International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 1, pages 87–106, January 2016

Efficient Multicore Algorithms For Identifying Biconnected Components[1]

Meher Chaitanya
meher.c@research.iiit.ac.in
and
Kishore Kothapalli
kkishore@iiit.ac.in

International Institute of Information Technology
Hyderabad, Gachibowli, India 500 032.

Received: July 30, 2015
Revised: October 26, 2015
Accepted: December 1, 2015
Communicated by Akihiro Fujiwara

Abstract

In this paper we design and implement an algorithm for finding the biconnected components of a given graph. Our algorithm is based on experimental evidence that finding the bridges of a graph is usually easier and faster in the parallel setting. We use this property to first decompose the graph into independent and maximal 2-edge-connected subgraphs. To identify the articulation points in these 2-edge connected subgraphs, we again convert this into a problem of finding the bridges on an auxiliary graph.

It is interesting to note that during the conversion process, the size of the graph may increase. However, we show that this small increase in size and the run time is offset by the consideration that finding bridges is easier in a parallel setting. We implement our algorithm on an Intel i7 980X CPU running 12 threads. We show that our algorithm is on average 2.45x faster than the best known current algorithms implemented on the same platform. Finally, we extend our approach to dense graphs by applying the sparsification technique suggested by Cong and Bader in [7].

Keywords: Biconnected components, Least common ancestor, 2-edge connected components, Articulation points

1 Introduction

The biconnected components of a given graph are its maximal 2-connected subgraphs. Finding the biconnected components of a graph is an important problem in graph theory. This problem has been pushed to the fore recently for its use as a subroutine in other graph based computations such as shortest paths, betweenness-centrality, and the like [6]. The problem has applications to network

[1] A part of this work has appeared previously in 17th Workshop on Advances in Parallel and Distributed Computational Models, IPDPS Workshops, 2015.

design too as bridges and articulation points in a communication network indicate lack of robustness and resiliency to single edge and vertex failures.

Of particular interest has been to design algorithms for sparse graphs as most real world graphs tend to be sparse in nature [2]. In this direction, Madduri and Slota [15] present the most recent algorithm for finding the biconnected components of a given graph on multicore architectures. The work of [15] presents two algorithms, one based on performing multiple BFS, and one based on graph coloring. The algorithms of Madduri and Slota improve on previous algorithms for the problem in the parallel setting, most notably those of Tarjan and Vishkin [19] and of Cong and Bader [7]. However, it is not clear as to which of the two algorithms is a better choice on a given graph. Adding to the difficulty, the algorithms can differ in their run time significantly, of the order of 5x in some cases.

In this paper, we design a simple and efficient algorithm for finding the biconnected components of a given sparse graph. Our algorithm has two interesting characteristics to it. Firstly, we observe that identifying the bridges of a graph can be done more easily compared to identifying the articulation points. Secondly, based on our first observation, we construct an auxiliary graph that is *larger* in size than the given graph with the property that bridges in the auxiliary graph can be used to identify the articulation points in the input graph. Finally, we provide an extension of this approach to identify the biconnected components in dense graphs by using edge-pruning technique proposed in [7].

Our algorithm is easy to parallelize. An implementation of our algorithm on an Intel i7 980X CPU running 12 threads is 2.45x faster than both the algorithms presented in [15] on an average on a wide variety of real-world graphs. We also believe that some of the techniques we introduce in this paper can be of independent interest in designing other graphs algorithms.

1.1 Our Contributions

Some of the notable contributions from our paper are as follows. We design a new parallel algorithm, Algorithm LCA-BiCC, for finding the biconnected components of a graph.

One of the important aspects that motivate our algorithm and its approach is to note that identifying bridges (an edge whose removal disconnects the graph) in a given undirected graph is a much simpler task in general, and particularly so in sparse graphs. In a sparse graph, one can build a BFS tree, or just a spanning tree, and mark all edges that are part of some fundamental cycle (i.e., the cycle induced[2] by any non-tree edge). As a result, edges that are not marked are simply the bridges of the graph. We use this idea in our algorithm (see also Algorithm 3). Since the depth of the BFS tree in real world graphs is observed to be small, this process of identifying the bridges is usually very fast. The easy identification of bridges allows us to treat the graph obtained after removing the bridges as consisting of independent maximal 2-edge connected subgraphs. On these subgraphs, we now have to find articulation points (a vertex whose removal disconnects the graph) and identify the biconnected components. Using the observation that finding bridges is easier, we build an auxiliary graph G'_i for each maximally 2-edge-connected subgraph G_i, $i \geq 1$, of G such that bridges in G'_i can be used to quickly locate the articulation points in G_i (and hence in G). This mechanism of removing the bridges to arrive at a decomposition consisting of independent and maximal 2-edge-connected subgraphs, and exploiting the properties of 2-edge-connected subgraphs, may be of independent interest in the design of other graph algorithms in the parallel setting.

We propose modifications to one of the algorithms from [15]. These modifications are based on the observations from the design of Algorithm LCA-BiCC (Algorithm 2), and the modified algorithm is noticed to be 1.46x faster than Algorithm LCA-BiCC. We extend our results to dense graphs by making use of the sparsification technique mentioned by Cong and Bader in [7].

1.2 Related work

There are several known parallel algorithms for identifying the biconnected components in the given graph. For a graph G with n vertices and m edges, Eckstein [8] provided the first parallel algorithm that takes $\mathcal{O}(d \log^2 n)$ time using $\mathcal{O}((n+m)/d)$ processors, where d is the diameter of the BFS tree.

[2]See [21] for graph theoretic notataions.

Savage and JáJá [14] proposed parallel algorithms on CREW PRAM for both sparse and dense graphs. One of them takes $\mathcal{O}(\log^2 n)$ time on $\mathcal{O}(n^2/\log^2 n)$ processors and is suitable for dense graphs. The other is for graphs which are sparse in nature and it requires $\mathcal{O}(\log^2 n \log k)$ time with $\mathcal{O}(mn + \log^2 n)$ processors where k is the number of biconnected components in the given graph.

Tsin and Chin [20] developed an optimal algorithm for dense graphs that runs in $\mathcal{O}(\log^2 n)$ time using $\mathcal{O}(n^2/\log^2 n)$ processors. Tarjan and Vishkin [19] provided a $\mathcal{O}(\log n)$ time algorithm that uses $\mathcal{O}(n+m)$ processors. Cong and Bader [7] demonstrated a parallel speedup over the Tarjan-Vishkin algorithm on symmetric multiprocessor systems. Edwards and Vishkin [9] demonstrated the parallel speedup over Tarjan serial algorithm [18] and Tarjan-Vishkin algorithm [19] on XMT manycore computing platform. Most recently Madduri and Slota [15] proposed two parallel algorithms that provides considerable speedup over the Cong and Bader approach [7]. A brief description of these algorithms is presented in Section 2.

1.3 Organization of the Paper

The rest of the paper is organized as follows. In Section 2, we briefly describe the existing algorithms that are directly relevant to our current work. In Section 3, we develop the required lemmata for our algorithm that is described in Section 4. We provide the implementation details of our approach in Section 5. Experimental results of our algorithm are presented in Section 6. Further improvements are discussed in Section 7 along with experiments. An extended approach to handle dense graphs is presented in Section 8. The paper ends with some concluding remarks in Section 9.

2 Existing Algorithms

We review in brief some of the parallel algorithms that are most relevant to our present work.

2.1 Tarjan-Vishkin Parallel Algorithm (TV) [19]

The algorithm of Tarjan and Vishkin for identifying the biconnected components in G is designed for the PRAM model [19] and requires $\mathcal{O}(\log(n))$ time with $\mathcal{O}(n+m)$ processors. The main steps in the Tarjan-Vishkin algorithm are as follows. A rooted spanning tree T for the input graph G is constructed (For notations interested reader can refer to [21]). Using T, two functions `low`, the lowest vertex that is either a descendent of v or adjacent to the descendent of v by an edge in $G \setminus T$, and `high`, the highest vertex that is either a descendent of v or adjacent to the descendent of v by an edge in $G \setminus T$, are computed for each vertex $v \in G$. These functions help define an auxiliary graph G' such that the connected components of G' are the biconnected components of G.

2.2 Cong and Bader [7] Improvement to TV (TV-filter)

An experimental study by Cong and Bader [7] provides an improvement to the Tarjan-Vishkin ([19]) by removing non-essential edges in its computation. This leads to a significant reduction in computing `low`, `high` values and connected components computation. In particular, they define an edge e as *non-essential* for biconnectivity if removing e does not change the biconnectivity of the component it belongs to. They show that edges of G that are not in a BFS tree T and are also not in a spanning forest F of $G \setminus T$ are non-essential. However, in this algorithm, T must be a BFS tree, which can be difficult to compute in parallel compared to a simple spanning tree. The runtime for the Cong-Bader approach is $\mathcal{O}(dia + \log n)$ where dia is the diameter of the graph. The algorithm is as follows.

2.3 BFS-BiCC

The BFS-BiCC algorithm [15] is an improvement over Cong-Bader's approach [7] on sparse graphs. This algorithm is similar to that used by Eckstein [8]. A rooted BFS tree T of the input graph G is constructed. To identify the articulation points, the BFS-BiCC algorithm considers every vertex u

Algorithm 1 CONG-BADER(G)

```
1: procedure CONG-BADER(Graph G)
2:     T ← BFS(G)  /*Phase I*/
3:     F ← BFS(G\T)
4:     G^1 ← (F∪T)
5:     Biconnected_Components = TARJANVISHKIN-PARALLEL(G^1) /*Phase II*/
6:     for all e = (w,v) ∈ G \ (F ∪ T) do  /*Phase III*/
7:         label e to be in biconnected component containing w and p(w)
8:     end for
9: end procedure
```

and performs a BFS on the graph after removing its parent $P(u)$ (according to T) from the graph G. During this process BFS-BiCC algorithm keeps track of the level of vertices reached from u. If any vertex w with level $L(P(u))$ or less is reached and u has a path to all its siblings after the removal of $P(u)$, then $P(u)$ is not an articulation point. The main drawback of this approach is that it performs BFS from all the vertices of the graph. While there are optimizations introduced to stop some of these breadth-first traversals, there exist graphs on which such early stopping cannot be done and on such graphs, Algorithm BFS-BiCC from [15] suffers heavily.

2.4 Color-BiCC

The Color-BiCC algorithm [15] is an improvement over the Cong and Bader approach [7] on sparse graphs. This is an iterative strategy that is similar to recursive doubling used to compute connected components in undirected graphs as well as weakly and strongly connected components in directed graphs [16]. Let $par(v)$ signify the parent articulation point that separates the vertex v from the root. This algorithm is based on the observation that any two vertices in a biconnected component will have their least common ancestor set to the $par(v)$. The goal of this approach is to color all vertices in the biconnected component with a parent level articulation point that is separating the vertex from the root. As we will show in Figure 5 that the Color-Bicc algorithm is heavily dependent on the structure of the graph and can be slower than the sequential Tarjan [18] approach in some cases.

3 Our Approach for BiCC on Sparse Graphs

In this section we present a simple yet efficient algorithm for identifying the biconnected components of graph G. Our algorithm is based on our experimental evidence that identifying bridges of a graph in a parallel setting is a much easier and simpler task. Based on the above observation, we initially decompose the graph into maximal 2-edge-connected components $G_1, G_2, \cdots$. For each such component, G_i, $i \geq 1$, we construct an auxiliary graph G'_i where articulation points in G_i translate to bridges in G'_i. Therefore, identifying the bridges of G'_i allows us to identify the articulation points of G_i, and hence those of G. Using this information, we then identify the biconnected components of G.

We develop two results (Lemmata 1, 2 and 3) below that will help us present our algorithm. Towards this, let T be a rooted BFS tree of G and LCA denotes the least common ancestor. Each non-tree edge (u, v) in $G \setminus T$ is a cross edge that connects two different branches of a tree. For an edge e to be a bridge, e must be part of BFS spanning tree and e cannot be on any cycle induced by the non-tree edges $(u, v) \in G \setminus T$.

Now, consider the graph G' obtained by removing the bridges of G. The resulting graph consists of maximal 2-edge connected components $G_1, G_2, \cdots$, such that for each pair of vertices u, v in the same 2-edge-connected component, there are at least two edge disjoint paths between u and v. We can now treat each such component independently and in parallel to identify the articulation points within each component. These will also be articulation points of G.

Let G be a 2-edge-connected graph and T_G be a rooted BFS tree of G. We use the notation $V_{\text{lca}}(G)$ to denote the set of vertices that are the LCA of the end points of some non-tree edge of G according to a given BFS on G. We can classify the vertices of G into two categories as follows.

1. Potential articulation points: We will prove shortly that all the vertices $V_{\text{lca}}(G)$ belong to this category. A subset of the vertices in $V_{\text{lca}}(G)$ is the set of articulation points of G.

2. Non-articulation points: These are the set of safe vertices whose removal does not disconnect the graph. All the vertices $v \in V(G) \setminus V_{\text{lca}}(G)$ belong to this set.

The above categorization is supported by the following lemma which shows that vertices not in $V_{\text{lca}}(G)$ cannot be articulation points in G.

Lemma 1 *Let G be a 2-edge-connected graph and let T be a BFS tree of G. If v is an articulation point of G, then $v \in V_{lca}(G)$.*

Proof: On the contrary, assume that a vertex v is an articulation point and is not the LCA of any non-tree edge of G. If v is on only one cycle in G, then v cannot be an articulation point. So, we assume in the rest of the proof that v is on at least two cycles in G. Let $C_1, C_2, \cdots, C_k$ be the fundamental cycles induced respectively by non-tree edges $e_1, e_2, \cdots, e_k \in G \setminus T$ and pass through vertex v. Let C_i and C_j be *any* two cycles from the set $\{C_1, C_2, ..., C_k\}$ induced by non-tree edges e_i and e_j respectively. Let vertices x, y be LCA of the endpoints of e_i and e_j respectively. It is evident that x and y should be the ancestors of v as v lies on both the cycles and $v \notin \{x, y\}$. The relation between x and y can be categorized as follows.

- $x = y$: In this case the two cycles C_i and C_j share the same LCA say x and also the vertex v. This implies that C_i and C_j share at least an edge (as there are at least two vertices, x and v, common to both C_i and C_j). So, even after the removal of v, all edges belonging to C_i and C_j remain in a single biconnected component. Hence, v is not an articulation point.

- $x \neq y$, $z = LCA(x, y)$, and $z \notin \{x, y\}$: As x and y are ancestors of v there is a path x to v and v to y in T. As z is the ancestor of x and y there is a path z to x and y to z in T. This concludes that there is a path from $z \rightsquigarrow x \rightsquigarrow v \rightsquigarrow y \rightsquigarrow z$ which leads to a cycle in T. However, T is a BFS tree and cannot have cycles. Therefore, our assumption that v is an articulation point is not valid.

- $x \neq y$ and $\text{LCA}(x, y) \in \{x, y\}$: Without loss of generality, we will assume that $y = \text{LCA}(x, y)$. Let C_i and C_j be any pair of cycles induced by non-tree edges e_i and e_j and pass through v with $\text{LCA}(e_i) = x$ and $\text{LCA}(e_j) = y$. Since y is a proper ancestor of x, there is a path from $x \rightsquigarrow v$ (in T and also in G) that is common to C_i and C_j. This ensures that there is at least an edge common between the cycles C_i and C_j. Similar to the case where $x = y$, this allows us to argue that even after the removal of v, all edges of C_i and C_j remain in a single connected component. Since the above holds for any pair of cycles passing through v, v is not an articulation point.

□

The above lemma indicates that we only have to check whether vertices in $V_{\text{lca}}(G)$ of G are articulation points in G. To find these articulation points, we now construct an auxiliary graph G' as follows. Let T_G be a BFS tree of G. We use the notation $\text{LCA}(e)$ to refer to the LCA of the end points of the edge e. (Such a notation is used in other earlier works too [12]).

Initialize $V(G') = V(G)$ and $E(G') = E(G)$. For every non-tree edge e in $G \setminus T_G$, compute the $LCA(e) = x$. Let b_1 and b_2 be the neighbours (also called as base vertices) of x in the fundamental cycle induced by e. We now remove the edges xb_1 and xb_2 from G', add an *alias* vertex for vertex x as x' to G', and add edges xx', $x'b_1$, and $x'b_2$ to $E(G')$. All other edges in G with end points as x, b_1 or b_2, remain unchanged. Also, if k cycles $C_1, C_2, \cdots, C_k$ induced by the non-tree edges $e_1, e_2, \cdots, e_k$, respectively share any of the base vertices, then their common LCA vertex, say v,

cannot be an articulation point with respect to the cycles $C_1, C_2, \cdots, C_k$ as all these cycles share at least an edge. Thus, we create only one alias vertex v' in the auxiliary graph for v with respect to the above k cycles.

The alias vertices and edges with one end point as an alias vertex have the property that articulation points of G are transformed as bridges in the auxiliary graph G' as we will show shortly. An example of the construction of the auxiliary graph is shown in Figure 1. In Figure 1, we consider a BFS of the graph in Figure 1(a) with z as the source vertex, and edges wx and ts as the non-tree edges. In the following, we show that bridges in G' can be used to identify the articulation points of G.

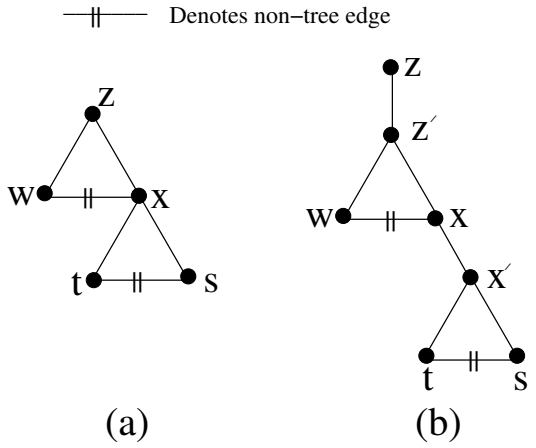

Figure 1: Figure (b) shows the auxiliary graph for Figure (a). Edges wx and ts are kept as non-tree edges and the rest of the edges are tree edges according to a BFS starting from vertex z as the source.

Lemma 2 *Let G be a 2-edge-connected graph, T_G a rooted BFS tree of G with root as r, and G' be the auxiliary graph of G constructed as earlier. For vertices u in G' with $u \neq r$, u is an articulation point of G iff u is an end point of some bridge uv in G' with $u \in G$ and $v \notin G$.*

Proof:

(only − if) ⇐: Consider a vertex u which is not an articulation point in graph G with $u \neq r$. We will show that any edge of type uu', where u' is the alias of u, cannot be a bridge in auxiliary graph G'.

Let $\mathcal{C} := \{C_1, C_2, \cdots, C_k\}$ be the cycles that pass through vertex u in G. The relation between vertex u and the such cycles can be categorized as follows.

- u is not the LCA of any of the end points of non-tree edges that induces cycles in $\mathcal{C}$: In this case, no alias vertices are introduced in G' due to u. Therefore, bridges with u as one end points does not exist in G'. (Note that G is already 2-edge-connected and has no bridges).

- u is the LCA any two non-tree edges that induces cycles C_i and C_j in $\mathcal{C}$: According to the construction of G', two alias vertices u_i and u_j are introduced in the auxiliary graph G'. Further, two edges uu_i and uu_j are also added to G'. An example is illustrated in Figure 2.

  Let x and y be any two distinct vertices on the cycles C_i and C_j respectively. Since u is not an articulation point in G, there must be some path P_{xy} in G' between x and y that does not pass through u as shown in Figure 2. The path P_{xy} along with paths P_{yu_j}, $P_{u_j u_i}$, and $P_{u_i x}$ forms a simple cycle in G'. This indicates that edges uu_i and uu_j on this cycle cannot be bridges in G'. So there is no bridge in G' with one of the endpoint as u pertaining to cycles C_i and C_j. The above property holds for any two cycles C_i and C_j.

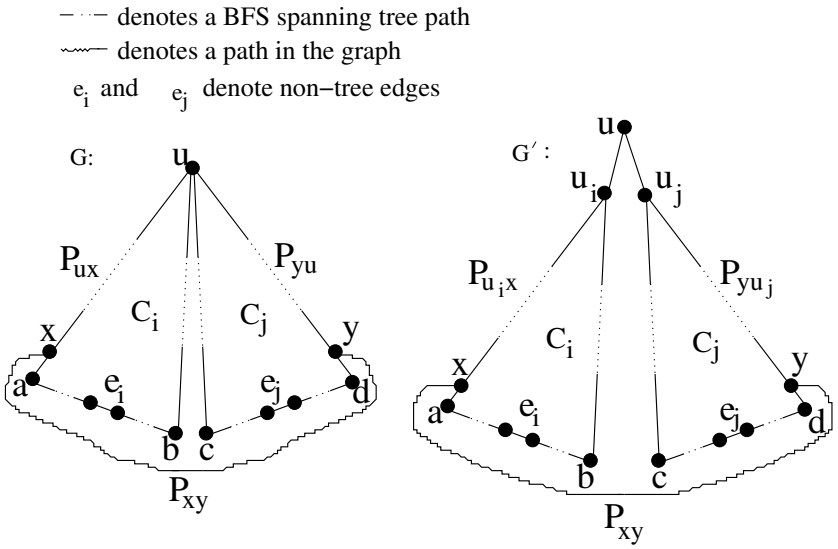

Figure 2: Figure shows the cycle created by paths P_{xy}, P_{yu_j}, $P_{u_ju_i}$, and P_{u_ix}. For ease of exposition, the auxiliary graph shown contains only the changes made with respect to u and not the changes induced with respect to other vertices.

- u is the LCA of some non-tree edge that induces cycle C_i in $\mathcal{C}$: Consider the case where the number of cycles through u is at least 2. By our assumption, u is not an articulation point. Let u_1 be the alias vertex of u. Hence, for some vertex x in C_i that is not equal to u, and another vertex, say the parent of u, there is a path that does not go through u. This path along with edges uu_1 and $uP(u)$ mean that the edge uu_1 is part of a cycle. Therefore, in G', the edge uu_1 will not be a bridge.

(**if**) $\Rightarrow$: Let u be an articulation point of a 2-edge-connected graph G and let G' be the corresponding auxiliary graph. It holds that u has at least two cycles passing through it and not sharing any of the base vertices. Let b_i and b_j be one of the base vertices on two fundamental cycles. Let x and y be two vertices, both distinct from u, on two such cycles C_i and C_j that get disconnected by the removal of u. Since u is an articulation point in G, there exists only one path between x and y that passes through u, say, $x - b_i$, $b_i - u$, $u - b_j$ and $b_j - y$. Let u_1 and u_2 be any two alias points created for C_i and C_j. The corresponding path $P_{xy}(G')$ has the form $x - b_1$, $b_1 - u_1$, $u_1 - u$, $u - u_2$, $u_2 - b_2$, $b_2 - y$. Since u is an articulation point, there cannot be a path P_{xy} between x and y in G (and in its corresponding auxiliary graph G') that does not pass through u and its alias vertices. As a result uu_1 remains uncovered in any cycle of G'. Hence, uu_1 will be a bridge in G'.
□

Lemma 3 *Let G be a 2-edge-connected graph, T_G a rooted BFS tree of G with root as r, and G' be the auxiliary graph of G constructed as earlier. Vertex r is an articulation point in G iff r is the LCA of more than one non-tree edge of G' according to a BFS in G' from r, and r is also an end point of some bridge in G'.*

Proof: We use $P_{uv}(G)$ to denote a path between vertices u and v in the graph G.

(**only − if**) $\Leftarrow$**:** Notice that since G is 2-edge-connected, vertex r is on at least one cycle. Further, since r is the root of the BFS tree of G, for every fundamental cycle that contains r, vertex r is the LCA of the non-tree edge that induces the cycle. We now make a case distinction as follows. Now consider the case that more than one cycle passes through r. Let C_i and C_j be any two cycles through r induced by non-tree edges e_i and e_j. In G', we now introduce two alias vertices r_i and r_j

and also the edges rr_i and rr_j, along with edges between r_i and r_j to the base vertices of C_i and C_j. If r is not an articulation point, then we notice that there are any two distinct vertices x and y in C_i and C_j respectively such that there is a path between x and y that does not go through r. This path between x and y, P_{xy}, along with paths P_{xr_i}, the edges r_ir and rr_j, and path P_{r_jy} creates a cycle that contains the edges rr_i and rr_j. Therefore, the edges rr_i and rr_j cannot be bridges in G'.

(**if**) $\Rightarrow$: The same argument from proof of (**if**) part of Lemma 2 holds true when u is an articulation point and is the root of the spanning tree.
□

Thus, bridges in the auxiliary graph are a good indicator of articulation points in the original 2-edge-connected graph. Further, it is relatively easy to find the biconnected components of a graph when the bridges and articulation points are identified. Our approach also indicates that the size of auxiliary graph in terms of both the number of vertices and the number of edges, is more than the size of original graph. But, we will see later that for real-world graphs that are sparse in nature, this increase in size is usually small, and the additional increase in run time can be offset by the simplicity of our algorithm.

4 Our Algorithm for Biconnected Components

Given Lemmata 1, 2 and 3 we now provide the following algorithm to identify the biconnected components of a graph G. Algorithm LCA-BiCC shown as Algorithm 2 describes our approach in a high level as consisting of 5 steps. In Step I, we obtain a BFS tree of the input graph G. In Step II, we find the bridges of G and also decompose G into its 2-edge-connected components $G_1, G_2, \cdots$. Step III onwards, each such 2-edge connected components is treated independently. In step III, for each G_i, an auxiliary graph G'_i is constructed. Step IV identifies non-tree edges of each G'_i, and Step V identifies the bridges of the auxiliary graph, and hence the articulation points of G. In Step VI, the bridges and the articulation points of G are used to identify the biconnected components of G.

Algorithm 2 LCA-BiCC(G)

```
1:  procedure LCA-BiCC(Graph G)
2:      T ← BFS(G)  /*Step I*/
3:      {G_1, G_2, ···} = Bridges(G, T) /* Step II*/
4:      for all  G_i i = 1, 2, ··· in parallel do
5:          Construct the auxiliary graph G'_i./*Step III*/
6:          Identify the non-tree edges in G'_i among the newly
            added edges to G_i from Step III/*Step IV*/
7:          {H_1, H_2, ···} = Bridges(G'_i, T'_i)/*Step V*/
8:          Check if r_i is an articulation point in G'_i
9:          Run a connected components algorithm on (G'_i \ Bridges(G'_i, T'_i)) to identify
            the Biconnected Components of G_i/*Step VI*/
10:     end for
11: end procedure
```

The correctness of Algorithm 2 is immediate from Lemmata 1,2, and 3. An example run of the algorithm is presented in Figure 3. In the following, we elaborate on each step of Algorithm 2.

4.1 Step I: BFS on input graph G

We choose an arbitrary vertex r as the source vertex and perform BFS from r and also root the BFS tree at r. The output of BFS is stored in two arrays namely $L(v)$ and $P(v)$, where $L(v)$ signifies the level of the vertex in the BFS spanning tree and $P(v)$ stores the parent of v in the corresponding BFS tree. For the root r, we set $P(r) = -1$ and $L(r) = 0$.

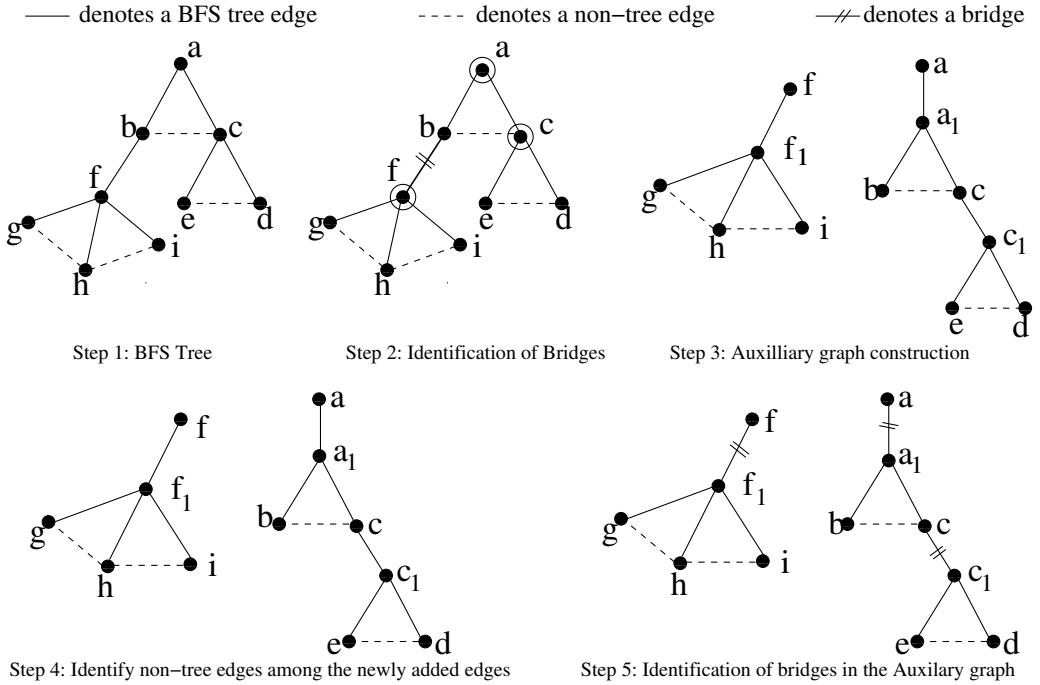

Figure 3: An example run of Algorithm LCA-BiCC. Graph G is BFS tree with non-tree edges (shown in step I). Vertices marked in a circle in step II are the LCA vertices.

4.2 Step II: Finding the Bridges of G

Recall that an edge in G is a bridge if and only if the edge is not on any cycle in G. The above property can be modified further to say that an edge is a bridge if it is not on any *fundamental cycle*, i.e., on cycles induced by non-tree edges according to a spanning tree. To this end, we consider each non-tree edge e according to the BFS tree T from Step I and mark all edges in T that are in the cycle induced by e. Algorithm 3 explains the above steps. As shown in Algorithm 3, for each non-tree edge $e = xy$, we traverse up the tree edges from x and y till we reach the LCA of x and y. Each edge encountered in this process is marked. Edges of T that are not marked in the above process are the bridges of G. Also, end points of these bridges with degree at least two are articulation points in G.

For each bridge xy identified above with $x = P(y)$, we set $P(y) = -1$ that essentially decomposes G into its 2-edge-connected components. We return these 2-edge-connected components as the output of Algorithm Bridges.

Algorithm 3 BRIDGES(G)

1: **procedure** BRIDGES(*Graph* G, *Tree* T)
2: **for all** $e = (w,v) \in G \setminus T$ **do**
3: Mark the tree edges that we encounter in the process of computing the $LCA(w, v)$.
4: **end for**
5: **for all** $e \in E(G)$ **do**
6: If e is not marked then $B \leftarrow B \cup \{e\}$
7: **end for**
8: Return the connected components of $G - B$
9: **end procedure**

We choose to find the LCA of the end points of a non-tree edge by using a traversal from these end points while more robust algorithms exist for computing the LCA. In the parallel setting, such algorithms are studied by Soman et al. [17] as an application of range minima queries. Our choice is however justified by two reasons. Firstly, most real-world graphs have a low diameter as Table 1 shows. As the number of traversals is upper bounded by the diameter, such traversals do not pose a serious performance bottleneck. Secondly, using range minima query to compute the LCA points involves non-trivial steps that can be computationally intensive compared to simple traversal.

4.3 Step III: Auxiliary Graph construction

Let $G_1, G_2, \cdots$, be the 2-edge-connected components of G. As described in Section 3, we create auxiliary graphs $G'_1, G'_2, \cdots$ corresponding to the 2-edge-connected components of G. (See also Figure 3 for an illustration.)

4.4 Step IV: Identify non-tree edges among the newly added edges

In this step, for each 2-edge-connected component G_i of G, $i \geq 1$, we do the following. We consider all the edges added to the auxiliary graph G'_i and mark them as either edges in the BFS tree or non-tree edges according to BFS. Notice that we do not have to run a BFS traversal again on G'_i, and can extend the BFS on G_i to identify the non-tree edges.

In particular, consider a vertex u which is the LCA of a non-tree edge e and vertices v and w are the children (neighbors) of u in the BFS tree T that are on the cycle induced by e. As part of the auxiliary graph construction, we add a vertex u', and edges uu', $u'v$, and $u'w$. We delete edges uv and uw. Such vertices v and w can now be end points of edges added during the construction of the auxiliary graph. Each such vertex v picks one of the alias vertices of its parent in the BFS of G'. The remaining edges between v and alias vertices will be marked as non-tree edges.

4.5 Step V: Identifying Articulation Points of G

In this phase we use the Algorithm 3 for identifying the bridges in each of the graphs $G^{'}_1.G'_2, \cdots$. For each such bridge $e = xy$, notice that one of the end points is a vertex that is not in G and is added to the corresponding auxiliary graph during Step III of the algorithm. Such vertices are the articulation points of G.

4.6 Step VI: Finding the Biconnected Components of G

In this step, we utilize the information of bridges identified in Step V to identify the biconnected components of G. Let F' denote the forest of auxiliary graphs generated after the removal of bridges identified on auxiliary graph G'. We invoke connected components algorithm on F' and rename each alias vertex in F' to its original to get the biconnected components of G.

Analysis: We analyze the work done in Algorithm LCA-BiCC as follows. BFS requires $O(n+m)$ work, and finding the bridges in both original and auxiliary graph requires $O(d_T)$ per non-tree edge. where d_T is the depth of the BFS tree. So the total work done for all non-tree edges is $O(md_T)$. Rest of the steps such as constructing the auxiliary graph, finding the connected components (Step VI) all can be done in time $O(m + d_T)$ time. Thus, the algorithm in the worst case takes $O(md_T)$ time in a sequential setting.

However, as observed in Table 2, the average number of traversals towards the root of the tree required in identifying bridges is often much smaller than the depth of the BFS tree. This is the reason why our algorithm is a nearly linear (in m and n) in time.

5 Implementation Details

In this section, we describe the implementation details of our algorithm. We also justify our choices made during the implementation.

We use the compact adjacency list representation(CSR), to represent the graph in the memory. In this representation all the adjacency lists are packed together into a single large array. An array E_a is used to store the adjacency lists where the list for vertex i+1 immediately follows that of vertex i for all vertices in G. An array V_a, stores the starting indices of the corresponding adjacency lists in E_a. CSR representation helps in reducing the irregularity memory access. For a detailed description, an interested reader can refer to [4].

We perform the following program optimizations while implementing Algorithm LCA-BiCC. The performance of Algorithm LCA-BiCC is influenced by factors such as the depth of the BFS tree produced in Step I, the time taken to identify vertices in V_{lca} in Step II, and the size of the auxiliary graph constructed in Step III. We use the heuristic based approach of selecting the largest degree vertex as the source of the BFS to minimize the depth of the BFS spanning tree.

To minimize the time taken to identify the LCA vertices, we introduce the following optimization. Consider a non-tree edge, $e = (u, v)$ with both u and v performing a walk towards the root of the tree. If *both* u and v encounter a tree edge that is marked by another non-tree edge, say f, then it holds that LCA(e) = LCA(f). Therefore, we stop the walks originating from u and v. These optimizations also reduce the size of the auxiliary graph.

We note that Lemmata 1, 2 and 3 can be modified suitably to work with any spanning tree. Using any spanning tree in Algorithm 3 requires one to associate level numbers to vertices in the tree. However, using a BFS tree allows to obtain the required level numbers as part of the BFS traversal and no additional computation is required for the same. Therefore, we chose to present the lemmata and the algorithms in terms of a BFS tree.

6 Experimental Results

6.1 Platform

We use an Intel i7 980x processor with 8 GB main memory as the experimental platform to test our results. The 980x is based on the Intel Westmere micro-architecture. This processor is from the Intel family with each core running at 3.4 GHz and with a thermal design power of 130 W. The i7-980X has six cores and with active SMT(hyper-threading) and can handle twelve logical threads. Beyond the memory, the i7-980x has a three level cache structure with an L1 cache per core of 64 KB split into two halves for instruction and data, an L2 cache per core of size 256 KB, and a shared L3 cache of size 12 MB.

6.2 Datasets

We experiment on a variety of real-world and synthetic datasets. For simplicity purposes, directed edges were considered undirected. Multiple edges and self loops were removed. The input graphs are assumed to be a single connected component. If not they are made connected by adding explicit edges. We experiment with real world graphs from the dataset of University of Florida Sparse Matrix collection [2] and SNAP database [1]. These matrices can be converted to graphs naturally. A list of the instances we consider are given in Table 1. Synthetic graphs were generated with the GTGraph suite [3] using the default parameters.

6.3 Results on Real World Graphs

We study the results of our approach on the graphs mentioned in Table 1. We will demonstrate both the relative and absolute speedup compared to prior work on the same platform and analysis of our algorithm with respect to the graph computations involved in it.

Overall Improvement We consider the overall improvement obtained by our approach compared to the best known implementation for finding the biconnected components on the same platform. In Figure 4, the baseline we use for comparison is the *best* runtime achieved by either of Algorithm BFS-BiCC and Algorithm Color-BiCC reported in [15].

Graph name	Source	$\|V\|$	$\|E\|$	Diameter
web-google	[11]	916,428	5,105,039	23
Webbase	[22]	1,000,005	3,105,536	27
Roadnet-PA	[11]	1,090,920	3,083,796	794
Roadnet-CA	[11]	1,971,281	5,533,214	863
web-Stanford	[10]	281,903	2,312,497	745
wb-edu	[11]	9,845,725	57,156,537	511
amazon	[11]	262,111	1,234,877	29
Great-Britan	[11]	7,733,822	16,313,034	9340
asia-osm	[11]	11,950,757	25,423,206	48,126
Patents	[11]	3,774,768	16,518,948	29

Table 1: List of graphs that we use in our experiments.

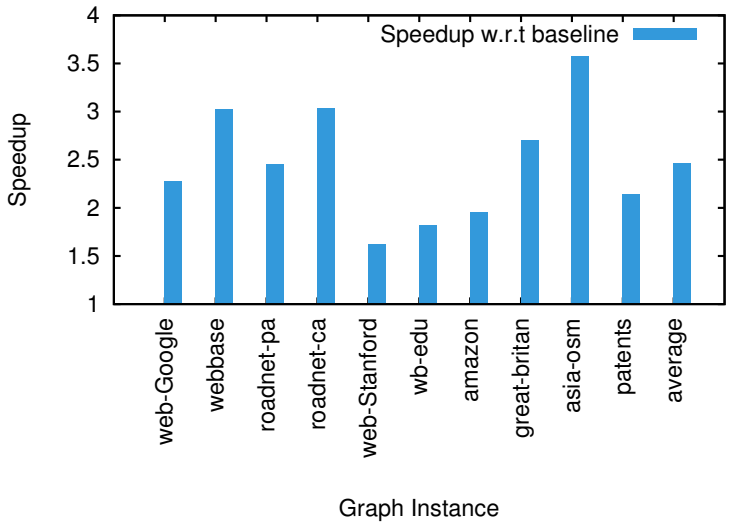

Figure 4: Comparing the overall performance improvement of our approach with respect to a baseline implementation. The Y-axis shows the ratio of the time taken by the baseline to our implementation.

As can be observed from Figure 4, Algorithm LCA-BiCC outperforms the baseline by a factor of 2.45x on average. As claimed in [15], the best of the above two algorithms is an improvement over the results of [7]. So, we expect that Algorithm LCA-BiCC too would outperform the results of [7].

Figure 5 shows the absolute times taken for all the three algorithms on the real world graphs. It is visible from Figure 5 that the run time of Algorithm BFS-BiCC and Algorithm Color-BiCC can vary hugely across instances and it is not possible to determine a-priori which algorithm might perform better among the two as both are heavily dependent on the structure of the graph.

Understanding the Results One can analyze the improvement is in terms of the basic steps in each of the algorithms under comparison. Algorithm BFS-BiCC [15] does, in principle, n BFS computations. There are however several optimizations that Slota and Madduri [15] introduced to ensure that several of these BFS computations do not visit all the vertices. On the other hand, Algorithm LCA-BiCC performs only one BFS computation (Step I) and one connected components computation (Step VI), apart from two calls to Algorithm Bridges (Steps II and V) to find the bridges of a given graph. The rest of the computation is to construct the auxiliary graph G'. During Algorithm Bridges, the end points of each non-tree edge march up the tree using the parent pointers. As described before, the number of traversals for each end point of a non-tree edge is upper bounded by the depth of the BFS tree produced in Step I. The depth of a BFS tree is also related to the

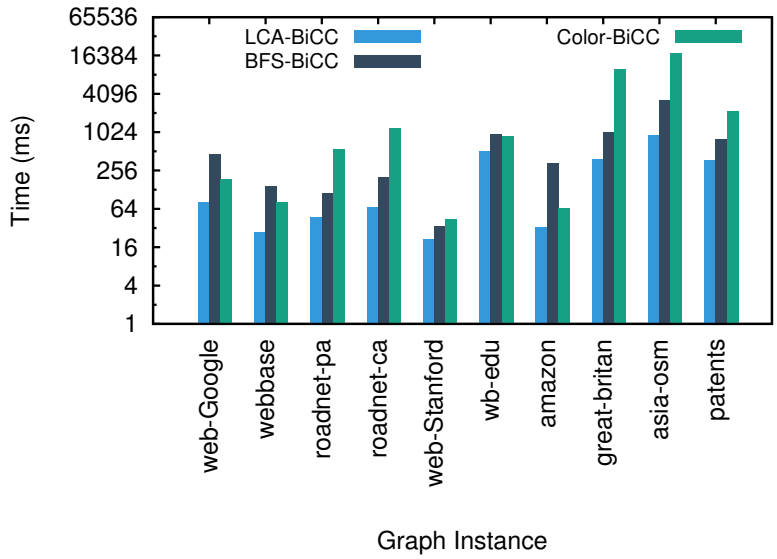

Figure 5: Figure shows the absolute runtime of Algorithm LCA-BiCC and Algorithms BFS-BiCC, Algorithms Color-BiCC from [15] (in left-to-right order of bars) on graphs listed in Table 1. Note that the Y-axis is in log-scale.

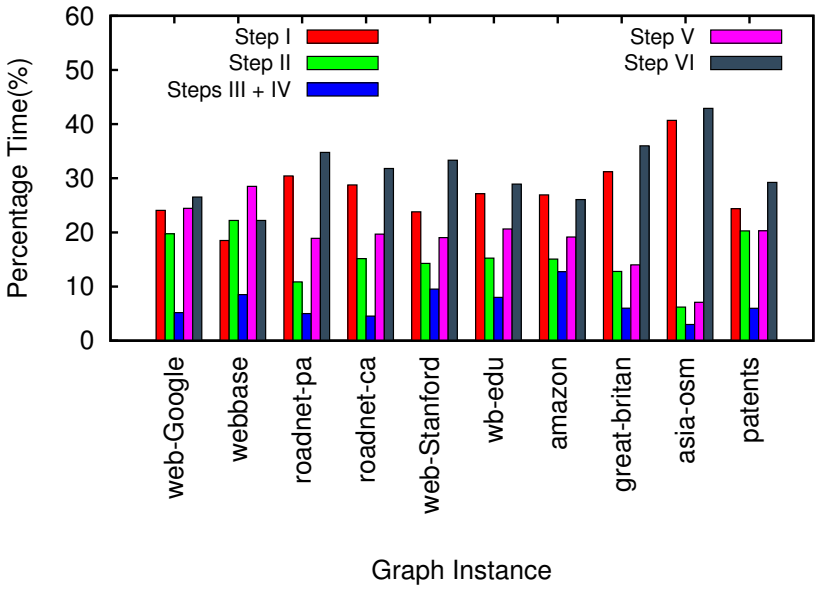

Figure 6: Profile of time taken across various steps of Algorithm LCA-BiCC.

diameter of the graph, and it is observed that most real-world graphs have a low diameter. The diameter of the graphs used in our experiments is also shown in Table 1.

Further, we also computed the average number of traversals needed by each end point to locate the LCA. The average number of traversals along with the depth of the BFS tree constructed in Step I are listed in Table 2 for the graphs from Table 1. It can be noticed that the average number of traversals needed is smaller than the depth of the BFS tree and is under 10 in all the graphs. This small number of traversals on sparse graphs is what also helps in keeping the runtime of our algorithm low.

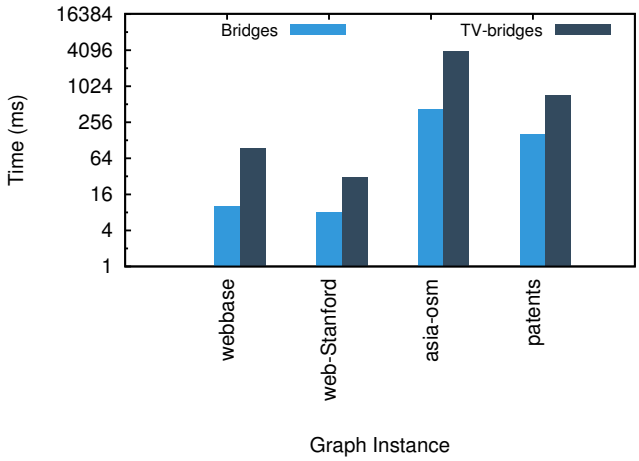

Figure 7: Figure shows the time taken to identify bridges in a graph G using our algorithm (Algorithm 3 and the algorithm of Tarjan and Vishkin [19]).

We study the size of the auxiliary graph constructed in Step III of Algorithm LCA-BiCC. Since an auxiliary graph is constructed for each 2-edge-connected component of G, we measure the sum of the sizes of the auxiliary graphs constructed with respect to each 2-edge-connected component of G. It is noted that the maximum increase in number of vertices at 25% and 20% occurred on the graphs roadnet-CA and roadnet-PA respectively. For all the other graphs from Table 1, the number of vertices increased by are under 5%. For each vertex added to the auxiliary graph, there are two edges removed and three edges that are added to the auxiliary graph. Thus, the increase in the number of edges is equal to the number of newly created vertices. In terms of relative increase, the number of edges had a maximum increase on the above instances again, at 8% and 7% respectively, and in all other instances from Table 1, the increase in the number of edges is under 3%. This shows that the actual increase in the size of the auxiliary graph is rather marginal and does not affect the performance of Algorithm LCA-BiCC in a significant manner.

Profiling and the Choice of the Source Vertex We finally show the time taken in each step of Algorithm LCA-BiCC as a percentage of overall time. Such a study shows the relative cost of each of the steps of the algorithm. The results of study are shown in Figure 6. Some of the improvement of Algorithm LCA-BiCC can be attributed to the fact that finding the bridges of a graph is a much easier task in a parallel setting. As shown in Figure 6, this step on the original graph G takes under 16% of the overall time on average, and takes under 20% on average on the auxiliary graph G'. These are labeled as Steps II and V respectively in Figure 6.

Note that Tarjan and Vishkin [19] also outline an approach to identify the bridges, their algorithm is usually more time consuming as it involves computation of functions low(v) and high(v) (as defined in Section2.1). Using the two functions low and high, a tree edge in T from $v \rightarrow w$ is marked as a bridge if and only if $w \leq low(w)$ and $high(w) \leq w + nd(w) - 1$ where $nd(w)$ refers to the number of descendants for a vertex w. To illustrate the simplicity of our approach for identifying bridges on sparse graphs, we consider some graphs from Table 1 and perform identification of bridges using our approach (Algorithm 3) and also the approach of Tarjan and Vishkin [19]. The time taken for both these approaches is plotted in Figure 7. As can be seen, our approach outperforms that of Tarjan and Vishkin.

In practice we have observed that unlike BFS-BiCC and Color-BiCC our LCA-BiCC approach is not heavily impacted by the choice of the source vertex while constructing the BFS spanning tree. The difference in the run time of Algorithm LCA-BiCC when a vertex of the highest degree is chosen as the source vertex versus an arbitrary source vertex is usually under 10% on the graphs from Table 1. On the other hand, as reported in [15], and also as witnessed in our experiments too, Algorithm

BFS-BiCC from [15] is significantly affected by the choice of the source vertex for all graphs listed in Table 1 and those considered in [15].

Graph name	%of LCA vertices	BFS depth	avg # LCA-traversals
web-google	3.31	12	1.09
Webbase	0.31	16	2.5
Roadnet-PA	15.02	534	9.91
Roadnet-CA	14.90	554	9.09
web-Stanford	1.90	727	2.82
wb-edu	11.91	319	9.81
amazon	8.20	23	6.47
Great-Britan	2.96	6841	7.55
asia-osm	2.93	38,793	4.93
Patents	1.97	17	3.91

Table 2: List of graphs along with the average number of traversals made per non-tree edge. The column labeled "% of LCA vertices" indicates the number of vertices that are in V_{lca} as a percentage of the total number of vertices.

6.4 Results on synthetic graphs

Figure 8 shows the speedup obtained by Algorithm LCA-BiCC on synthetic graphs. These graphs were generated using GTGraph suite [3] with default parameters. In Figure 8, graph rand_pM_qM refers to a random graph generated with p million vertices and q million edges. We obtained an average speedup of 2.53x with respect to the best of BFS-BiCC and Color-BiCC [15].

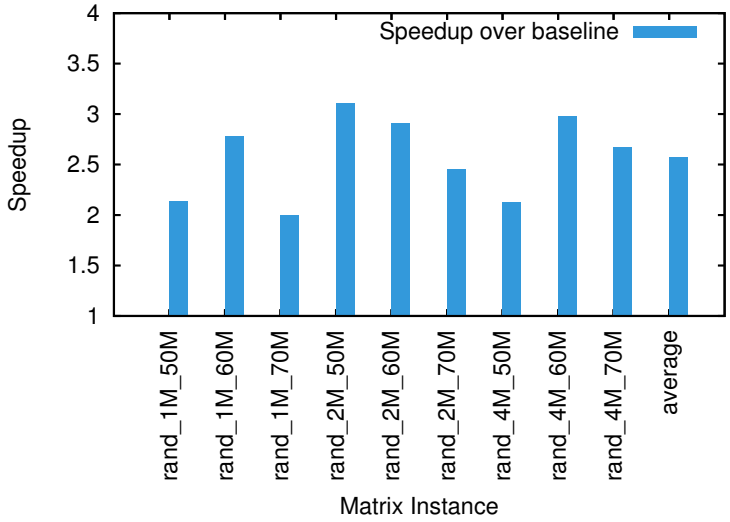

Figure 8: Comparing the overall performance improvement of our approach with respect to best of BFS-BiCC and Color-BiCC. The Y-axis shows the ratio of the time taken by the baseline to LCA-BiCC algorithm.

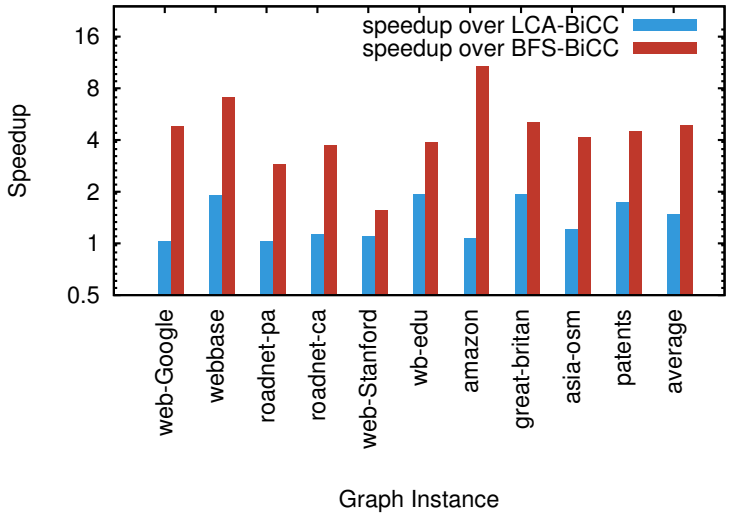

Figure 9: Speedup obtained by Algorithm LCA-BFS-BiCC over Algorithm LCA-BiCC and Algorithm BFS-BiCC ([15]), for the graphs from Table 1.

7 Further Improvements

In this section we present a simple yet efficient optimization for Algorithm BFS-BiCC [15] for identifying the biconnected components of graph G. Our improvements is based on two observations. Firstly, we make use of the observation from Section 1.1 that identifying the bridges in a graph is an efficient operation, especially in a parallel setting. Bridges also help in partitioning a graph into its 2-edge-connected components that can be processed independently. Secondly, given a 2-edge-connected graph, we use Lemma 1 to discard vertices that are certainly not articulation points and work with a small subset of potential articulation points. We call the vertices that are certainly not articulation points as *non-essential* vertices. Such a notation with respect to edges was also used by Cong and Bader[7].

We use the above two observations to modify Algorithm BFS-BiCC from [15]. Unlike Algorithm BFS-BiCC, we do not examine each vertex for its possibility to be an articulation point. The LCA-BFS-BiCC approach is limited only to potential articulation points. Rest of the details are similar to that of Algorithm BFS-BiCC. As can be seen from Table 2, since the percentage of LCA vertices is small in most real-world graphs, we expect that our modifications to Algorithm BFS-BiCC would result in an improvement.

Our algorithm, called Algorithm LCA-BFS-BiCC, is shown as Algorithm 4. Some of the steps such as computing a BFS tree (Step I), finding the bridges of G (Step II) are identical to that of Algorithm LCA-BiCC. In Step III, we limit the call to Procedure BFS-L described by Madduri et al. [15, Algorithm 6] to only vertices in $V_{\text{lca}}(G_i)$ for each i. In Step III, once we identify the articulation points of each G_i, we use a step similar to that Step VI of Algorithm LCA-BiCC to identify the biconnected components of G.

7.1 Experimental Results

We use the experimental platform as described above in Section 6. We compare the results of LCA-BiCC algorithm with optimized BFS-BiCC approach on graphs given in Table 1.

In Figure 9, we show the improvement obtained by Algorithm LCA-BFS-BiCC compared to Algorithm LCA-BiCC. On average, Algorithm LCA-BFS-BiCC is about 1.46x faster compared to Algorithm LCA-BiCC. Figure 9 also shows the improvement obtained by Algorithm LCA-BFS-BiCC over the BFS-BiCC algorithm of [15]. Central to the improvement obtained by Algorithm LCA-BFS-BiCC is the idea that we have to perform BFS from only vertices in V_{lca}.

Algorithm 4 LCA-BFS-BiCC(G)

```
1: procedure LCA-BFS-BiCC(Graph G)
2:     T ← BFS(G)  /*Step I*/
3:     {G_1, G_2, ··· ,} = Bridges(G, T) /*Step II*/
4:     for all G_i, i = 1, 2, ··· , in parallel do /*Step III*/
5:         for all v ∈ G_i do
6:             Articulation(v) = false; visited(v) = false
7:         end for
8:         for all u ∈ V_lca(G_i) \ r_i do /* r_i is the root of
9:                 the BFS tree of G_i */
10:            v ← P(u)
11:            if Articulation(v) = false then
12:                l ← BFS-L(G, L, v, u, visited) /*BFS-L refers to [15, Algorithm 6]*/
13:                if l ≥ L(v) then
14:                    Articulation(v) ← true
15:                end if
16:            end if
17:        end for
18:        Check if r_i is an articulation point
19:        Identify the Biconnected Components of G_i using the Articulation Points of G_i /*Step
   IV*/
20:    end for
21: end procedure
```

To understand the extent of improvement of Algorithm LCA-BFS-BiCC over Algorithm BFS-BiCC, we note the following. From the column labeled "% LCA Vertices" of Table 2, we see that in general, real-world graphs have a small percentage of vertex that are in V_{lca}. So, we expect significant performance gain for Algorithm LCA-BFS-BiCC. However, Figure 9 indicates the performance gain of Algorithm LCA-BFS-BiCC does not match the corresponding expected gain. For instance, if a graph has 10% of vertices in V_{lca}, one can expect a 10x improvement in the run time of Algorithm LCA-BFS-BiCC compared to that of Algorithm BFS-BiCC of [15]. This is not the case in general for the following reasons.

Algorithm BFS-BiCC introduces optimizations such as truncating the BFS-like traversals that are deemed unnecessary, invalidating the vertices of an already established biconnected component, and the like. Algorithm LCA-BFS-BiCC also benefits from these optimizations. However these optimizations mean that in Algorithm BFS-BiCC, even though most BFS traversals are terminated early on, there is still redundant work that is removed by using Algorithm LCA-BFS-BiCC. For experimental purposes, when the above mentioned optimizations from BFS-BiCC are removed, we do notice that the speedup achieved by Algorithm LCA-BFS-BiCC compared to that of Algorithm BFS-BiCC is near the expected speedup.

Also in the BFS-BiCC approach, the choice of root vertex affects the runtime of the level-truncated BFS. Since the percentage of LCA-vertices (from Table 2) are quite low for real world graphs, it is observed during our empirical study that this choice would have minimum affect on the overall runtime of LCA-BFS-BiCC.

8 Our Approach for Dense Graphs

The algorithms described above might not work well for dense graphs mainly due to large volumes of computation being involved in processing the non-tree edges. One of the possible approaches is to convert a dense graph to a sparse graph by applying the Cong-Bader approach [7] described in Section 2.2. Their algorithm removes edges that are non-essential during the computation of biconnected components. The sparsified graph G^1 of G will have n vertices and at most $2n - 2$

edges (as it contains $n-1$ tree edges and at most $n-1$ non-tree edges). It can be clearly observed that for a given BFS spanning tree T, LCA vertices before and after the sparsification step remains unchanged.

Let us define the Sparsity Quotient of a graph to be the ratio of number of edges to its vertices. Empirically we noticed that when $n \geq 1M$ and as the Sparsity Quotient of the graph approaches 110, the Biconnected components computation gets benefited by applying this Cong-Bader sparsification technique. The algorithm is described below.

Algorithm 5 SPARSIFY-LCA-BFS-BICC(G)

```
1: procedure SPARSIFY-LCA-BFS-BICC(Graph G)
2:     T ← BFS(G)  /*Phase I*/
3:     F ← BFS(G\T)
4:     G^1 ← (F∪T)
5:     Biconnected_Components = LCA-BFS-BICC(G^1) /*Phase II*/
6:     for all  e = (w,v) ∈ G \ (F ∪ T) do  /*Phase III*/
7:         label e to be in Biconnected Component Containing w and p(w)
8:     end for
9: end procedure
```

In Phase I, sparsification is performed on the input graph. We initially filter out the non-essential edges as described in the Section 2.2. This pruning step reduces the LCA computation from $m-n+1$ edges to at most $n-1$ edges. On this sparsified graph, in Phase II, we invoke the Algorithm 4 (from the Section 7) which perform MultiBfs only on LCA vertices. Note that we do not require an additional construction of a spanning tree and can use the existing spanning tree constructed in the Phase I while invoking Algorithm 4. Finally in Phase III, for each non tree edge w v, we assign its label to the biconnected component containing the tree edge from w to $p(w)$ or from v to $p(v)$.

8.1 Experimental Results

We apply our approach on wide range of synthetic graphs ranging from 110 M edges to 130 M edges. These are generated by fixing the required structure of the graph. For instance if we need k biconnected components, we fix a value of n and generate k values, $n_1, n_2, ..., n_k$ such that $\sum_{i=1}^{k} n_1 + n_2 + ... + n_k = n$. We then generate k graphs according to Erdos-Reyni graph model [5]. The average degree of each graph is set to p_i, which is calculated based on the following equation from [13]. At this value of p_i, it is shown in [13] that the resulting graph will be a dense biconnected component with high probability.

$$p_i = \frac{1}{n_i}\Big((\log n_i) + \frac{\beta}{2}(\log \log n_i) + \alpha + o(\frac{1}{n_i})\Big) \tag{1}$$

Here α is some constant, β denotes the connectivity of the component and n_i is the number of vertices in the subgraph. Since we are dealing with biconnected components we set the value of β to be 2. We then connect the k graphs so that the resulting graph is connected and has k biconnected components.

We compare our results of Algorithm 5 to Algorithm 4. It can be observed from Figure 10 that our approach on average is 1.41x faster over Algorithm 4 for dense graphs. From Figure 11 it can be observed that majority of time is consumed on the two BFS operations performed for sparsifying the graph. We can say that this improvement for dense graphs is remarkable, given the fact that the sequential version has a linear time complexity involving very small hidden constants.

9 Conclusions

In this paper, we have introduced a novel shared memory parallel algorithm for identifying the biconnected components. Our experimental results on wide variety of real-world graphs and syn-

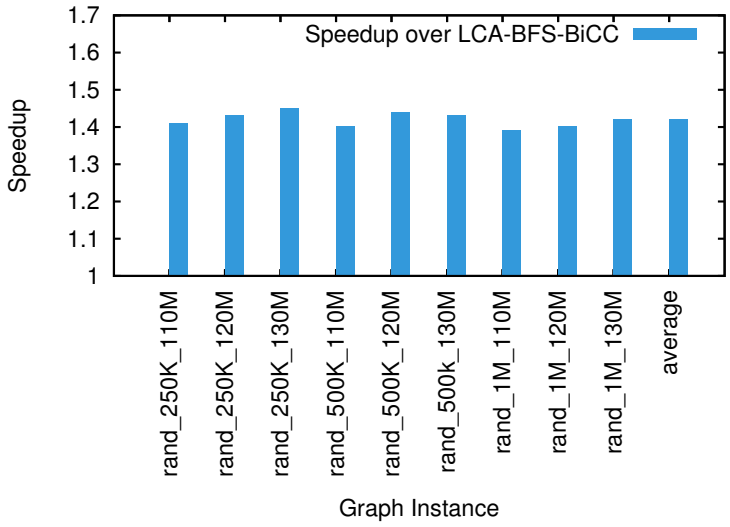

Figure 10: Speedup obtained by Algorithm Sparsified-LCA-BFS-BiCC over Algorithm LCA-BFS-BiCC.

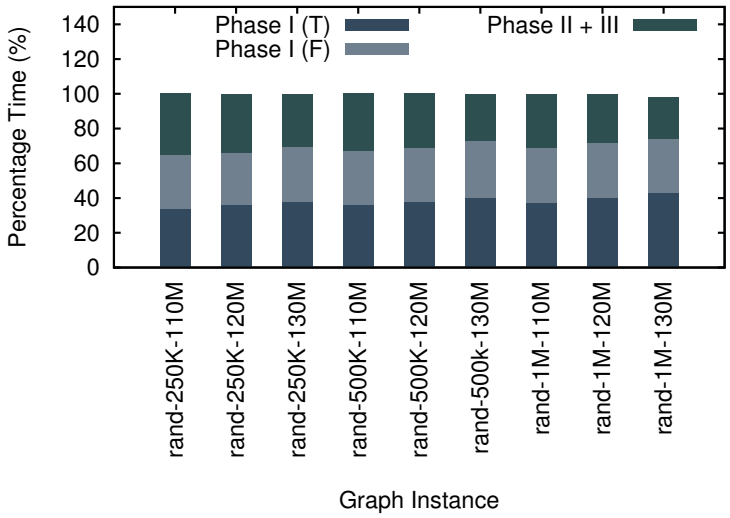

Figure 11: Profile of time taken across various steps of Algorithm 5. Phase I (T) refers to the percentage of time consumed for constructing the spanning tree. Phase I (F) refers to the percentage of time consumed for constructing the spanning forest and the sparsified graph.

thetic graphs indicate that our algorithm offers a considerable speedup over the current best known implementation for identifying the biconnected components on identical platforms.

References

[1] Stanford network analysis platform dataset. `http://www.cise.u.edu/research/sparse/matrices/{SNAP}`.

[2] The University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[3] Bader, D. A., and Madduri, K. GTgrpah: A suite of synthetic graph generators.

[4] Banerjee, D. S., Sharma, S., and Kothapalli, K. Work efficient parallel algorithms for large graph exploration. In *Proc. HiPC* (2013), pp. 433–442.

[5] Bollobas, B. *Random Graphs*. Cambridge University Press, 2001.

[6] Çatalyürek, Ü., Kaya, K., Sariyüce, A., and Saule, E. Shattering and compressing networks for betweenness centrality. In *Proc. SIAM Conf. on Data Mining* (2013), pp. 686–694.

[7] Cong, G., and Bader, D. A. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps). In *Proc. IEEE IPDPS* (2005).

[8] Eckstein, D. M. BFS and biconnectivity. In *Technical Report 7911, Dept. of Computer Science* (1979).

[9] Edwards, J. A., and Vishkin, U. Better speedups using simpler parallelprogramming for graph connectivity and biconnectivity. In *InProc. Intl. workshop on Programming Models and Applications for Multicores and Manycores (PMAA)* (2012), pp. 103–114.

[10] Leskovec, J., Kleinberg, J., and Faloutsos, C. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (ACM TKDD) 1*, 1 (2007).

[11] Leskovec, J., Lang, K., Dasgupta, A., and Mahoney, M. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. In *WWW08*. 2008, pp. 695–704.

[12] Ramachandran, V. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. Morgan-Kaufmann, 1993, pp. 275–340.

[13] Reif, J. H., and Spirakis, P. G. k-connectivity in random undirected graphs. *Discrete Mathematics* (1985), 181–191.

[14] Savage, C., and JáJá, J. Fast, efficient parallel algorithms for some graph problems. In *SIAM Journal on Computing* (1981), pp. 682–691.

[15] Slota, G. M., and Madduri, K. Simple parallel biconnectivity algorithms for multicore platforms. In *HiPC* (2014).

[16] Slota, G. M., Rajamanickaman, S., and Madduri, K. BFS and Coloring-based parallel algorithms for strongly connected components and related problems. In *Proc. of IPDPS* (2014), pp. 550–559.

[17] Soman, J., Kothapalli, K., and Narayanan, P. J. Discrete range searching primitive for the GPU and its applications. *J. Exp. Algorithmics 17*, 1 (2012).

[18] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM* (1972), 146–160.

[19] Tarjan, R., and Vishkin, U. An efficient parallel biconnectivity algorithm. *SIAM* (1985), 862–874.

[20] Tsin, Y. H., and Chin, F. Y. Efficient parallel algorithms for a class of graph theoretic problems. In *SIAM Journal on Computing* (1984), pp. 580–599.

[21] West, D. *Introduction to Graph Theory*, 2 ed. Pearson, 2000.

[22] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *In Proc. of ACM SC* (2007).