Scalable Biconnected Components on GPUs

Soumendra Nath Banerjee, Turja Kundu, and Sanjukta Bhowmick
University of North Texas
SoumendraBanerjee@my.unt.edu, turjakundu@my.unt.edu, Sanjukta.Bhowmick@unt.edu

Sriram Srinivasan
Bowie State University
sriram882009@gmail.com

I. Introduction

Graphs are powerful tools for studying complex systems, revealing structural and functional insights through graph-property analysis. Graph datasets continue to grow rapidly, with modern graphs containing billions of vertices and edges. Graph biconnectivity is one of the fundamental primitives in this space, and researchers have studied it extensively for decades. A biconnected component (BCC) is a maximal subgraph that remains connected even after removing any single vertex. Finding biconnected components has direct applications across domains such as social network analysis and VLSI circuit design [1].

A classical sequential solution, due to Hopcroft and Tarjan [2], computes biconnectivity via a depth-first search (DFS) traversal, running in $O(V + E)$ time. This approach performs well sequentially, but DFS is inherently sequential by construction (in fact, DFS is P-complete [3]), which limits its scalability when applied to accelerators or other parallel computing architectures.

Motivated by this limitation, Tarjan and Vishkin [4] proposed the first PRAM algorithm for biconnectivity. Rather than relying on DFS, their algorithm builds an *arbitrary* spanning tree one that can be constructed in parallel and characterizes biconnectivity through an auxiliary graph G' whose connected components correspond exactly to the biconnected components of G. A practical drawback of this formulation is that $|V(G')| = \Theta(m)$, since every edge of the original graph becomes a vertex in G', leading to substantial memory overhead on large graphs [5], [6].

Subsequent work has sought to reduce this overhead. Cong and Bader [6] introduced a filtering technique (TV-filter) that builds a sparser skeleton from a BFS tree together with a spanning forest of the non-tree edges, reducing the skeleton to $O(n)$ edges at the cost of $\Theta(\text{diam}(G) + \log n)$ depth, since BFS tree construction is inherently diameter-dependent. GPU-native approaches such as WK-17 [7] and WK-18 [8] similarly build a BFS tree and an auxiliary graph from fundamental cycles, mapping cut vertices of G to cut edges of the auxiliary graph. More recently, Dong et al. [9] (FAST-BCC) reduced the skeleton's space complexity by classifying tree edges into *fence* and *plain* categories, and Sahu et al. [5] (FAST-BCC-Filter) extended this line of work with a pruning technique that operates correctly on an arbitrary spanning tree rather than a BFS tree specifically, avoiding the BFS depth bottleneck altogether and further supporting out-of-memory execution for graphs exceeding GPU capacity. In this work, we propose a new algorithm for finding biconnected components in large-scale undirected graphs. Our approach uses a multi-phase pipeline that first identifies which edges genuinely require analysis for biconnectivity, before computing the biconnected components themselves.

In the first phase, we filter out 1-core vertices i.e. the vertices with degree at most one, and iteratively re-examine their affected neighbors, continuing until no 1-core vertices remain. This step alone has a substantial impact on real-world networks: on the `road-usa` network, we eliminate approximately 29.5% of vertices before any biconnectivity computation begins.

In the second phase, we select random seed vertices from the remaining (active) graph and grow a breadth-first search (BFS) tree from each seed simultaneously, in parallel. To keep the resulting trees load-balanced, we bound each tree by both a maximum depth and, implicitly, the region it can claim: once a vertex is claimed by one tree, no other tree may claim it. We repeat this process, seeding new trees for any vertex the initial wave did not reach, until every active vertex identified in the first phase belongs to exactly one tree.

Once the trees are established, we identify the *cross edges* i.e. the edges whose two endpoints fall in different trees, and, for each tree independently, examine its *non-tree edges* (edges within the same tree that are not part of the tree structure itself). For every such non-tree edge, we trace the path it closes back to a common ancestor; every tree edge along that path belongs to the same biconnected component as the non-tree edge itself. This yields a local BCC decomposition for each tree.

In the final phase, we process the cross edges identified earlier and determine which of these locally-computed BCCs must merge across tree boundaries, and which remain correctly separated by a genuine articulation point. Figure 1 illustrates our multi-phase pipeline, and Figure 2 walks through every phase on a toy network, from initial peeling through to the final BCC and articulation point result.

We evaluate our approach and compare it against existing algorithms, including [4], [5], [7]–[9]. We perform an extensive evaluation across 17 real-world datasets; our proposed approach achieves an average speedup of 2.49x over FAST-BCC-Filter [5], the current state-of-the-art GPU approach.

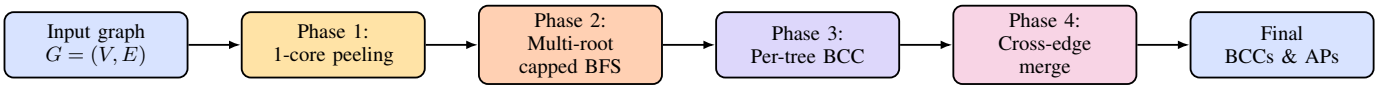
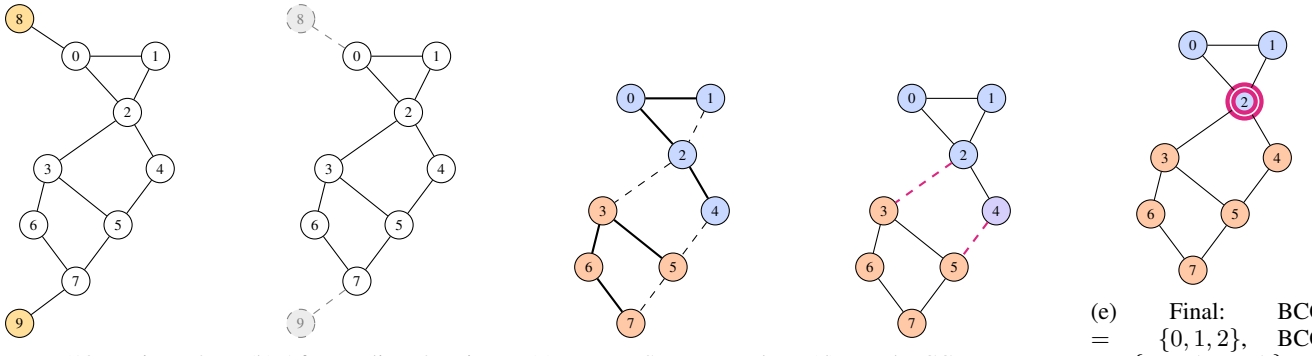


Fig. 1: Our multi-phase pipeline.



(a) Input (10 vertices; 8, 9 have degree 1).

(b) After peeling: 8 active (80%).

(c) Two BFS trees: seeds 0, 6.

(d) Local BCCs; magenta = cross edges.

(e) Final: BCC1 $= \{0, 1, 2\}$, BCC2 $= \{2, 3, 4, 5, 6, 7\}$, AP $= 2$.

Fig. 2: Steps of our Proposed approach for detecting BCC

II. Experiments

We evaluate our approach on 17 real-world networks spanning web graphs, social networks, road networks, and citation and biological networks. We run all experiments on a single NVIDIA A100 GPU. Figure 3 report per-network runtimes against five baselines: WK-18 [8], WK-17 [7], the CPU-based FAST-BCC [9] implementation, FastBCC-GPU [9] [5], and Filter [5].

Across this evaluation, our approach performs well across network types, including web graphs, social networks, and road networks. Our approach substantially outperforms WK-18, WK-17, and the CPU-based FAST-BCC baseline on every network we test. When we compare our approach against FastBCC-GPU, the current state-of-the-art GPU-based approach, our approach is competitive, though it does not yet exceed FastBCC-GPU's performance on every network. We discuss the parameter sensitivity underlying this gap, along with a detailed ablation study we plan to characterize how the depth cap and root count selection affect performance across network types, in Section IV.

III. Limitations

We manually set the maximum per-tree depth and random root count in Phase 2. These two parameters significantly impact overall performance. On high-diameter graphs (e.g., `road-usa`), an overly restrictive depth cap forces a large fraction of vertices into single-vertex fallback trees. This fragmentation produces a massive number of cross-tree edges, which drastically increases the computational cost of the Phase 4 merge step. We view Phase 4 as our primary bottleneck because its workload scales directly with this cross-edge fragmentation.

While increasing the depth cap mitigates fragmentation, the ideal threshold depends heavily on graph-specific structural properties rather than a universal constant. Furthermore, increasing the root count without concurrently tuning the depth cap also inflates cross-edge counts and Phase 4 runtime. We have not yet fully characterized the complex interactions between these parameters across varying graph topologies.

IV. Conclusion and Future Work

We presented a GPU-resident, multi-phase pipeline that computes biconnected components on large-scale undirected graphs. By combining 1-core peeling, depth-capped multi-root BFS, and local non-tree edge resolution, our approach completely avoids sequential DFS and outperforms earlier GPU baselines.

Our current limitations define our immediate future work. First, we plan a systematic ablation study to vary depth caps, vertex caps, and root counts across diverse graph topologies. This study will precisely characterize how these parameters impact fragmentation and runtime. Second, we will introduce a lightweight preprocessing step that inspects basic structural characteristics (such as vertex/edge counts and estimated diameter) to automatically select optimal tree constraints, effectively replacing our manual tuning. We expect this automation to inherently reduce cross-tree fragmentation. Finally, we will develop GPU-native optimizations for Phase 4 to reduce the sequential dependency of merging components across tree boundaries. This will allow the algorithm to scale gracefully even when it encounters high cross-edge counts.

References

[1] J. Cheriyan and R. Thurimella, "Algorithms for parallel k-vertex connectivity and sparse certificates," in *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing (STOC)*, pp. 391–401, 1991.

[2] J. Hopcroft and R. Tarjan, "Efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, 1973.

[3] J. H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

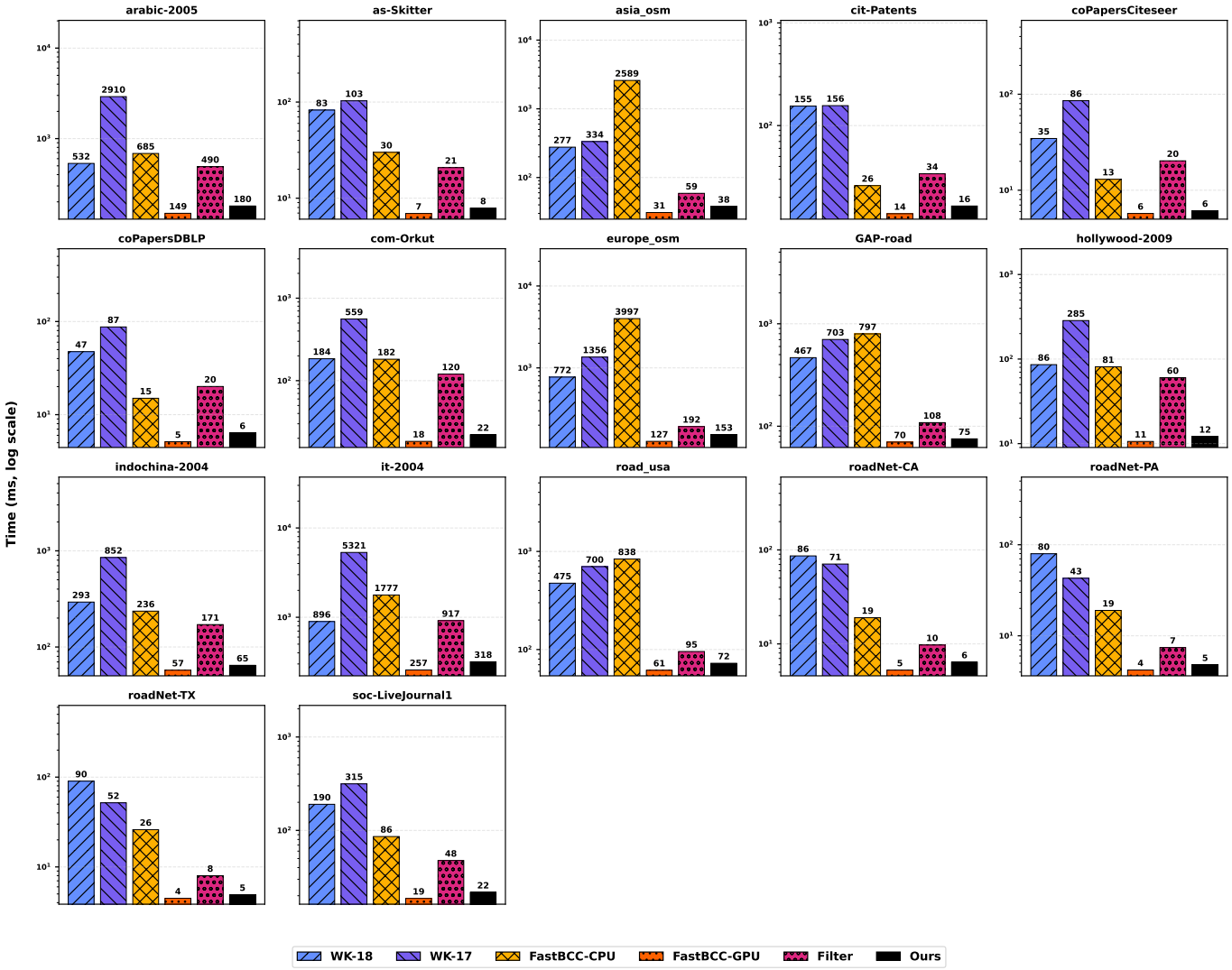


Fig. 3: Runtime comparison, log scale, in milliseconds, across 17 real-world graphs. Bars are annotated with exact values.

[4] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM Journal on Computing*, vol. 14, no. 4, pp. 862–874, 1985.

[5] A. Sahu, A. S. P. V. M. Aditya, G. Ramakrishna, K. Kothapalli, and D. S. Banerjee, "GPU algorithms for biconnected components on large graphs," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2026.

[6] G. Cong and D. A. Bader, "An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs)," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[7] M. Wadwekar and K. Kothapalli, "A fast GPU algorithm for biconnected components," in *2017 Tenth International Conference on Contemporary Computing (IC3)*, pp. 1–6, IEEE, 2017.

[8] K. Kothapalli and M. Wadwekar, "Expediting parallel graph connectivity algorithms," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pp. 72–81, IEEE, 2018.

[9] X. Dong, L. Wang, Y. Gu, and Y. Sun, "Provably fast and space-efficient parallel biconnectivity," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 52–65, 2023.