Expediting Parallel Graph Connectivity Algorithms

Mihir Wadwekar and Kishore Kothapalli
International Institute of Information Technology, Hyderabad,
Gachibowli, Hyderabad, India 500 032.
Email: {mihir.wadwekar@research., kkishore@}iiit.ac.in

***Abstract*—Finding whether a graph is k-connected, and the identification of its k-connected components is a fundamental problem in graph theory. For this reason, there have been several algorithms for this problem in both the sequential and parallel settings. Several recent sequential and parallel algorithms for k-connectivity rely on one or more breadth-first traversals of the input graph.**

While BFS can be made very efficient in a sequential setting, the same cannot be said in the case of parallel environments. A major factor in this difficulty is due to the inherent requirement to use a shared queue, balance work among multiple threads in every round, synchronization, and the like. Optimizing the execution of BFS on many current parallel architectures is therefore quite challenging. For this reason, it can be noticed that the time spent by the current parallel graph connectivity algorithms on BFS operations is usually a significant portion of their overall runtime.

In this paper, we study how one can, in the context of algorithms for graph connectivity, mitigate the practical inefficiency of relying on BFS operations in parallel. Our technique suggests that such algorithms may not require a BFS of the input graph but actually can work with a sparse spanning subgraph of the input graph. The incorrectness introduced by not using a BFS spanning tree can then be offset by further post-processing steps on suitably defined small auxiliary graphs. Our experiments on finding the 2, and 3-connectivity of graphs on Nvidia K40c GPUs improve the state-of-the-art on the corresponding problems by a factor 2.2x, and 2.1x respectively.

I. Introduction

A graph $G = (V, E)$ is said to be k-connected, for $k \geq 1$, if every pair of vertices in the graph have at least k vertex disjoint paths between them. The k-connected components of G are its maximal k-connected subgraphs. Finding whether a graph is k-connected and identifying its k-connected components is a fundamental problem with a variety of applications including planarity testing [17], isomorphism in planar graphs [20], network analytics [6], [15], [43], clustering [5] and data visualization [39].

It is therefore not surprising that several researchers have explored this problem in various settings such as sequential algorithms [8], [11], [19], [21], [35], parallel algorithms [18], [21], [22], [25], [30], [35], and implementations [7], [10], [32], [34], [37], and also distributed algorithms [27]. Most of these algorithms use graph traversal techniques to create one (or more) spanning tree(s) and use the properties of the spanning trees to test the k-connectivity of the graph and obtain its k connected components.

In particular, in the parallel setting, PRAM algorithms that require poly-logarithmic time and $O(m+n)$ work are known for $k = 1, 2, 3,$ and 4 [18], [21], [25], [30], [35]. However, in practice, the constants hidden in the big–O notation are significantly high for $k \geq 2$. Therefore, algorithms that run faster in practice are sought after. In a remarkable result, Cheriyan and Thurimella [8] showed that the k-connectivity of an undirected graph can be tested by using a $k \cdot (n-1)$ sized subgraph of the graph instead of using the entire graph. Formally, let T_i for $i \geq 1$ is the BFS spanning forest of $G \setminus \left(\cup_{j=1}^{i-1} T_j\right)$. Cheriyan and Thurimella show that the graph $H := \cup_{i=1}^{k} T_i$ is k-connected if and only if G is k-connected. The graph H is said to be a *certificate* for the k-connectivity of G. Similar results are also shown by Khuller and Scheiber [22].

The technique of Cheriyan and Thurimella [8] did improve the practical performance of parallel algorithms for testing the k-connectivity of an undirected graph. Evidence for this can be seen from the work of Bader and Cong [10], Chaitanya and Kothapalli [7], and that of Wadwekar and Kothapalli [37] for finding the biconnected components of a graph on symmetric multiprocessors, multi-core CPUs, and GPUs respectively. Much of this improvement can be attributed to the smaller size of the certificate in terms of the number of edges in the input graph.

However, in general, on large input graphs the time taken to obtain the certificate via parallel BFS operations can be a significant portion of the total run time. For instance, consider Algorithm N-GPU-BiCC from [37], which we rename as Algorithm Cert-GPU-BiCC in this paper. This algorithm is so far the fastest known implementation for finding the biconnected components of a graph in parallel. Algorithm Cert-GPU-BICC performs two BFS traversals on the graph G to obtain a certificate of size at most $2n-2$ edges for testing the biconnectivity of G. Figure 1 shows the time spent by Algorithm Cert-GPU-BiCC on BFS operations on a set of eight graphs while using the GPU BFS implementation of Merrill et al. [24]. (The GPU based BFS algorithm and its implementation proposed by Merrill et al. [24] is the best known so far and is also incorporated in frameworks for graph analytics on GPUs such as gunrock [44].) As shown in Figure 1, these two BFS operations consume on average 66% of the time spent by Algorithm Cert-GPU-BiCC. It indicates that to design faster parallel algorithms for graph k-connectivity, one must relook at the expensive BFS operations.

The large time spent by BFS operations can be attributed to the fact that a BFS traversal requires assigning vertices to levels such that for $i \geq 0$, the shortest hop distance from the source of the BFS to any node in level i is i. Arriving at such an assignment in parallel requires expensive

algorithmic/programming constructs such as synchronization, concurrent data structures, and work balancing among threads.

It must be noted that certificate based algorithms for graph k-connectivity are efficient only after obtaining the necessary certificate using k BFS traversals. The above situation inspires us to design parallel graph k-connectivity algorithms that mitigate the inefficiencies of BFS operations. Therefore, we suggest designing parallel algorithms that do not perform BFS operations on large graphs. One way of achieving this goal is to trade-off the cost of obtaining the certificate to its accuracy.

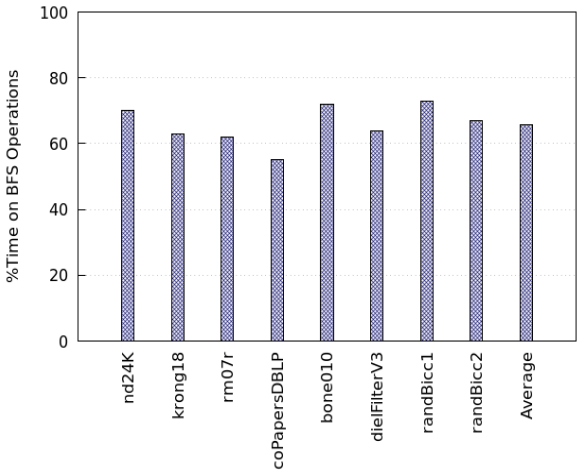

Fig. 1: Figure shows the percentage time spent by Algorithm Cert-GPU-BiCC (cf. [37]) on BFS operations.

In this paper, we show that by using novel strategies we can avoid performing BFS on the input graph G. Instead, we use a sparse spanning subgraph H' of the input graph. It must be noted that as H' may not be an accurate certificate for the k-connectivity of G, the k-connectivity of H' may not provide an answer to the k-connectivity of G immediately. To make up for this inaccuracy, we include additional steps on an auxiliary graph F created out of G and the k-connectivity information obtained from H'. The auxiliary graph F is constructed such that G is k-connected if and only if F is k-connected. The sizes of H' and F are usually smaller compared to that of G resulting in a low overall run time.

We implement our approach for testing the 2, and 3-connectivity and obtaining the 2, and 3-connected components of a graph on Nvidia GPUs. Our results are summarized in the following.

- For testing the 2-connectivity and obtaining the biconnected components of a graph our approach results in a speedup of 2.2x over [37] on a variety of real-world [40] and random graphs.
- We provide the first known GPU based algorithms for testing 3-connectivity of a graph and finding the 3-connected components of a graph.
- For testing 3-connectivity and obtaining the 3-connected components of a graph our approach results in a speedup of 2.1x over a corresponding certificate-based approach implemented in this paper.

Note that in the case of 3-connectivity, the certificate based approach of Cheriyan and Thurimella [8] involves three BFS traversals. So, one naturally expects a higher speedup for our approach on 3-connectivity. However that is not to be case as the graphs we use have a large number of 3-connected components resulting in more time spent in finding the 3-connectivity and the 3-connected components of F.

We believe that our technique has applications to other graph problems where one can algorithmically replace structures that are expensive to compute with simple to obtain and possibly inaccurate structures followed by a post-processing step. Our work therefore opens the possibility of reinterpreting important steps in parallel graph algorithms so as to make them more efficient in practice.

A. Related Work

Due to its varied applications, testing graph k-connectivity has been a problem of immense research interest. Early PRAM algorithms for testing the connectivity of a graph were proposed by Hirschberg et al. [18] and Shiloach and Vishkin [30]. The algorithm of Shiloach and Vishkin is shown to run in $O(\log n)$ time using $O(n + m)$ work in the PRAM model. Several experimental studies on finding the connected components of a graph are based on this algorithm [16], [31], [33]. In a recent work, Sutton et al. [34] argue that the Shiloach and Vishkin algorithm [30] can be applied on an $O(n)$ edge spanning subgraph of the input graph. The connected components of the subgraph can be used to find the connected components of the original graph by using the algorithm of Shiloach and Vishkin [30]. Our work in this paper seems to provide a good reason for the speedup achieved by Sutton et al. and extends their work for 2- and 3-connectivity.

The first PRAM algorithm for finding the 2-connected components of a graph in parallel is given by Tarjan and Vishkin [28]. This algorithm reduces the problem of finding the 2-connected components of a given graph to finding the connected components of an auxiliary graph. The construction of the auxiliary graph is shown to be in $O(\log n)$ time using $O(m+n)$ processors. It is identified by Bader and Cong [10] that the process of constructing the auxiliary graph is however quite slow in practice. Bader and Cong [10] proceed to use a formulation akin to that of Cheriyan and Thurimella [8] and show a speed-up of up to 2x on a variety of graphs on multi-core CPUs. More recently, Slota and Madduri [32] proposed that one can test the biconnectivity of a graph by performing multiple BFS traversals on multi-core CPUs. This result has been subsequently improved by Chaitanya and Kothapalli [7]. The approach of Chaitanya and Kothapalli [7] is then adapted to work on GPUs by Wadwekar and Kothapalli [37].

Ramachandran and Vishkin [28], and Miller and Ramachandran [25] present PRAM algorithms for finding the 3-connected components of a graph G. Their algorithms make use of the ear decomposition of a graph and define an auxiliary graph for every ear of G. These auxiliary graphs are then used to check the 3-connectivity of G followed by finding the 3-connected components of G. These algorithms [25], [28] can be recast to use the result of Cheriyan and Thurimella [8]. Vishkin and Edwards [13], [14] study parallel implementations

of 2- and 3-connectivity algorithms on the XMT architecture [36] and compare how these XMT implementations scale with increasing number of cores.

The inherent difficulty of efficiently performing a BFS traversal of a graph led to several researchers identifying numerous algorithmic and data structure optimizations on various modern architectures. Some of these include the direction optimizing BFS by Beamer et al. [2], cache- and data structure optimizations by Chhugani et al. [9] fine-grained task management based approach on GPUs by Merrill et al. [24], and graph decomposition based methods by Buluç et al. [4]. Some of these BFS implementations are now included in GPU based graph processing frameworks such as gunrock [44]. Despite these advances, we notice in our study that BFS traversals still consume a significant portion of the run time of parallel graph connectivity algorithms.

B. Organization of the Paper

The rest of the paper is organized as follows. Section II presents our technique in brief. Sections III and IV discuss our approach applied to the problem of 2, and 3-connectivity respectively. The paper then ends with concluding remarks in Section V.

II. An Overview of Our Approach

Several recent studies on parallel graph algorithms have explored varied techniques to improve their practical efficiency on multi-core and accelerator based architectures. Many such studies use well-known graph computations such as traversals, spanning trees, and edge/vertex decompositions as a subroutine. These algorithms can be summarized as follows. From the input graph G, one obtains a structural subgraph H such that computation on G can be translated or reduced to computations on H followed by additional post-processing steps as required. An example of this can be seen in the work of [12], [26] where a reduced graph that shrinks all vertices of degree two is used as the graph H.

The above mentioned approach of computing on a subgraph H often helps if H is of a smaller size than G. The benefits however will be limited if identifying H is expensive possibly due to strict structural guarantees required on H. As can be noticed from Figure 1, a large portion of time spent in obtaining H indicates scope for revisiting the approach.

In this direction, we propose to consider the algorithmic implication of replacing H with a suitable, easy to create structure H' such that the computation can be done on H' instead of H. In case the result of the computation on H' does not provide a correct result for the required computation on G, additional steps may be required depending on the nature of the problem. However, in these additional post-processing steps, the size of the problem is expected to be much smaller than the size of the original graph resulting in the cost of post-processing being small.

The approach can be seen to have three stages. In Stage I, we obtain a subgraph H' of G. Stage II performs computation on H'. An optional Stage III introduces a post-processing step, if required. The technique as presented above allows for multiple possibilities at all stages. In Stage I, H' can be obtained by (i) uniformly sampling the input graph G, (ii) by relaxing the structural properties required of H, (iii) using importance sampling, and the like. In Stage II, the computation on H' is chosen based on the input problem. Depending on the choices exercised in Stage I and Stage II, we consider the question of whether the output of Stage II can lead to the required output on the original graph. If the output of Stage II is insufficient to arrive at the final answer, we consider Stage III as the post-processing stage. In Stage III also, the computation required depends on the nature of the problem and the utility of H'. Stage III, depending on the problem can use possibilities such as iterating, and augmenting the result, and constructing an auxiliary graph for suitable computation.

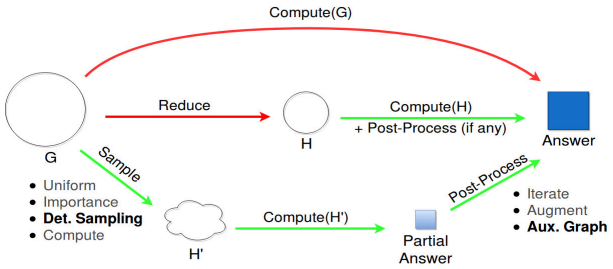

Fig. 2: Figure that illustrates our technique in comparison to other approaches towards practical parallel graph algorithms. The top path (colored red) represents direct computation that is usually expensive. The middle path indicates preprocessing via strict structural subgraphs or constructs that are sometimes expensive to create. The bottom path (colored green) corresponds to the less expensive approach proposed in this paper.

We note that as H' and H are expected to be of similar size, the time taken for computing on H and H' will not differ significantly. Hence, for our technique to be useful in practice, the cost of Stages I and III should be lesser than the cost of obtaining H from G. Figure 2 illustrates the idea of our approach. In Figure 2, we also list some of the possibilities at each stage of the approach. The particular choice used in this paper is shown in bold text in Figure 2.

In this paper, we apply our approach to test the k-connectivity and find the k-connected components of an undirected graph G for $k = 2$ and $k = 3$. Cheriyan and Thurimella [8] show that a subgraph H can be constructed as a certificate for G via k BFS traversals. (The k-connectivity of H offers a quick way to test the k-connectivity of G.) As obtaining H via multiple BFS traversals of the input graph can consume a significant portion of the overall run time (cf. Figure 1), we show that our approach can be helpful in arriving at faster parallel algorithms for graph k-connectivity.

III. Application to 2-Connectivity

Recall from graph theory that a graph G is said to be biconnected, or 2-connected, if every pair of vertices $v, w \in V(G)$ have at least two vertex disjoint paths between them.

The maximal 2-connected subgraphs of G are called as the biconnected components of G. A vertex v of G is called an *articulation point* if the removal of v from G disconnects G. An edge vw of G is called a *bridge* if removing vw from G disconnects G.

On GPUs, the only known algorithm for this problem is presented recently in [37]. Algorithm GPU-BiCC from [37] argues that in a parallel setting, finding the bridges of a graph G is much easier compared to finding the articulation points. Based on this observation, the algorithm first identifies the bridges of G and separates G into its 2-edge-connected components. (The 2-edge-connected components of G are its maximal subgraphs such that every pair of vertices in each subgraph have at least two edge disjoint paths between them). To identify the articulation points in each 2-edge-connected component G_i of G, Algorithm GPU-BiCC builds an auxiliary graph G'_i such that bridges of G'_i can be used to locate the articulation points of G_i, and hence those of G. This information is then used to subsequently identify the biconnected components of G. For further details, we refer the reader to [37]. Algorithm GPU-BiCC is 4x faster compared to other parallel approaches [7], [32]. On dense graphs, Algorithm Cert-GPU-BiCC from [37] uses the certificate as defined by Cheriyan and Thurimella [8] to obtain a further 2x speedup on Algorithm GPU-BiCC.

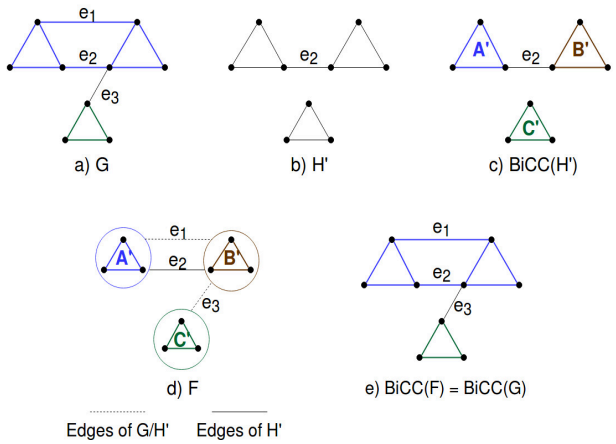

Fig. 3: An example run of Algorithm Sample-GPU-BiCC on the graph in part (a) of the figure.

Algorithm 1: Algorithm Sample-GPU-BiCC

Input: A connected graph G
Output: The Biconnected Components (BCCs) of G.
1 Obtain subgraph H' from G
2 Find the BCCs of H' using Algorithm GPU-BiCC.
3 Extract F using the BCCs of H' and edges of G
4 Find the BCCs of F using Algorithm GPU-BiCC.

A. Our Approach

As mentioned in the previous section, one can take H as the subgraph formed by taking the union of a BFS tree T of G and the BFS spanning tree of $G \setminus T$. This certificate H will have n vertices and at most $2(n-1)$ edges. However, as Figure 1 shows, obtaining H is an expensive step, taking an average of 66% of the total time of Algorithm Cert-GPU-BiCC. We therefore use our approach as outlined in Section II by replacing H with a suitable H'.

To this end, we start with H' as a kn sized spanning subgraph of G for an appropriately chosen constant k and proceed to find the biconnected components of H'. As H' may miss including certain edges critical to answer the biconnectivity of G, H' is not a certificate for biconnectivity of G. Nevertheless, the biconnected components of H' can be used to create an auxiliary graph F. Each vertex in F roughly corresponds to a biconnected component of H' and edges of F represent edges between these components. The edges of $G \setminus H'$ are used to add additional edges to F so that F acts as a valid certificate for the biconnectivity of G. As H' is of comparable size to H and the size of F is expected to be small, our approach can help reduce the time spent in BFS operations. More formal details of our approach are presented in the following.

1) Our Algorithm: Algorithm Sample-GPU-BiCC for finding the biconnected components (BCCs) of a connected graph G is listed as Algorithm 1. An example of Algorithm 1 is shown in Figure 3. Each of the steps of the algorithm are explained below.

- Step 1 – Obtain subgraph H' from G: Recall that H' is a kn sized subgraph of G. We identify H' by viewing the edges of G as an edge list and including every m/kn^{th} edge for a total of kn edges. Note that no randomness is used as any kn edges suffice for our purpose.
- Step 2 – Find BCCs of H': Once H' is obtained, we find the BCCs of H' using Algorithm GPU-BiCC from [37]. These BCCs are used to define the vertices of F.
- Step 3 – Extract F using BCCs of H' and the edges of G: In this step, we create an auxiliary graph F. The BCCs of H are treated as super-vertices that correspond to the vertices of F. Recall that a vertex can be part of several BCCs. In particular, articulation points belong to multiple BCCs. Hence we keep such vertices as a separate vertex in F. Two vertices in F are joined by an edge if there exists an edge $vw \in E(G)$ such that v and w are in different super-vertices of F.
  This results in F being a multi-graph. In such cases, since we need to know if there are at least two edges between nodes in F, we need to add at most two edges between any two vertices of F. For every pair of vertices v, w in F with two edges between them, we keep only one such edge between v and w, and add an auxiliary vertex v' and edges $vv', v'w$ to F. By doing so, F is now a simple graph.
  We note that as the edges of G are all used to define the edges of F, F acts as a certificate for the 2-connectivity of G. In other words, if vertices x and y have more than one vertex-disjoint path between them in G, then either x and y belong to the same super-vertex of F or the super-vertices of F that contain x and y have more than one vertex-disjoint path between them. The former happens when all the edges on at least one cycle containing x and

y is in H'. The latter happens when no cycle containing x and y is in H' in which case, the edges of the cycle(s) that are not in H' induce edges in F that ensure that the super-vertices of F containing x and y have at least two vertex-disjoint paths.

- Step 4 – Find BCCs of F: In this step, we find the BCCs of F using Algorithm GPU-BiCC [37]. The biconnected components identified in this step can be used to identify the biconnected components of G.

B. Implementation Details

We implement Algorithm Sample-GPU-BiCC on a GPU. For BFS on a GPU, we use the implementation from [24] that uses a fine-grained task management strategy. According to our approach, these BFS operations are done on subgraphs H' and F thereby requiring lesser time. This is followed by identifying the Least Common Ancestor (LCA) of the end points of every non-tree edge. Here, we launch one thread for every non-tree edge. Generating F requires a lookup of the edge list of G along with the information of BCCs of H'. This is easily implemented on a GPU by assigning a thread to every edge of G. We therefore note that Algorithm Sample-GPU-BiCC is amenable to a GPU-based execution where a massive thread pool is supported. We arrange the threads into blocks with 1024 threads per block.

TABLE I: Graphs used in our experiments. In the table, the letter K (*resp. M*) stands for a thousand (million).

Graph	Description	Nodes	Edges
Real-World Graphs [40]			
nd24k	3D Mesh, ND set.	72K	14.3M
kron18	kronecker, DIMACS10	262K	10.5M
rm07r	3D viscous case	381K	37.4M
coPaperDBLP	coauthor citation network	540K	15.2M
bone010	3D trabecular bone	986K	36.3M
dielFilterV3	High-order vector finite element method in EM	1.1M	45.2M
Random Graphs			
rand-Bicc1	1 BCC	1M	75M
rand-Bicc2	10000 BCCs	1M	75M

C. Experimental Results and Discussion

1) Experimental Platform: All our experiments are performed using an NVIDIA Tesla K40c GPU. This GPU is attached to an Intel i7-4790K CPU with 32GB RAM. The K40c has 2880 cores organized in 15 SMXs. The K40c provides 12 GB of GDDR5 ECC RAM with a maximum memory bandwidth of 288 GB/sec. Each core runs at a clock speed of 745 MHz. The K40c GPU supports a peak double precision floating point performance of 1.43 TFlops and a single precision floating point performance of 4.29 TFlops. Each SMX has a 64KB cache that is shared by the 192 cores of that SMX. A L2 cache of 1.5 MB is available across the SMXs. We use CUDA Version 7.5 [41] in our implementation.

2) Dataset: The graphs we use for our experiments are taken from real-world datasets [40] and random graphs following the Erdős-Rényi model [3] generated using the GTGraph generator [1]. All the graphs we consider are undirected and unweighted. Directed graphs are made undirected by removing the direction on the edge. Graphs that are not connected are augmented with additional edges to make them connected. Key properties of the graphs are shown in Table I.

3) Results: In this section we compare our implementation of Algorithm Sample-GPU-BiCC to that of Algorithm Cert-GPU-BiCC [37]. The overall improvement in performance on the graphs listed in Table I is shown in Figure 4. Algorithm Sample-GPU-BiCC achieves a speed-up ranging from 1.47x to 3.35x compared to Algorithm Cert-GPU-BiCC. The average speedup as shown in Figure 4 is 2.2x. All the above experiments were run with $k = 4$. The time spent by our approach on BFS operations is listed on the top of each bar. As can be noted, this time is on average only 15% of the total time indicating that our approach is successful in mitigating the practical inefficiency of BFS operations in the context of parallel graph biconnectivity algorithms. The graph nd24K has a high speedup of 3.35x as it is dense and is biconnected. Even a small sample of edges keeps almost all the nodes in a single BCC. For the graph coPaperDBLP, the lower than average speed-up can be attributed to its graph structure and the sampling strategy used. In this case, H' has several long chains of vertices of degree two. This increases the BFS time and the subsequent time for finding the BCCs of H'.

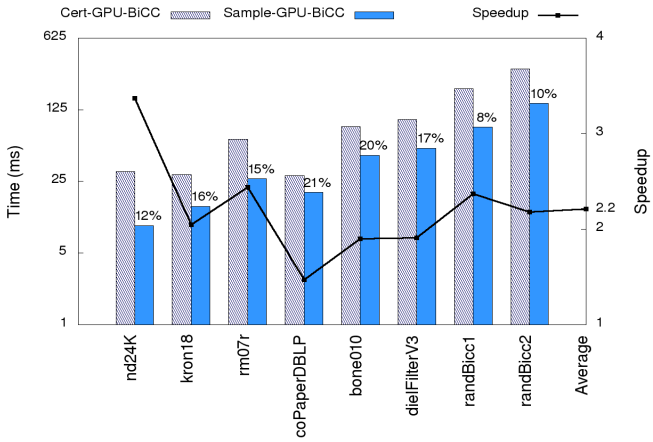

Fig. 4: Figure showing the time taken by Algorithms Cert-GPU-BiCC and Sample-GPU-BiCC on the graphs listed in Table I. Numbers on the right bars indicate the percentage of time spent by Algorithm Sample-GPU-BiCC in BFS operations. The Secondary Y-axis gives the speedup of Algorithm Sample-GPU-BiCC over Algorithm Cert-GPU-BiCC.

To study the impact of the choice of k on the performance of Algorithm Sample-GPU-BiCC, we plot time taken by our algorithm on two graphs from Table I as we vary k. The results of this experiment are shown in Figure 5 and 6 for graphs kron18 and coPaperDBLP respectively. When k is small, the size of H' is small. As a result, the time taken in Step 2 is small. However, if only few edges of G are included in H',

the number of biconnected components found in Step 2 tends to be high. Therefore, the size of F grows, resulting in Step 4 consuming more time. On the other hand if the value of k is high, the size of H' is high. As a result, the time taken in Step 2 is high. But, since more edges have been included in H', the size of F decreases thereby making Step 4 relatively faster. Steps 1 and 3 are not significantly impacted by the choice of k and hence omitted from Figures 5 and 6. In Figures 5 and 6, the vertical line at $k = 4$ indicates that the minimum for total time is achieved at $k = 4$.

Another factor to be noted is that the size of F, shown in Figures 5 and 6, indeed decreases as we increase k. This is in tune with our original motivation that doing a BFS on such a small graph will be significantly better than doing a BFS on G. The BFSs on smaller-sized H' and F combined are cheaper than a BFS on G. (Obtaining a certificate from G requires 2 BFSs on G, not one). The above discussion suggests that k should be chosen appropriately. We see from Figures 5 and 6 that a good value of k is around 4 while values of k between 3 to 5 offer a similar result in general.

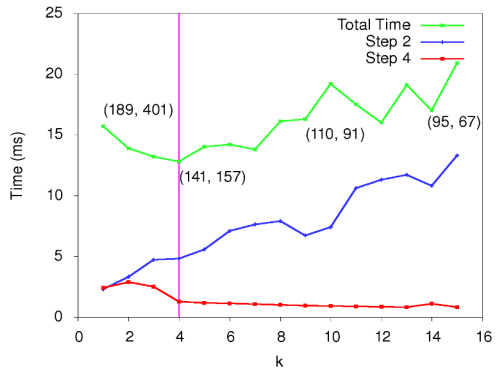

Fig. 5: Figure represents the time taken by Algorithm Sample-GPU-BiCC on the graph kron as k is varied. Tuples on the line labeled "Total Time" show the number of vertices and edges of F in thousands at $k = 1, 4, 9$, and 14.

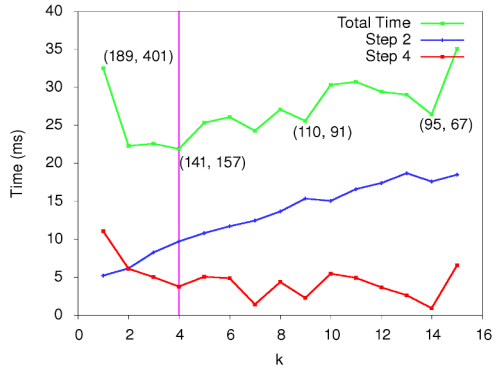

Fig. 6: Figure represents the time taken by Algorithm Sample-GPU-BiCC on the graph coPaperDBLP as k is varied. Tuples on the line labeled "Total Time" show the number of vertices and edges of F in thousands at $k = 1, 4, 9$, and 14.

4) Discussion: In this section, we summarize a few important points concerning our approach.

- Obtaining H': We observe that H' can be generated in several other ways such as a uniformly-at-random process on the edges, a selection based on degree of the vertices, and other such strategies. With a deterministic post-processing phase, we believe that one should focus more on trying to reduce the overall run time instead of getting a "good" H'. Hence we use deterministic sampling.
- Certificate based approaches: From the work of Bader and Cong [10] and also that of Cheriyan and Thurimella [8], it is apparent that using a certificate for testing the biconnectivity of a graph is practically efficient. In our approach, as the graphs H' and F are very sparse, such a certificate is not required and Algorithm GPU-BiCC is enough.

IV. Application to 3-Connectivity

A. Overview

Hopcroft and Tarjan [19] presented the first sequential algorithm for finding the triconnected components of a graph. The algorithm from [19] is based on the depth-first traversal (DFS) of a graph. Given that DFS is a P-complete problem [23], this approach would not be parallelizable in the PRAM sense. Over the years, few PRAM style parallel algorithms have been presented for finding the triconnected components of a graph. Ramachandran and Vishkin [28] present a PRAM algorithm for testing triconnectivity that runs in $O(\log n)$ time using $O(n + m)$ work. Miller and Ramachandran [25] extend the work from [28] to also obtain the triconnected components of a graph using $O(\log^2 n)$ time and $O(n+m)$ work on a CRCW PRAM. To date, the algorithm of Miller and Ramachandran is the fastest PRAM algorithm for identifying the triconnected components of a graph in parallel. In this work, we implement the algorithm of Miller and Ramachandran [25] on a GPU and also extend our approach from Section II for the graph triconnectivity problem. We start by briefly describing the algorithm of Miller and Ramachandran [25].

B. The Algorithm of Miller and Ramachandran for Graph Triconnectivity

The algorithm of Miller and Ramachandran [25] is based on an open ear decomposition of a graph. An open ear decomposition of a graph $G(V, E)$ is a partition of E into ordered edge-disjoint paths $P_0, P_1, P_2, ...$ such that P_0 is a simple cycle and every other path P_i, $i \geq 1$, has its endpoints on previous paths (ears) and no internal vertices of P_i lie on P_j, for $j < i$. Since a vertex cannot be internal to two ears, an open ear decomposition provides scope for traversing the graph in parallel. In addition, Miller and Ramachandran [25] prove that the two vertices from any separating pair [1] in a biconnected graph are non-adjacent vertices of some ear P_i.

[1] A *separating pair* in a graph G is a pair of vertices v, w such that removing v and w from G disconnects G.

As a result, Miller and Ramachandran start with an open ear decomposition of a biconnected graph. The algorithm then generates the bridges for every non-trivial ear in parallel. (An ear is non-trivial if it has at least three vertices.) For a given subgraph S, the bridges with respect to S is a partition of $V(G \setminus S)$ such that two vertices are in the same class if and only if there is path connecting them without using any vertex of S. For the graph in Figure 7(a) the bridge graph of ear P_1 is shown in Figure 7(b). Each such bridge is then compressed into a single vertex as indicated by vertices B_1 through B_5 in Figure 7(c). This single vertex is connected to the original ear through the same attachments as the corresponding bridge. This step is shown in Figure 7(c). This is done for every bridge for every non-trivial ear in parallel. The bridge graph is simplified into an ear graph by merging bridges which share the same attachments on an ear as shown in Figure 7(d).

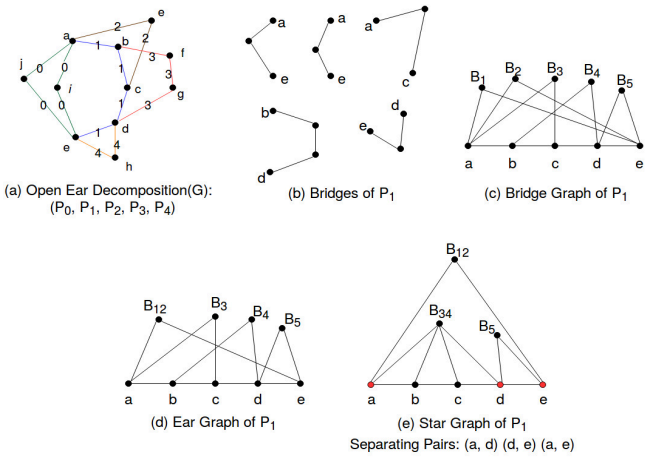

Fig. 7: Figure showing the stages in the algorithm of Miller and Ramachandran [25]. The numbers on edges in part (a) of the figure show the ear that the edge belongs to.

The ear graph for every ear is further simplified by merging interlacing bridges into an non-overlapping graph, called a star graph. All separating pairs can be easily discovered through a star graph. Figure 7(e) shows the formation of a star graph from the corresponding ear graph and the subsequent separating pairs with respect to a single ear. This process is done across the graph for all ears in parallel. Once the separating pairs are identified, the triconnected components are generated by splitting the graph into Tutte splits [42] for every separating pair. The entire algorithm is shown in run in $O\log^2 n)$ time using $O(n+m)$ work in the CRCW PRAM model. We refer the reader to [25] for further details.

C. Triconnectivity on GPU

To the best of our knowledge, there is no known GPU based algorithm for the graph triconnectivity problem. In this section, we provide a GPU based implementation for the algorithm of Miller and Ramachandran [25]. A brief summary of our implementation is given below. Henceforth, we refer to our GPU implementation for triconnectivity as Algorithm GPU-TriCC listed as Algorithm 2.

Algorithm 2: Algorithm for GPU-TriCC

Input: Biconnected graph G
Output: TriCC(G)

```
Find an open ear decomposition of G
for every nontrivial ear P_i do
    Construct the bridge graph from bridges
    Obtain the ear graph G_i(P_i) from the bridge graph
    Coalesce the interlaced ear graph into a star graph G*_i(P_i)
    Identify the separating pairs from G*_i(P_i)
Use Tutte splits to obtain the triconnected components
```

We employ Ramachandran's [29] popular ear decomposition algorithm for generating the open ear decomposition. Our ear decomposition requires two graph traversals and a sorting of the edge list. Obtaining bridges and the subsequent bridge graph requires a connected components algorithm. We use Soman et al. [33] GPU implementation for the same. The ear graphs are generated through a divide and conquer approach as mentioned in [25]. Assuming r ears, the first step in the divide and conquer approach generates the ear graph for the first and the last $r/2$ ears. Connected components of the i^{th} stage are utilized at the $(i+1)^{th}$ stage as we narrow down to generating the ear graph for every individual ear. Every ear graph is then coalesced in parallel to generate the star graph. Coalescing involves resolving all overlapping attachments in the ear graph. Separating pairs can be easily identified once the star graph is generated as shown in Figure 7(e). The graph is then split into upper split and lower split graphs for every separating pair (a, b) on the ear P_i. The upper split and lower split graphs are a division of vertices belonging to ears $P_j, j < i$ and ears $P_k, k > i$. Miller and Ramachandran [25] prove that each of the split is biconnected and every separating pair lies either in the upper split graph or in the lower split graph but not in both. Hence this procedure is applied recursively till no separating pair is present in either of the split graphs generated. Thus the triconnected components of G are identified.

Notice from the algorithm of [25] that the bulk of the work done can be associated with each ear subsequent to obtaining an open ear decomposition. As every graph G has $m-n+1$ ears, the number of edges in G heavily impacts the performance. Hence a reduction in the size of the graph through use of certificates provides scope for a better performance. To this end, we make use of the idea of Cheriyan and Thurimella [8]. Accordingly, a certificate for triconnectivity of G is obtained as the union of T, $F_1 = BFSSpanningForest(G/T)$ and $F_2 = BFSSpanningForest(G/(T \cup F_1))$, where T is a BFS tree of G. The graph $H := T \cup F_1 \cup F_2$ is then provided as the input to Algorithm GPU-TriCC. This modification is named as Algorithm Cert-GPU-TriCC and is listed as Algorithm 3.

Results: On a collection of real-world graphs listed in Table I along with the random graphs of Table I, we study the performance of Algorithms GPU-TriCC and Cert-GPU-TriCC. The random graphs are generated to have a particular number of TCCs as shown in Table II. The GPU used for these experiments is an NVIDIA K40c GPU (cf. Section III-C1).

Algorithm 3: Algorithm Cert-GPU-TriCC.

Input: Biconnected graph G
Output: $TriCC(G)$
1 $T := BFS(G)$
2 $F_1 :=$ BFSSpanningForest(G/T)
3 $F_2 :=$ BFSSpanningForest$(G/(T \cup F_1))$
4 $H := T \cup F_1 \cup F_2$
5 Run GPU-TriCC on H

From Figure 8, we can observe that Algorithm Cert-GPU-TriCC is on an average 4x faster compared to Algorithm GPU-TriCC.

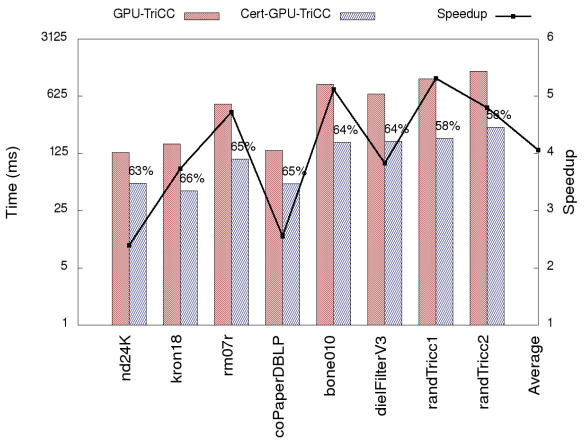

Fig. 8: Figure showing the time taken by Algorithms GPU-TriCC and Cert-GPU-TriCC on the graphs listed in Table II. Numbers on right bars indicates the percentage of time spent by Algorithm Cert-GPU-TriCC in BFS operations.

In Figure 8, we show the percentage of the time spent by Algorithm Cert-GPU-TriCC in obtaining the certificate H using three BFS traversals of G. As can be observed from Figure 8, Algorithm Cert-GPU-TriCC despite being 4x faster than Algorithm GPU-TriCC, spends nearly 63% of total time in obtaining H. The high cost of procuring the certificate leads us to try our approach from Section II for this problem.

D. Our Approach

As discussed and shown in the previous section, the three BFS traversals required to obtain a certificate H for testing the triconnectivity of a graph take up almost 63% of the total time. A suitable H' can reduce the cost of the three BFSs. We begin with a kn sized random subgraph H'. As in the case of biconnectivity, H' can miss out out on some critical edges required for triconnectivity of G. It is not a valid certificate yet. We then find the TCCs of H'. The TCCs are treated as super-vertices. These super-vertices form the vertex set of an auxiliary graph F. The edges of the rest of G are then used to define the edges of F. In order to keep the size of F small, the graph F is refined to ensure that at most three edges are present between any two vertices of F. Finally, the TCCs of F are identified which correspond to the TCCs of G. The algorithm is explained in-depth in the following.

1) Our Algorithm: Algorithm Sample-GPU-TriCC for finding the TCCs of a connected graph G is listed as Algorithm 4. Each of the steps are explained below.

Algorithm 4: Algorithm Sample-GPU-Tricc

Input: Biconnected Graph G
Output: TCCs of G
1 Obtain a spanning subgraph H' from G
2 Find the TCCs of H'
3 Extract F using the TCCs of H' and edges of G
4 Find the TCCs of F

- Step 1 – Obtain a spanning subgraph H' from G: As in the case of Algorithm Sample-GPU-BicC, we identify H' by viewing the edges of G as an edge list and including every m/knth edge for a total of kn edges. Note that no randomness is used as any kn edges suffice for our purpose.
- Step 2 – Find the TCCs of H': We find the TCCs of H' using Algorithm GPU-TriCC. Since H' is a sampled subgraph, it may not be biconnected. However, Algorithm GPU-TriCC only requires the ears and not their numbering. Therefore, we modify the ear decomposition algorithm of Ramachandran [29] to find an open ear decomposition within individual biconnected components. Due to this modification, the ears are identified correctly, though they may not be correctly numbered.
- Step 3 – Constructing F using the TCCs of H' and edges of G: The TCCs of H' are compressed into super-vertices. Since a vertex in a separating pair can belong to multiple triconnected components, vertices in each separating pair are treated as independent super-vertices. These super-vertices form the vertices of F. The edges of F are identified in three steps. First, an edge is added between two nodes of F if there exists an edge $vw \in E(G)$ such that v and w are part of different TCCs of H'. In second step, F is filtered to ensure that no more than three edges are present between any two vertices of F. This is done to keep the size of F as small as possible. In the third step, we convert F to be a simple graph. To this end, for every two nodes in F with more than one edge between them, we split each such edge by introducing auxiliary vertices.
  Similar to the arguments provided in Section III-A, we note that the graph F has the property that if vertices a, b, c of G have at least three vertex-disjoint paths between them in G, then either they belong to the same super-vertex of F, or the super-vertices of G containing these vertices have at least three vertex-disjoint paths between them in F. Therefore, F can be used to identify the triconnectivity and the triconnected components of G.
- Step 4 – Find the TCCs of F: We run Algorithm GPU-TriCC on F to generate the TCCs of F. These components can be used to identify the triconnected components of G.

E. Implementation Details

As can be noticed from [25], for the graph triconnectivity problem, some computations such as BFS and LCA traversals are common to the biconnectivity problem. In this case too, on the GPU, we therefore use the BFS implementation from [24]. Open ear decomposition is implemented through sorting and LCA traversals. Sorting is performed using Thrust library [38]. LCA traversals are done by assigning a thread to every non-tree edge. Generating the bridge graph for every ear involves finding the connected components of various appropriately defined subgraphs. For this purpose, we use the GPU based algorithm from Soman et al. [33]. Generating the star graph with respect to every ear and the subsequent identification of triconnected components can also be done on a GPU by expressing the computation as a sequence of multiple kernels.

F. Experimental Results, Analysis, and Discussion

The experimental platform we use for our experiments is described in Section III-C1. We scheduled the above algorithm on 1024 threads per blocks.

1) Dataset: We use two different datasets for our experiments. We use the real-world datasets [40] from Table I. In addition, we use random graphs generated according to the Erdos-Renyi model [3] that have a fixed number of TCCs. All the graph we consider are undirected and unweighted. Key properties of the graphs are shown in Table II.

TABLE II: Graphs used in our experiments for triconnectivity. In the table, K refers to a thousand and M refers to a million.

Graph	Nodes	Edges	Description
Real-World Graphs			
Same as in Table I			
Random Graphs			
rand-Tricc1	500K	30M	1 TCC
rand-Tricc2	500K	30M	10000 TCCs

2) Results: We compare the performance of Algorithm Sample-GPU-TriCC to that of Algorithm Cert-GPU-TriCC. As noted earlier, Algorithm Cert-GPU-TriCC is to the best of our knowledge, the fastest algorithm on GPUs for finding the triconnected components of a graph.

Figure 9 shows the time taken by Algorithm Sample-GPU-TriCC on the graphs listed in Table II. As can be observed, Algorithm Sample-GPU-TriCC achieves a speedup of 2.1x on average over Algorithm Cert-GPU-TriCC. Moreover, the percentage of time spent in BFS operations by Algorithm Sample-GPU-Tricc is now on average 17%. The value of k is set at 4 in this experiment.

We now proceed to study the performance of Algorithm Sample-GPU-TriCC as k is varied. On graphs rm07r and rand-Tricc1, Figures 10 and 11 respectively show the results of this study. As k increases, the size of H' increases resulting in increase in the time taken by Step 2. On the other hand, the decrease in the size of F with increasing k reduces the time taken in Step 4. The choice of k is to be made considering this trade-off. From our experiments, we note that $k = 4$ is a good choice in the case of triconnectivity. In Figures 10 and 11, the vertical line at $k = 4$ indicates that the minimum for total time is achieved at $k = 4$. However, values of k between 4 and 6 offer a near equal minimum.

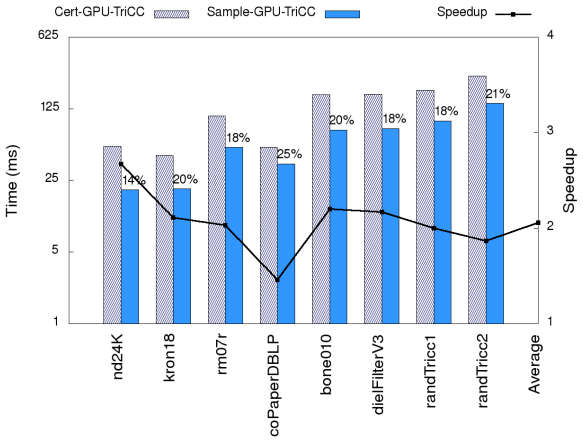

Fig. 9: Figure showing the time taken by Algorithms Cert-GPU-TriCC and Sample-GPU-TriCC on the graphs listed in Table II. Numbers on the right bar indicate the percentage of time spent by Algorithm Cert-GPU-TriCC in BFS operations. The secondary Y-axis gives the speedup of Algorithm Sample-GPU-TriCC over Algorithm Cert-GPU-TriCC.

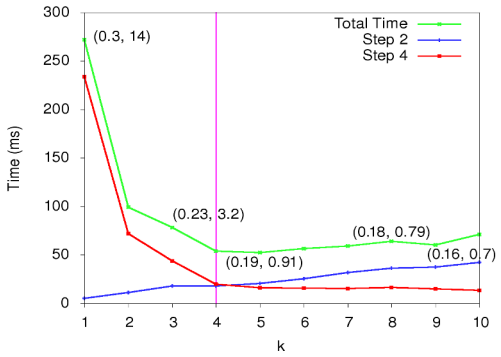

Fig. 10: Figure represents the time taken by Algorithm Sample-GPU-TriCC on the graph rm07r as k is varied. Tuples on the line labeled "Total Time" show the number of nodes and edges of F in millions at various values of k.

3) Discussion: One can observe in Figure 10 and Figure 11 or even in Figure 5 and Figure 6 (in Section III), that the time spent in Step 4 does not decrease significantly with increasing k. This is due to the fact that after some k, most of the biconnected/triconnected components of G are identified via H' alone.

In general, Algorithm Cert-GPU-TriCC involves more BFS operations than Algorithm Cert-GPU-BiCC. Thus, it seems that Algorithm Sample-GPU-TriCC should benefit more from our technique than Algorithm Sample-GPU-BiCC. However, as shown in Figure 4 and Figure 9, our technique results in a near similar speedup in both cases. This is due to the reason that for the graphs we considered in our dataset, and in general, we expect more triconnected components than biconnected

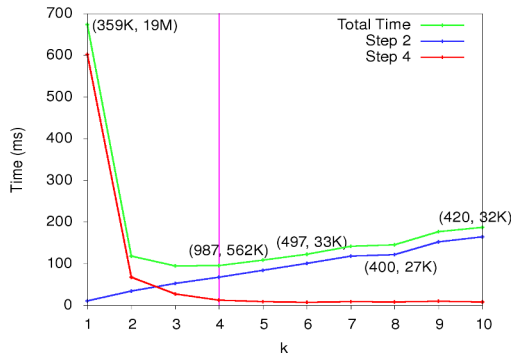

Fig. 11: Figure represents the time taken by Algorithm Sample-GPU-TriCC on the graph rand-TriCC1 as k is varied. Tuples on the line labeled "Total Time" show the number of nodes and edges of F at various values of k.

components. So, the size of the auxiliary graph F generated using our technique is larger in the case of triconnectivity as compared to biconnectivity.

V. Conclusions

In this paper, we studied how parallel graph connectivity algorithms can be improved by reducing the time spent in BFS operations. Our results indicate that a significant gain in performance can be obtained by reinterpreting algorithms to perform BFS on graphs that are much smaller in size compared to the input graph. We believe that our approach can be useful in other settings too. As our results show promise, in future, we want to also understand how to theoretically analyze the speedup that can be obtained using our approach.

References

[1] D. Bader, and K. Madduri, GTgraph: A suite of synthetic random graph generators. http://www.cse.psu.edu/~kxm85/software/GTgraph/

[2] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In Proc. ACM SC, pp. 12:1–12:10, 2012.

[3] B. Bollobás. Random Graphs, Cambridge University Press, 2011

[4] A. Buluç, S. Beamer, K. Madduri, K. Asanovic, D. Patterson, and D. Bader. Distributed-memory breadth-first search on massive graphs. in Parallel Graph Algorithms, Boca Raton, FL, USA:CRC Press, 2015.

[5] R. Butafogo, and B. Schneiderman. Identifying aggregates in hypertext structures, Proc. 3rd ACM Conference on Hypertext, pp. 63–74, 1991.

[6] U. Brandes, D. Wagner, M Junger, and P Mutzel. Visone - Analysis and Visualization of Social Networks, in Graph Drawing Software, Springer-Verlag, pp. 321-340, 2003.

[7] M. Chaitanya, and K. Kothapalli. Efficient Multicore Algorithms For Identifying Biconnected Components. IJNC 6(1): 87-106 (2016)

[8] J. Cheriyan, and R. Thurimella. Algorithms for Parallel k-Vertex Connectivity and Sparse Certificates. in Proc. ACM Symp. Th. Comp. (STOC), pp. 391–401, 1991.

[9] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In Proc. Intl Conf. on High Perf. Comp., Net., Storage and Anal. (SC). pp. 14:1–14:10, 2012.

[10] G. Cong, and D. A. Bader. An Experimental Study of Parallel Biconnected Components Algorithms on Symmetric Multiprocessors (SMPs). Proc. of IPDPS 2005.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. The MIT PRess, Third Edition, 2009.

[12] D. Dutta, M. Chaitanya, K. Kothapalli, and D. Bera. Applications of Ear Decomposition to Efficient Heterogeneous Algorithms for Shortest Path/Cycle Problems. in Proc. IPDPS Workshops, pp. 864–873, 2017.

[13] J. A. Edwards, and U. Vishkin. Better speedups using simpler parallel programming for graph connectivity and biconnectivity. in Proc. of Intl. Work. Prog. Mod. and App. for Multicores and Manycores (PMAM), pp. 103-114, 2012.

[14] J. A. Edwards, and U. Vishkin. Speedups for parallel graph triconnectivity. in Proc. ACM Symp. Par. Alg. and Arch. (SPAA), pp. 190-192, 2012.

[15] D. K. Goldenberg, P. Bihler, M. Cao, J. Fang, B. Anderson, A. S. Morse, and Y. Yang. Localization in sparse networks using sweeps, In Proc. Intl. Conf. Mobile comp. and Net., pp. 110 – 121, 2006.

[16] J. Greiner. A Comparison of Parallel Algorithms for Connected Components. Proc. ACM Symp. Par. Alg. and Arch. (SPAA), pp. 16–25, 1994.

[17] B. Haeupler, K. R. Jampani, and A. Lubiw. Testing Simultaneous Planarity When the Common Graph Is 2-Connected, in Algorithms and Computation. Proc. of ISAAC, 2010.

[18] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing Connected Components on Parallel Computers. Commun. ACM 22(8), pp. 461-464, 1979.

[19] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. SIAM J. Computing, 2(3), pp. 135–158, 1973.

[20] J. E. Hopcroft, and R. E. Tarjan. Isomorphism of Planar Graphs. In: Miller R.E., Complexity of Computer Computations. The IBM Research Symposia Series. Springer, pp. 131–152, 1972.

[21] A. Kanevsky, and V. Ramachandran. Improved algorithms for graph four-connectivity, in Journal of Computer and System Sciences Volume 42, pp. 288–306, 1991.

[22] S. Khuller, and B. Schieber. Efficient Parallel Algorithms for Testing Connectivity and Finding Disjoint $s-t$ Paths in Graphs. in Proc, of IEEE Symp. Found. Comp. Sci. (FOCS), pp. 288–293, 1989.

[23] J. JaJa. An Introduction To Parallel Algorithms, Addison-Wesley, 2004.

[24] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal, in ACM Transactions on Parallel Computing, Volume 1 Issue 2, 2015.

[25] G. L. Miller, and V. Ramachandran. A new graph triconnectivity algorithm and its parallelization. Combinatorica 12(1): 53-76 (1992)

[26] C. Pachorkar, M. Chaitanya, K. Kothapalli, and D. Bera. Efficient Parallel Ear Decomposition of Graphs with Application to Betweenness-Centrality. in Proc. of Intl. Conf. High Perf. Comp., pp. 301-310, 2016.

[27] G. Pandurangan, P. Robinson, and M. Scquizzato. Fast Distributed Algorithms for Connectivity and MST in Large Graphs, in Proc. ACM Symp. Par. Alg. and Arch. (SPAA), pp. 429–438, 2016.

[28] V. Ramachandran and U. Vishkin. Efficient parallel triconnectivity in logarithmic time,. In: Reif J.H. (eds) VLSI Algorithms and Architectures, Lecture Notes in Computer Science, vol 319, pp. 33–42, 1988.

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

[30] Y. Shiloach, and U. Vishkin. An $O(\log n)$ Parallel Connectivity Algorithm. J. Algorithms 3(1), pp. 57–67, 1982.

[31] J. Shun, L. Dhulipala, and G. E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In Proc. ACM Symp. Par. Alg. and Arch. (SPAA), pp. 143–153, 2014.

[32] G. M. Slota and K. Madduri. Simple parallel biconnectivity algorithms for multicore platforms. in Proc. of Intl. Conf. High Perf. Comp. (HiPC), pp. 1–10, 2014.

[33] J. Soman, K. Kothapalli, and P. J. Narayanan. Some GPU Algorithms for Graph Connected Components and Spanning Tree. Parallel Processing Letters 20(4): pp. 325–339, 2010.

[34] M. Sutton, T. Ben-Nun, and A. Barak. Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling, in Proc., of IEEE Intl. Par. Dist. Proc. Symp, pp. 12–21, 2018.

[35] R. E. Tarjan. Depth first search and linear graph algorithms. SIAM Journal on Computing, Vol. 1, pp. 146–160, 1972.

[36] U. Vishkin, S. Dascal, E. Berkovich and J. Nuzman. Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism, In Proc. 10th ACM Symp. Par. Alg. and Arch. (SPAA), pp. 140–151, 1998.

[37] M. Wadwekar and K. Kothapalli. A fast GPU algorithm for biconnected components, in Proc. of Intl. Conf. Cont. Comp., pp. 1–6, 2017.

[38] Thrust C++ library, https://developer.nvidia.com/thrust.

[39] SAS(R) OPTGRAPH, http://support.sas.com/documentation/cdl/en/procgralg/68145/PDF/default/procgralg.pdf

[40] SuiteSparse Matrix Collection. https://sparse.tamu.edu/

[41] NVIDIA Corporation, https://docs.nvidia.com/cuda/

[42] W. T. Tutte. Connectivity in Graphs, University of Toronto Press, 1966.

[43] F. Wang, M. T. Thai, and D. Z. Du. On the construction of 2-connected virtual backbone in wireless networks, in IEEE Transactions on Wireless Communications, pp. 1230–1237, 2009.

[44] Y. Wang, Y. Pan, A. D. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens. Gunrock: GPU graph analytics, in Proc. of ACM Trans. Parallel Computing, 4(1), pp. 3:1–3:49, 2017.