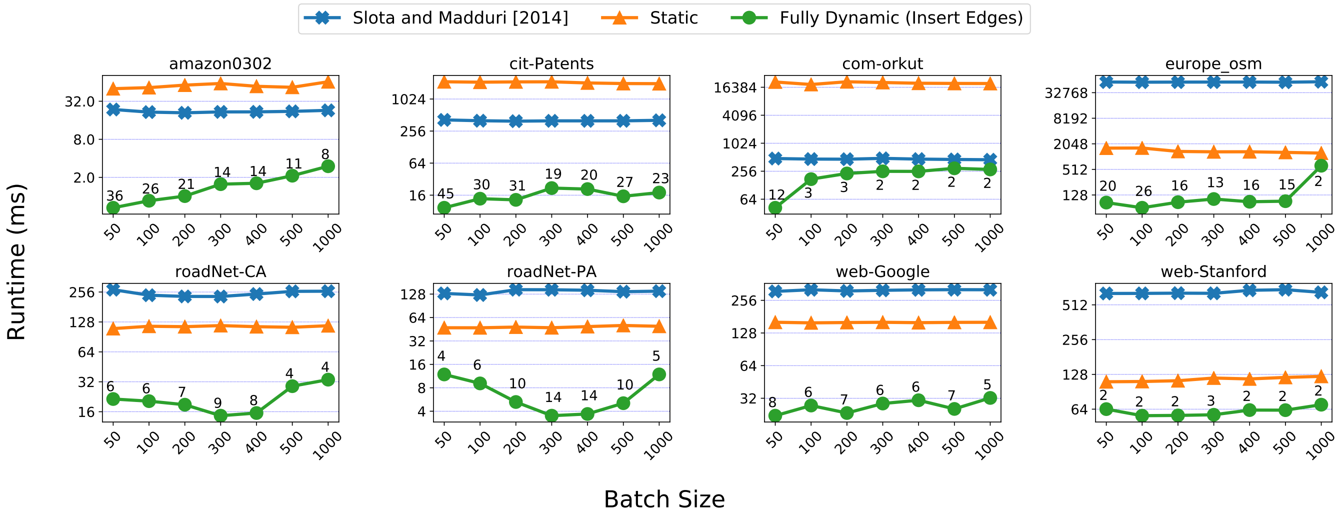

Fig. 4. Performance of Algorithm 2, Algorithm of Slota and Madduri [16], and our static algorithm over varying batch sizes for inserting edges. The numbers on the line corresponding to "Fully Dynamic (Insert Edges)" refer to the (rounded) speedup of Algorithm 2 over the best of the other two approaches.

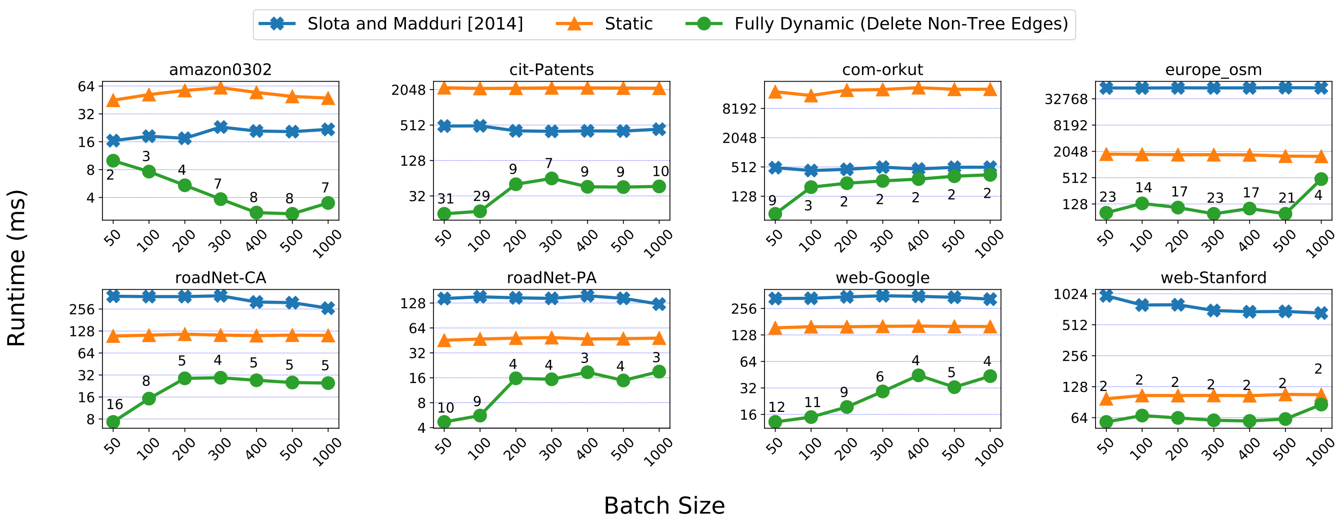

Fig. 5. Performance of Algorithm 3, Algorithm of Slota and Madduri [16], and our static algorithm over varying batch sizes for deleting non-tree edges. The numbers on the line corresponding to "Fully Dynamic (Delete Non-Tree Edges)" refer to the (rounded) speedup of Algorithm 3 over the best of the other two approaches.

static algorithm to process a batch of inserting edges, deleting non-tree edges, and deleting tree edges, respectively. We note from these figures that our algorithms achieve a speedup of $11.17\times$, $7.75\times$, $2.8\times$, respectively, compared to the best of the algorithm from [16] and our static algorithm.

From our experiments, we note that our implementations of our fully dynamic algorithms support a throughput of 68K edge insertions per second, 54K delete non-tree edges per second, and 1K delete tree edges per second.

D. Scalabilty Analysis

We now study the strong- and weak-scalability of Algorithm 1. To this end, we consider three instances from Table II and consider three different batch sizes of 10K, 50K, 100K edges for insertion and vary the number of threads from 4 to 128. Figure 7 shows the results of this experiment. We note from Figure 7 that for the three instances, with batch size fixed at 10K, the run time of Algorithm 1 decreases proportionately as we vary the number of threads from 4 to 128. We also note from Figure 7 that Algorithm 1 exhibits good weak-scaling property as its run time decreases proportionately as the batch size decreases while keeping the number of threads fixed. The plateauing that we notice on increasing the thread count from 64 and 128 is due to the NUMA effect. We observe similar behavior across other instances and other algorithms [3].

[3]See [1] for details.

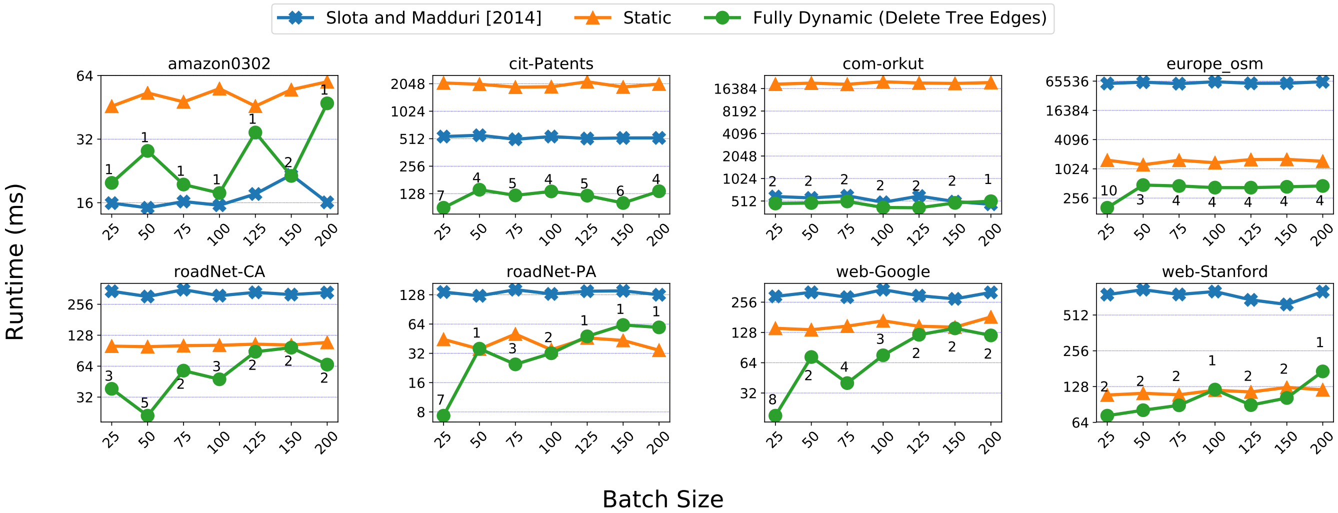

Fig. 6. Performance of Algorithm 3, Algorithm of Slota and Madduri [16], and our static algorithm over varying batch sizes for deleting tree edges. The numbers on the line corresponding to "Fully Dynamic (Delete Tree Edges)" refer to the (rounded) speedup of Algorithm 3 over the best of the other two approaches.

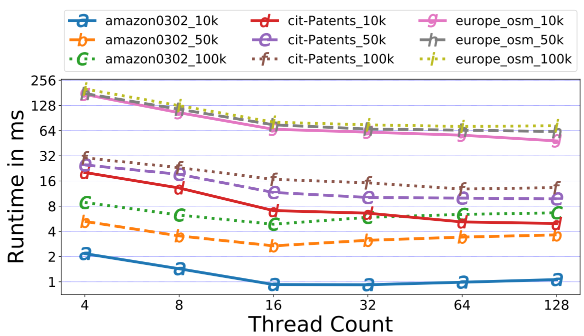

Fig. 7. Performance of Algorithm 1 on varying number of threads over three instances and three different batch sizes.

VII. Conclusions

We designed parallel algorithms to identify the biconnected components of a graph when a batch of edges are inserted or deleted. Our algorithms traverse only the *affected* fundamental cycles and update the necessary LCA-graphs to finally update the biconnected components. Our algorithms significantly outperform existing approaches. We examined the running times of our incremental algorithm and fully dynamic algorithms against the best of Slota and Madduri [16] implementation and our own static algorithm, and obtained significant benefits. Our algorithm allows us to insert a non-tree edge and delete a non-tree edge in parallel. As we process non-tree edges followed by tree edges, and the graph is always connected, our algorithm simultaneously supports a mixed batch of inserting/deleting edges.

References

[1] Anonymous, "Parallel fully dynamic maintenance of 2-connected components," https://tinyurl.com/Dyn-Par-BCC, 2021.

[2] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, pp. 1–25, Dec. 2011.

[3] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, "Sparsification - A technique for speeding up dynamic graph algorithms," *J. ACM*, vol. 44, no. 5, pp. 669–696, 1997.

[4] Z. Galil and G. F. Italiano, "Fully dynamic algorithms for 2-edge connectivity," *SIAM J. Comput.*, vol. 21, no. 6, pp. 1047–1069, 1992.

[5] O. Green, R. McColl, and D. A. Bader, "A fast algorithm for streaming betweenness centrality," in *IEEE SocialCom*, 2012, pp. 11–20.

[6] M. R. Henzinger and V. King, "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *J. ACM*, vol. 46, no. 4, pp. 502–516, 1999.

[7] D. S. Hochbaum, "Why should biconnected components be identified first," *Discrete Applied Mathematics*, vol. 42, pp. 203–210, 1993.

[8] F. T. Jamour, S. Skiadopoulos, and P. Kalnis, "Parallel algorithm for incremental betweenness centrality on large graphs," *IEEE TPDS*, vol. 29, no. 3, pp. 659–672, 2018.

[9] A. Khanda, S. Srinivasan, S. Bhowmick, B. Norris, and S. K. Das, "A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks," *IEEE TPDS*, 2021.

[10] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data.

[11] R. McColl, O. Green, and D. A. Bader, "A new parallel algorithm for connected components in dynamic graphs," in *20th IEEE HiPC*, 2013, pp. 246–255.

[12] G. Ramalingam, *Bounded Incremental Computation*. Lecture Notes in Computer Science, Springer, 1996, vol. 1089.

[13] A. E. Sariyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek, "Betweenness centrality on GPUs and heterogeneous architectures," in *6th GPGPU Workshop*, 2013, pp. 76–85.

[14] K. Shukla, S. C. Regunta, S. H. Tondomker, and K. Kothapalli,

"Efficient parallel algorithms for betweenness- and closeness-centrality in dynamic graphs," in *ACM ICS*, 2020, pp. 10:1–10:12.

[15] N. Simsiri, K. Tangwongsan, S. Tirthapura, and K. Wu, "Work-efficient parallel union-find with applications to incremental graph connectivity," in *in Proc. Euro-Par*, 2016, pp. 561–573.

[16] G. M. Slota and K. Madduri, "Simple parallel biconnectivity algorithms for multicore platforms," in *21st IEEE HiPC*, 2014, pp. 1–10.

[17] J. Soman, K. Kothapalli, and P. J. Narayanan, "A fast GPU algorithm for graph connectivity," in *24th IEEE IPDPS Workshops*, 2010, pp. 1–8.

[18] S. Srinivasan, S. Bhowmick, and S. K. Das, "Application of Graph Sparsification in Developing Parallel Algorithms for Updating Connected Components," in *IPDPS Workshops*, 2016, pp. 885–891.

[19] S. Srinivasan, S. Pollard, S. Das, B. Norris, and S. Bhowmick, "A Shared-Memory Algorithm for Updating Tree-Based Properties of Large Dynamic Networks," *IEEE Trans. on Big Data*, 2018.

[20] W. Liang, R. P. Brent, and Hong Shen, "Fully dynamic maintenance of k-connectivity in parallel," *IEEE TPDS*, vol. 12, no. 8, pp. 846–864, Aug 2001.