Provably Fast and Space-Efficient Parallel Biconnectivity

Xiaojun Dong
UC Riverside
xdong038@ucr.edu

Letong Wang
UC Riverside
lwang323@ucr.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

Abstract

Computing biconnected components (BCC) of a graph is a fundamental graph problem. The canonical parallel BCC algorithm is the Tarjan-Vishkin algorithm, which has $O(n+m)$ optimal work and polylogarithmic span on a graph with n vertices and m edges. However, Tarjan-Vishkin is not widely used in practice. We believe the reason is the space-inefficiency (it uses $O(m)$ extra space). In practice, existing parallel implementations are based on breath-first search (BFS). Since BFS has span proportional to the diameter of the graph, existing parallel BCC implementations suffer from poor performance on large-diameter graphs and can be slower than the sequential algorithm on many real-world graphs.

We propose the first p arallel b iconnectivity algorithm (FAST-BCC) that has optimal work, polylogarithmic span, and is space-efficient. Our algorithm creates a skeleton graph based on any spanning tree of the input graph. Then we use the connectivity information of the skeleton to compute the biconnectivity of the original input. We carefully analyze the correctness of our algorithm, which is highly non-trivial.

We implemented FAST-BCC and compared it with existing implementations, including GBBS, Slota and Madduri's algorithm, and the sequential Hopcroft-Tarjan algorithm. We tested them on a 96-core machine on 27 graphs with varying edge distributions. FAST-BCC is the fastest on *all* graphs. On average (geometric means), FAST-BCC is 3.1× faster than the *best existing baseline* on each graph.

CCS Concepts: • **Theory of computation → Shared memory algorithms**; **Graph algorithms analysis**; **Parallel algorithms**.

Keywords: Parallel Algorithms, Graph Algorithms, Biconnectivity, Connectivity, Graph Analytics

1 Introduction

Graph biconnectivity is one of the most fundamental graph problems. Given an *undirected* graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, a ***connected component (CC)*** is a maximal subset in V such that every two vertices in it are connected by a path. A ***biconnected component (BCC)*** (or blocks) is a maximal subset $C \subseteq V$ such that C is connected and remains connected after removing any vertex $v \in C$. In this paper, we use BCC (or CC) for both the biconnected (or connected) component in the graph and the problem of computing all BCCs (or CCs). BCC has extensive applications such as planarity testing [8, 24, 46], centrality computation [48, 59, 60], and network analysis [7, 56].

Sequentially, the Hopcroft-Tarjan algorithm [45] for BCC uses $O(n + m)$ work. However, this algorithm requires generating a spanning tree of G based on the depth-first search (DFS), which is considered hard to be parallelized [57]. Later, Tarjan and Vishkin proposed the canonical parallel BCC algorithm [65]. It uses an ***arbitrary spanning tree (AST)*** (a spanning tree with any possible shape) of the graph instead of the depth-first tree. Tarjan-Vishkin algorithm has $O(n+m)$ optimal work (number of operations) and polylogarithmic span (longest dependent operations), assuming an efficient parallel CC algorithm.

Although the Tarjan-Vishkin algorithm is theoretically considered "optimal" in work and span, significant challenges still remain in achieving a high-performance implementation in practice. The main issue in Tarjan-Vishkin is space-inefficiency. Tarjan-Vishkin generates an auxiliary graph $G' = (V', E')$ (which we refer to as the ***skeleton***), where every edge $e \in E$ maps to a vertex in V'. Tarjan and Vishkin showed that computing CC on G' gives the BCC on G, and we refer to this step as the ***connectivity*** phase. This ***skeleton-connectivity*** framework is adopted in many later papers. Such algorithms first generate a skeletonas an auxiliary graph G' from G, and then finds the CCs on G' that reflect BCC information on the input graph G. Unfortunately, in Tarjan-Vishkin, generating the skeleton G' and computing CC on G' take $O(m)$ extra space, which greatly increases the memory usage and slows down the performance.

In practice, most existing parallel BCC implementations also follow the ***skeleton-connectivity*** framework but overcome the space issue by using other skeletons based on breadth-first search (BFS) trees [25, 26, 29, 31, 40, 64, 68]. These algorithms either use skeletons with $O(n)$ size [25, 26, 29, 40, 68] or maintain implicit skeletons with $O(n)$ auxiliary space [31, 64]. We say a BCC algorithm is ***space-efficient*** if it uses $O(n)$ auxiliary space (other than the input graph). However, since computing BFS has span proportional to the graph, these BFS-based algorithms can be fast on small-diameter graphs (e.g., social and web graphs), but have poor performance on large-diameter graphs (e.g., k-nn and road

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577483

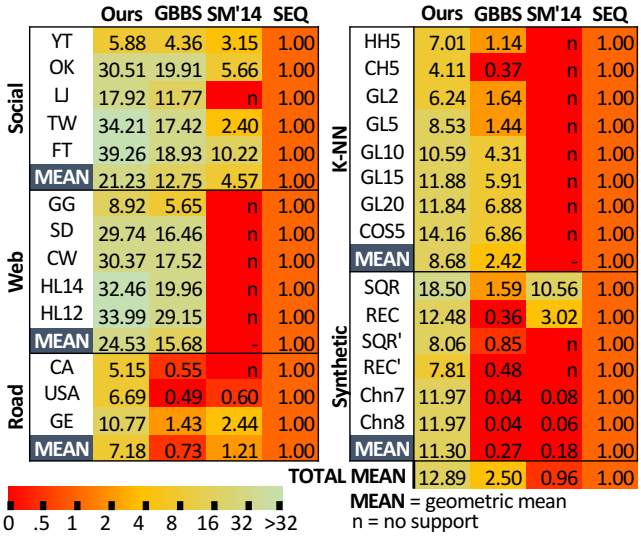

Figure 1. The heatmap of relative speedup for parallel BCC algorithms over the sequential Hopcroft-Tarjan algorithm [45] using 96 cores (192 hyper-threads). Larger/green means better. The numbers indicate how many times a parallel algorithm is faster than sequential Hopcroft-Tarjan (< 1 means slower). The two baseline algorithms are from [31, 64]. Complete results are in Tab. 2.

graphs). In our experiments, we observe that existing parallel implementations can even be slower than sequential Hopcroft-Tarjan on many real-world graphs (see GBBS [31] and SM'14 [64] in Fig. 1).

In this paper, we give the first space-efficient ($O(n)$ auxiliary space) parallel biconnectivity algorithm that has efficient $O(m+n)$ work and polylogarithmic span. Our skeleton G' is based on an arbitrary spanning tree (AST). Unlike Tarjan-Vishkin, our G' is a subgraph of G and can be maintained *implicitly* in $O(n)$ auxiliary space. The key idea is to carefully identify some ***fence*** edges, which indicate the "boundaries" of the BCCs. At a high level, we categorize all graph edges into fence tree edges, plain (non-fence) tree edges, back edges, and cross edges. Our skeleton G' contains the plain tree edges and cross edges. Using $O(n)$ space, we can efficiently determine the category of each edge in G. When processing the skeleton, we use the input graph G but skip the fence and back edges. We show that the BCC information of G can be constructed from the CC information of G' plus some simple postprocessing. Since our algorithm is based on **Fencing an Arbitrary Spanning Tree**, we call our algorithm **FAST-BCC**. More details of FAST-BCC are in Fig. 2. We note that conceptually our algorithm is simple, but the correctness analysis is highly non-trivial.

We implement our theoretically-efficient FAST-BCC algorithm and compare it to the state-of-the-art parallel BFS-based BCC implementations GBBS [31] and SM'14 [64]. We also compare FAST-BCC to the sequential Hopcroft-Tarjan algorithm. We test 27 graphs, including social, web, road, k-NN, and synthetic graphs, with significantly varying sizes and edge distributions. The details of the graphs and results are given in Tab. 2. We also show the relative running time in Fig. 1, normalized to the sequential Hopcroft-Tarjan.

On a machine with 96 cores, FAST-BCC is the fastest on ***all*** tested graphs. We use the geometric means to compare the "average" performance across multiple graphs. Due to work- and space-efficiency, our algorithm running on one core is competitive with Hopcroft-Tarjan (2.8× slower on average). Polylogarithmic span leads to good parallelism for ***all types of graphs*** (15–66× self-relative speedup on average). On small-diameter graphs (social and web graphs), although GBBS and SM'14 also achieve good parallelism, FAST-BCC is still 1.2–2.1× faster than the best of the two, and is 5.9–39× faster than sequential Hopcroft-Tarjan. For large-diameter graphs (road, k-nn, grid, and chain graphs), existing BFS-based implementations can perform worse than Hopcroft-Tarjan. Due to low span, FAST-BCC is 1.7–295× faster than GBBS (10× on average), and 4.1–18.5× faster than sequential Hopcroft-Tarjan (9.2× on average). On all graphs, FAST-BCC is 3.1× faster on average than the best of the three existing implementations. Our code is publicly available [36]. We present more results and analyses in the full version of this paper [37].

2 Preliminaries

Computational Model. We use the work-span (or work-depth) model for fork-join parallelism with binary forking to analyze parallel algorithms [15, 30], which is recently used in many papers on parallel algorithms [3, 10, 11, 13, 14, 16–22, 33–35, 42, 43, 62, 71]. We assume a set of threads that share a common memory. A process can `fork` two child software threads to work in parallel. When both children complete, the parent process continues. The ***work*** of an algorithm is the total number of instructions and the ***span*** (depth) is the length of the longest sequence of dependent instructions in the computation. We say an algorithm is ***work-efficient*** if its work is asymptotically the same as the best sequential algorithm. We can execute the computation using a randomized work-stealing scheduler [6, 23] in practice. We assume unit-cost atomic operation `compare_and_swap`(p, v_{old}, v_{new}) (or `CAS`), which atomically reads the memory location pointed to by p, and write value v_{new} to it if the current value is v_{old}. It returns *true* if successful and *false* otherwise.

Notation. Given an undirected graph $G = (V, E)$, we use $n = |V|$, $m = |E|$. Let $\mathrm{diam}(G)$ be the diameter of G, and x–y be an edge between x and y. ***CC*** and ***BCC*** are defined in Sec. 1. An ***articulation point*** (or cut vertex) is a vertex s.t. removing it increases the number of CCs. A ***bridge*** (or cut edge) is an edge s.t. removing it increases the number of CCs. A spanning tree T of a connected graph G is a spanning subgraph of G that contains no cycles. The spanning forest is defined similarly if G is disconnected. For simplicity, we assume G is connected, but our algorithm and implementation work on any graph. Given a graph G and a rooted spanning

$G = (V, E)$	: Input Graph	$T = (V, E_T)$	: A spanning tree in G
$a, b, c, u, v, h, w, x, y, z, u', v', c' \cdots \in V$			: Vertices in G
x–$y \in E$	: An edge in G	C, C_i	: A BCC in G
T_u	: u's subtree in T	h_C	: The BCC head of C
$p(u)$	: u's parent in T	$x \rightsquigarrow y$	: A tree path in T
$P = x$–y–$\cdots$	: A path	G'	: The skeleton

Fence edge: $(p(v), v) \in E_T$, $\nexists\, (x, y) \in E$, s.t. $x \in T_v$ and $y \notin T_{p(v)}$ (no edge from v's subtree escapes from $p(v)$'s subtree)
Plain edge: $(p(v), v) \in E_T$, $(p(v), v)$ is not a fence edge
Back edge, Cross edge: Edges in $E \setminus E_T$, defined as usual
Skeleton $G' = (V, E')$ in **FAST-BCC**: $E' = \{$plain & cross edges$\}$

Table 1. Notations and terminologies in this paper.

tree T, an edge is a ***tree edge*** if it is in T. A non-tree edge is a ***back edge*** if one endpoint is the ancestor of the other endpoint, and a ***cross edge*** otherwise. Fig. 2 Step 3 shows an illustration. If T is a BFS tree, there are no back edges; if T is a DFS tree, there are no cross edges. We use $x \rightsquigarrow y$ to denote the tree path between x and y on T. We denote the parent of vertex u as $p(u)$, and the subtree of u as T_u. The notation used in this paper is given in Tab. 1.

We use $O(f(n))$ *with high probability* (*whp*) in n to mean $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$.

Euler tour technique (ETT). ETT is proposed by Tarjan and Vishkin [65] in their BCC algorithm to root a spanning tree. Later, ETT becomes a widely-used primitive in both sequential and parallel settings, including computational geometry [2], graph algorithms [5, 28, 67], maintaining subtree or tree path sums [30], and many others. ETT is needed in Tarjan-Vishkin because when an arbitrary spanning tree is generated for a graph (e.g., from a CC algorithm), it is not rooted, and thus we do not have the parent-child information for the vertices. Given an unrooted tree T with $n - 1$ edges, ETT finds an Euler tour of T, which is a cycle traversing each edge in T exactly twice (once in each direction). ETT first constructs a linked list on the $2n - 2$ directed tree edges, and runs list ranking on it. We refer the audience to the textbooks on parallel algorithms [47, 58] for more details on ETT. Using the semisort algorithm from [15, 44] and list ranking from [15], ETT costs $O(n)$ expected work and $O(\log n)$ span *whp*. Given T, we can set any vertex as the root of T, and use ETT to determine the directions of the edges. We can then determine the parent of any vertex, and whether an edge is a tree edge, back edge, or cross edge in $O(1)$ work.

3 Existing BCC Algorithms

This section reviews the existing BCC algorithms and implementations. We will use the *skeleton-connectivity framework* to describe the existing BCC algorithms. The *skeleton phase* generates a skeleton G' from G, which is an auxiliary graph. Then the *connectivity phase* computes the connectivity on G' to construct the BCCs of G. Existing BCC algorithms can be categorized by how the skeleton G' is generated. The Hopcroft-Tarjan algorithm uses DFS-based skeletons; the Tarjan-Vishkin Algorithm generates a skeleton based on an arbitrary spanning tree (AST); almost all other BCC algorithms (see Sec. 3.3) use BFS-based skeletons.

3.1 The Hopcroft-Tarjan Algorithm

Sequentially, Hopcroft-Tarjan BCC algorithm [45] has $O(n + m)$ work using a depth-first search (DFS) tree T. Based on T, two ***tags*** $first[\cdot]$ and $low[\cdot]$ are assigned to each vertex. $first[v]$ is the preorder number of each vertex in T. $low[v]$ gives the earliest (smallest preorder) vertex incident on any vertex $u \in T_v$ via a non-tree edge and u itself. More formally,

$$low[v] = \min\{w_1[u] \mid u \in V \text{ is in the subtree rooted at } v\}$$
$$w_1[u] = \min(\{first[u]\} \cup \{first[u'] \mid (u, u') \notin T\})$$

To compute the BCCs, an additional stack is maintained. Each time we visit a new edge, it is pushed into the stack. When an articulation point $p(u)$ is found by u ($low[u] \geq first[p(u)]$), edges are popped from the stack until u–$p(u)$ is popped. These edges and the relevant vertices form a BCC.

Conceptually, the skeleton in Hopcroft-Tarjan is the DFS tree without the "fence edges" of u–$p(u)$ when $low[u] \geq first[p(u)]$. This insight also inspires our BCC algorithm.

3.2 The Tarjan-Vishkin Algorithm

Hopcroft-Tarjan uses a DFS tree as the skeleton, but DFS is inherently serial and hard to be parallelized [57]. To parallelize BCC, the Tarjan-Vishkin algorithm [65] uses an arbitrary spanning tree (AST) instead of a DFS tree. This spanning tree T can be obtained by any parallel CC algorithm. The algorithm then uses ETT (which was also proposed in that paper) to root the tree T (see Sec. 2). Then the algorithm builds a skeleton $G' = (E, E')$ and runs a connectivity algorithm on it. We describe G' in more details in the full paper [37], and only briefly review it here. The vertices in G' correspond to the edges in G[1]. To determine the edges in G', the algorithm uses four tags ($first[\cdot]$, $last[\cdot]$, $low[\cdot]$, and $high[\cdot]$) for each vertex. Here $first[u]$ and $last[u]$ are the first and last appearance of vertex u in the Euler tour (note that this is not the same $first[\cdot]$ in Hopcroft-Tarjan, but conceptually equivalent). $low[\cdot]$ is the same as defined in Hopcroft-Tarjan, and $high[\cdot]$ is defined symmetrically:

$$high[v] = \max\{w_2[u] \mid u \in V \text{ is in the subtree rooted at } v\}$$
$$w_2[u] = \max(\{first[u]\} \cup \{first[u'] \mid (u, u') \notin T\})$$

All tags can be computed in $O(n + m)$ expected work and $O(\log n)$ span *whp* using ETT. Tarjan-Vishkin then finds the CCs on G' to compute the BCCs of G. However, G' in Tarjan-Vishkin can be large, making the algorithm less practical.

Assuming an efficient ETT and a parallel CC algorithm, Tarjan-Vishkin uses $O(n + m)$ optimal expected work and polylogarithmic span. However, the space-inefficiency hampers the practicability of Tarjan-Vishkin since G' contains $O(m)$ edges. In our experiments, Tarjan-Vishkin takes up to

[1] In a later paper [39], it was shown that the number of vertices in G' can be reduced to $O(n)$, but $|E'|$ is still $O(m)$.

11× extra space than our FAST-BCC or GBBS. On our machine with 1.5TB memory, Tarjan-Vishkin ran out of memory when processing the Clueweb graph [54], although it only takes about 300GB to store the graph (see discussions in the full version [37]). The large space usage forbids running Tarjan-Vishkin on large-scale graphs on most multicore machines. Even for small graphs, high space usage can increase memory footprint and slow down the performance.

Some existing BCC implementations (e.g., GBBS [31] and TV-filter [29]) were also described as Tarjan-Vishkin algorithms, probably because they also use the skeleton-connectivity framework. We note that their correctness relies on BFS-based skeletons (i.e., sparse certificates [27]), and we categorized them below together with a few other algorithms.

3.3 Other Existing Algorithms / Implementations

Before Tarjan-Vishkin, Savage and JáJá [61] showed a parallel BCC algorithm based on matrix-multiplication with $O(n^3 \log n)$ work. Tsin and Chin [66] gave an algorithm that uses an AST-based skeleton. It is quite similar to Tarjan-Vishkin, but uses $O(n^2)$ work.

To achieve space-efficiency, many later parallel BCC algorithms use BFS-based skeletons [25, 26, 29, 31, 40, 50, 64, 68]. Many of them use the similar idea of sparse certificates [27]. BCC is much simpler with a BFS tree—all non-tree edges are cross edges with both endpoints in the same or adjacent levels. Cong and Bader's TV-filter algorithm [29] uses the skeleton as the BFS tree T and an arbitrary spanning tree/forest for $G \setminus T$ ($O(n)$ total size). Slota and Madduri's algorithms [64] and Dhulipala et al.'s algorithm [31] use the skeletons as the input graph G excluding $O(n)$ vertices/edges. The other algorithms [25, 26, 40, 68] use a BFS tree as the skeleton, and compute connectivity dynamically. All these algorithms are space-efficient. Their skeleton graphs either have $O(n)$ size [25, 26, 29, 40, 68] or can be implicitly represented using $O(n)$ information [31, 64]. However, the span to generate a BFS tree is proportional to the diameter of the graph, which is inefficient for large-diameter graphs.

3.4 Space-Efficient BCC Representation

Since some vertices (articulation points) appear in multiple BCCs (see Fig. 2 as an example), we need a representation of all BCCs in a space-efficient manner ($O(n)$ space). We use a commonly used representation [11, 31, 40] in our algorithm. Given a spanning tree T, we assign a label for each vertex except for the root of T, indicating which BCC this vertex is in. For all vertices with the same label, we find another vertex called the ***component head*** (see details in Sec. 4.1) attached to this label. All vertices with the same label and the corresponding component head form a BCC. An example of this representation is given in Fig. 2. It is easy to see that this representation uses $O(n)$ space since we have $n-1$ labels for all vertices and at most $n-1$ component heads.

Algorithm 1: The FAST-BCC algorithm

Input: An undirected graph $G = (V, E)$
Output: The labels $l[\cdot]$ for vertices, and the component head for each BCC

```
1  Compute the spanning forest F of G                          ▷ First CC
2  Root all trees in F using the Euler tour technique          ▷ Rooting
3  Compute tags (e.g., low, high) of each vertex based on the
     Euler tour                                                ▷ Tagging
4  Compute the vertex label l[·] using connectivity on G with
     edges satisfying InSkeleton(u, v) = true                  ▷ Last CC
5  ParallelForEach u ∈ V with l[u] ≠ l[p(u)]
6  |  Set the component head of l[u] as p(u)
7  Function InSkeleton(u, v) ▷ Decide if u–v is in skeleton G'
8  |  if (u, v) is a tree edge then
9  |  |  return ¬ Fence(u, v) and ¬ Fence(v, u)
10 |  else return ¬ Back(u, v) and ¬ Back(v, u)
11 Function Fence(u, v)        ▷ Decide if tree edge is fence edge
12 |  return first[u] ≤ low[v] and last[u] ≥ high[v]
13 Function Back(u, v)  ▷ Decide if non-tree edge is back edge
14 |  return first[u] ≤ first[v] and last[u] ≥ first[v]
```

4 The FAST-BCC Algorithm

In this section, we present our FAST-BCC algorithm with analysis. Our algorithm is the first parallel BCC algorithm that is work-efficient, space-efficient, and has polylogarithmic span. Recall that BFS-based algorithms are space-efficient, but BFS itself does not parallelize well. Tarjan-Vishkin is based on AST and is highly parallel, but generating the skeleton is space-inefficient. To achieve both high parallelism and space efficiency, we need novel algorithmic insights.

Interestingly, our key idea is to revisit the sequential DFS-based Hopcroft-Tarjan algorithm (Sec. 3.1). Although DFS is inherently sequential, the insights in Hopcroft-Tarjan inspire our parallel BCC algorithm. The (implicit) skeleton in Hopcroft-Tarjan is simple and the skeleton size is small ($O(n)$). Unlike many later parallel BCC algorithms with the high-level ideas to combine cycles (based on Fact 4.2), the idea in Hopcroft-Tarjan is the "fencing" condition as follows. When computing the CC on the skeleton G' (the DFS tree) and traversing the edge from v to $p(v)$, the CC on G' (BCC on G) is *fenced* if $low[v] \geq first[p(v)]$. This condition partitions the DFS tree T into multiple CCs that correspond to BCCs in G. Note that G' in Hopcroft-Tarjan only contains edges from the DFS tree, because there are no cross edges in DFS trees and all back edge information is captured by $low[\cdot]$.

Now we try to generalize this idea to an arbitrary spanning tree (AST). Directly using the "fencing" condition in Hopcroft-Tarjan does not work since we need to deal with cross edges. Note that a fence edge v–$p(v)$ in Hopcroft-Tarjan means that *vertices in u's subtree do not have an edge that escapes (i.e., the other endpoint is outside) $p(u)$'s subtree.* We define our fence edges also based on this condition. More formally, we say a tree edge (u, v) where $u = p(v)$ is a fence edge

if there is no edge $(x, y) \in E$ such that $x \in T_v$ and $y \notin T_u$. Intuitively, it means v's subtree T_v is "isolated" from other parts outside $p(v)$'s subtree, and only interacts with the outside world through $p(v)$. To get an equivalent condition for an AST, we borrow the idea from Tarjan-Vishkin and also compute four axillary arrays $first[\cdot]$, $last[\cdot]$, $low[\cdot]$, and $high[\cdot]$. The "fencing" condition then becomes $low[v] \geq first[p(v)]$ and $high[v] \leq last[p(v)]$. A non-fence tree edge is referred to as a ***plain edge***. Note that the information for back edges is already captured by the $low[\cdot]$ and $high[\cdot]$ arrays, which will also be used to decide fence edges. Our algorithm will ignore back edges as in Hopcroft-Tarjan, and our skeleton G' contains plain tree edges and cross edges. Since the main approach in our algorithm is Fencing an Arbitrary Spanning Tree, we call our algorithm **FAST-BCC**. We note that the high-level idea of fencing (finding some special edges on the spanning tree) is also used in some existing work [11, 31, 64]. Our design of the skeleton and the fencing condition is the first to achieve work-efficiency, polylogarithmic span, and space-efficiency for the BCC problem.

The outline of the algorithm is given in Fig. 2, and the pseudocode is in Alg. 1. Although our fencing algorithm is simple, we note that formally proving the correctness (Sec. 4.2) is highly non-trivial.

4.1 Algorithmic Details

Our FAST-BCC algorithm has four steps: *First-CC* (generate spanning trees), *Rooting* (root the spanning trees using ETT), *Tagging* (compute $first[\cdot]$, $last[\cdot]$, $w_1[\cdot]$, $w_2[\cdot]$, $low[\cdot]$, $high[\cdot]$, $p[\cdot]$), and *Last-CC* (run CC on the skeleton and post-processing). In the skeleton-connectivity framework, the first three steps are the skeleton phase (compute the skeleton G'), and the last step is the connectivity phase (run CC on G' to find all BCCs in G).

First-CC (Step 1 in Fig. 2, Line 1 in Alg. 1). This step finds all CCs in G and generates a spanning forest F of G. For simplicity, in the following, we focus on one CC and its spanning tree T, which is unrooted at this moment. If G contains multiple CCs, they are simply processed in parallel. Running CC only requires $O(n)$ auxiliary space.

Rooting (Step 2 in Fig. 2, Line 2 in Alg. 1). We use the Euler tour technique (ETT) in Sec. 2 to root T, which implies the tree edge directions (Fig. 2, Step 2). ETT requires $O(n)$ space.

Tagging (Step 3 in Fig. 2, Line 3 in Alg. 1). This step generates the tags used in the algorithm, including $w_1[\cdot]$, $w_2[\cdot]$, $low[\cdot]$, $high[\cdot]$, $first[\cdot]$, $last[\cdot]$ (same as in Tarjan-Vishkin, see Sec. 3) and the parent array $p[\cdot]$. $low[\cdot]$ and $high[\cdot]$ values are computed by looping over all edges and getting arrays w_1 and w_2, and applying n 1D range-minimum queries (RMQ). This step takes in $O(n + m)$ work and $O(\log n)$ span [15]. These tags will help to decide the four edge types (see details below). All the tag arrays have size $O(n)$.

Last-CC (Step 4 in Fig. 2, Line 4–6 in Alg. 1). As mentioned, our skeleton graph G' contains plain tree edges and cross edges. To achieve space efficiency, we do not explicitly store G'. Since G' is a subgraph of G, we can directly use G but skip the fence edges and back edges, which can be determined using the tags generated in Step 3 (Line 7–14). Then we compute the CCs on the skeleton G' (Line 4), which assigns a label $l[v]$ to each vertex (Fig. 2, Step 4.1). In Lem. 4.11, we show that if two vertices are connected in G', they must be biconnected on the input graph G. We then assign the head to each label (Lines 5 and 6) by looping over all fence edges (Fig. 2, Step 4.2). For a fence edge u–$p(u)$, if u and $p(u)$ have different labels (Line 5), $p(u)$ (intuitively) isolates vertices below u with the other parts in the graph. Thus, we assign $p(u)$ as the component head of u's CC in G'. We prove the correctness of this step in Lem. 4.9 and 4.12. This step also only requires $O(n)$ auxiliary space, which is needed by running CC on G but skip certain edges.

4.2 Correctness for the FAST-BCC Algorithm

We now prove the correctness of our algorithm. Note that our algorithm will identify the spanning forest in the first step and deal with each CC respectively. For simplicity, throughout the section, we focus on one CC in G.

In the following, when we use the concepts about a spanning tree of the graph (e.g., root, parent, child, and subtree), we refer to the specific spanning tree identified in Step 1 of our algorithm, and use T to represent it. Recall that T_u denotes the subtree rooted at vertex u, and $u \sim v$ denotes the tree path on T from u to v. Some other notation is given in Tab. 1. In a spanning tree, we say a node u is shallower (deeper) than v if u is closer (farther) to the root than v. We use node and vertex interchangeably.

We note that although Alg. 1 is simple, the correctness proof is sophisticated. We show the relationship of facts, lemmas, and theorems in Fig. 2. Due to the space limit, the proofs for Fact 4.1 and 4.2 and Lem. 4.3 to 4.5 are given in the full version of the paper, and we mainly focus on the proofs that reflect some key ideas in our new algorithm.

We first show some facts for BCCs based on the definition.

Fact 4.1. *Two BCCs share at most one common vertex.*

Fact 4.2. *For a cycle in a graph, all vertices on the cycle are in the same BCC.*

Lemma 4.3. *Given a graph G, vertices in each BCC $C \subseteq V$ must also be connected in an arbitrary spanning tree T for G.*

Since each BCC C must be connected in the spanning tree in T, there must exist a unique shallowest node in this BCC on T. We call this shallowest node the ***BCC head*** of the BCC C, and denote it as h_C.

Lemma 4.4. *Each non-root BCC head is an articulation point. An articulation point must be a BCC head.*

Lemma 4.5. *The function* InSkeleton *(Line 7) in Alg. 1 can correctly skip the fence and back edges.*

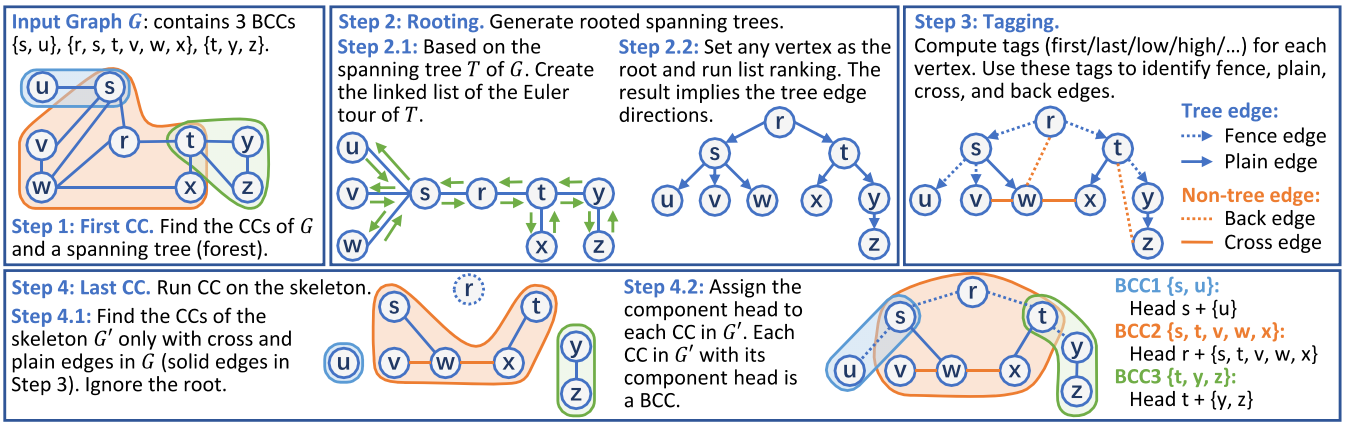

Figure 2. The outline of the FAST-BCC algorithm and a running example. The four steps are explained in detail in Sec. 4.1.

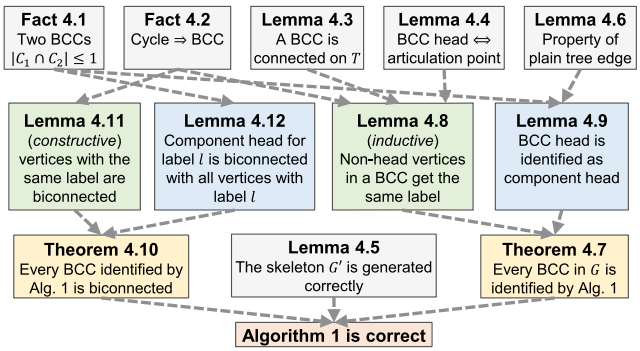

Figure 3. The structure of the correctness proof for Alg. 1.

Next, we show a useful property of the plain tree edges.

Lemma 4.6. *For a plain tree edge x–y where x is the parent of y, let z be x's parent, then x, y, z are biconnected.*

Proof. Since x–y is not a fence edge, there must be an edge a–b, s.t. $a \in T_y$ and $b \notin T_x$. The cycle $y \sim a$–$b \sim z$–x–y then contains x, y, and z. Due to Fact 4.2, x, y, and z are in the same BCC. □

Next, we show that Alg. 1 can correctly identify all BCCs. We will show two directions. First, if two vertices u and v are biconnected, Alg. 1 must put them in a BCC. Second, for any two vertices u and v in a BCC found by Alg. 1, they must be biconnected.

Theorem 4.7. *For $u, v \in V$, if they are biconnected, Alg. 1 assigns them to the same BCC.*

To prove Thm. 4.7, we discuss two cases: 1) one of u and v is a BCC head, and 2) neither of them is a BCC head.

Lemma 4.8. *For a BCC C and two vertices $u, v \in C \setminus \{h_C\}$, they are connected in the skeleton G' and will get the same label in Alg. 1.*

Proof. If all tree edges connecting $C \setminus \{h_C\}$ are plain tree edges, u and v are already connected in G'. Next, we show that the two endpoints of every fence edge are also connected in G'. To do so, we first sort (only conceptually) all vertices in $C \setminus \{h_C\}$ by their depth in T. Then we inductively show from bottom up (deep to shallow) that, given a vertex $v \in C$, $T_v \cap C$ (v's subtree in C) is connected in G'.

The base case is the deepest vertices in $C \setminus \{h_C\}$. In this case, their subtree contains only one vertex so they are connected.

We now consider the inductive step—if for all vertices with depth $\ge d$, their subtrees in C are connected in G', then for all vertices with depth $d-1$, their subtrees in C are also connected in G'. Consider a vertex $u \in C \setminus \{h_C\}$ with depth $d-1$. If u has only one child v in C, then u–v is a plain tree edge since otherwise v's subtree cannot escape u's subtree and u is an articulation point (disconnecting v and $p(u)$), contradicting Lem. 4.4. Assume u has multiple children $c_1, \ldots, c_k$ in C. Let u–v be a fence edge that is not in G', where $v = c_i$ is a child of u. We will show that u and v are still connected in G'.

Since u is not a BCC head, $p(u)$ must also be in C. Based on the definition of BCC, if we remove u, v and $p(u)$ are still connected C. Let the path be $P = v$–x_1–x_2–...–x_k–$p(u)$ where $x_i \in C$ and $x_i \ne u$. We will construct a path in G' from P that connects v and u. Let x_{j+1} be the first vertex on path P that is not in T_u. We will use the path $v = x_0$–x_1–x_2–...–x_j. All nodes in this path have depths $\ge d$. Due to the induction hypothesis, if some of the edges are back or fence edges, we can replace them with the paths in G', and denote this path as P'. Then, since $x_{j+1} \notin T_u$ is connected to $x_j \in T_u$, all edges on tree path $x_j \sim u$ are plain tree edges. As a result, u and v are connected in G' using the path P' from v to x_j, and the tree path from x_j to u (all edges are in G'). By the induction, all vertices in $C \setminus \{h_C\}$ are connected in G', and hence get the same label after Line 4. □

Lemma 4.9. *Any BCC head will be correctly identified as a component head in Alg. 1.*

Proof. Consider a BCC C and its BCC head h_C. Among all the children of h_C, a subset S of them are in the same BCC C. Consider any $c \in S$. We will show that the edge c–h_C must be identified correctly in Line 5.

We first show that c–h_C must be a fence. If h_C is the root of T, and in this case, all tree edges connecting to h_C are fence edges. Otherwise, this can be inferred from the contrapositive of Lem. 4.6. If c–h_C is a plain tree edge, c, h_C, and $p(h_C)$ must be biconnected, which means $p(h_C)$ is also in the BCC C. This contradicts the assumption that h_C is the shallowest node (BCC head) in the BCC.

We then show that after we run the CC on the skeleton G' (Line 4), h_C and c have different labels (i.e., h_C and c are not connected in G'). Assume to the contrary that there exists a path P from c to h_C on G'. Consider the last node t on the path before h_C. Because h_C–c is a fence edge and is ignored in G', $c \neq t$. We discuss three cases. (1) t is not in the h_C's subtree T_{h_C}. Consider the first edge x–y on the path P such that $x \in T_{h_C}$ and $y \neq T_{h_C}$. Since x–y escapes h_C's subtree, the tree path $P' = x \sim h_C$ only contains plain tree edges. Let c' be h_C's child on the path P'. From Lem. 4.6, c', h_C, and $p(h_C)$ are biconnected. In this case, h_C–$c \sim x \sim c'$–h_C is a cycle, and Fact 4.2 shows that c', h_C and c are biconnected. The contrapositive of Fact 4.1 indicates that c', h_C, c, and $p(h_C)$ are all biconnected, contradicting the assumption that h_C is the BCC head (the shallowest node in the BCC). (2) $t \in T_{h_C}$, but t is not h_C's child. This is impossible because t–h_C is a back edge, which is not in G'. (3) t is a child of h_C. This case is similar to (1). By replacing c' in the previous proof by t, we can get the same contradiction. Combining all cases proves that there is no path in G' between h_C and its children in C, so $l[h_C]$ is different from the labels of its children in C. □

Combining Lem. 4.8 and 4.9, we can prove Thm. 4.7.

We then show the other direction—all the BCCs computed by Alg. 1 are indeed biconnected.

Theorem 4.10. *If two vertices u and v are identified as in the same BCC by Alg. 1, they must be biconnected.*

Similar to the previous proof, we consider two cases: (1) none of the two vertices is a component head (they are connected in G'), proved in Lem. 4.11, and (2) one of them is identified as a component head in Line 6, proved in Lem. 4.12.

Lemma 4.11. *If two vertices u and v are connected in the skeleton G', they are biconnected.*

Proof. Since u and v are connected in G', there exists a path P from u to v only using edges in G'. Let P be $u = p_0$–p_1–...–p_{k-1}–$p_k = v$. We will show that after removing any vertex p_i where $1 \leq i < k$ on P, p_{i-1} and p_{i+1} are still connected, meaning that u and v are biconnected. We summarize all possible local structures in three cases, based on whether p_{i-1} (and p_{i+1}) is a child of p_i in T.

Case 1: both p_{i-1} and p_{i+1} are p_i's children. Since p_{i-1}–p_i is not a fence edge, there must be an edge x–y s.t. $x \in T_{p_{i-1}}$ and $y \notin T_{p_i}$. Similarly, for p_i–p_{i+1}, there exists an edge (x', y') s.t. $x' \in T_{p_{i+1}}$ and $y' \notin T_{p_i}$. Hence, without using p_i, p_{i-1} and p_{i+1} are still connected by the path $p_{i-1} \sim x$–$y \sim y'$–$x' \sim p_{i+1}$. Here since $y, y' \notin T_{p_i}$, $y \sim y'$ does not contain p_i.

Case 2: one of p_{i-1} and p_{i+1} is p_i's child. WLOG, assume p_{i-1} is the child. Since p_{i-1}–p_i is not a fence edge, there must be an edge x–y such that $x \in T_{p_{i-1}}$ and $y \notin T_{p_i}$. Also, since p_{i+1} is either the parent of p_i or connected to p_i using a cross edge, $p_{i+1} \notin T_{p_i}$. Hence, without using p_i, p_{i-1} and p_{i+1} are still connected using the path $p_{i-1} \sim x$–$y \sim p_{i+1}$.

Case 3: neither p_{i-1} nor p_{i+1} is a child of p_i, and neither of them is in T_{p_i} (otherwise they are connected by a back edge). Without using p_i, p_{i-1} and p_{i+1} are still connected using the tree path $p_{i-1} \sim p_{i+1}$.

Since removing any vertex on the path P does not disconnect the path, all vertices in the same CC of the skeleton are biconnected. □

Lemma 4.12. *If Line 6 in Alg. 1 assigns h as the component head of a connected component (CC) C in the skeleton G', then h is biconnected with C.*

Proof. First of all, assume h is assigned as the component head because of its child c, where h–c is a fence edge. We will show that the connected component C in G' containing c is biconnected with h. There are two cases.

Case 1: C only contains vertices in T_c. This means that no vertices in T_c have a cross edge to another vertex outside T_c. Therefore, either all edges incident on $c' \in T_c$ do not escape from T_c, or some node $c' \in T_c$ is connected to nodes outside T_c via back edges. In the former case, all the edges connecting c and its children are fence edges, and thus C only contains c. In this case, h is trivially biconnected with C. In the latter case, assume $x \in T_c \cap C$ has a back edge connected to $y \notin T_c$. Note that y can only be h—if y is h's ancestor, then edge x–y escapes T_h, so h–c is a plain tree edge (contradiction). Therefore, we can find a cycle h–$c \sim x$–h. From Fact 4.2, h, c, x are biconnected, and h is in the same BCC as c and x, and thus all vertices in C (Lem. 4.11 and Fact 4.1).

Case 2: C contains both vertices in T_c and some vertices in $T_h \setminus T_c$. Hence, there exists a cross edge x–y, where $x \in T_c$ and $y \notin T_c$. We can find a cycle h,~ x–$y \sim h$. From Fact 4.2, h, c, u are biconnected. h is in the same BCC as c and u. □

Combining Lem. 4.11 and 4.12 proves Thm. 4.10.

Thm. 4.7 shows that if two vertices are put in the same BCC by Alg. 1, they are biconnected in G. Thm. 4.10 indicates that two vertices biconnected in G will be put in the same BCC by Alg. 1. Lem. 4.5 indications back edges and fence edges are identified correctly by Alg. 1. Combining them together indicates that Alg. 1 is correct.

4.3 Cost Bounds for the FAST-BCC Algorithm

We now analyze the cost bounds of the algorithm.

Theorem 4.13. *Alg. 1 computes the BCCs of a graph G with n vertices and m edges using $O(n+m)$ expected work, $O(\log^3 n)$ span* whp, *and $O(n)$ auxiliary space (other than the input).*

Proof. The first and last steps compute the graph connectivity twice. Graph connectivity can be computed in $O(n+m)$ expected work and $O(\log^3 n)$ span *whp* [63]. In Step 2, ETT can be performed $O(n)$ expected work and $O(\log n)$ span *whp* (see Sec. 2). In Step 3, computing $low[\cdot]$ and $high[\cdot]$ arrays based on RMQ takes $O(m)$ work and $O(\log n)$ span [15]. Adding all pieces together gives the work and span bounds.

For the space, all arrays for the tags have size $O(n)$. As mentioned, we do not generate the skeleton explicitly. In the last step, we try all the edges in G but skipping the back and fence edges. In all, the auxiliary space needed is $O(n)$. □

5 Implementation Details

We discuss some implementation details of FAST-BCC in this section.

Connectivity. Connectivity is used twice in FAST-BCC. The only existing parallel CC implementation with good theoretical guarantee we know of is the SDB algorithm [63] (an initial version of GBBS is based on this algorithm). A recent paper by Dhulipala et al. [32] gave 232 parallel CC implementations, many of which outperformed the SDB algorithm, but no analysis of work-efficiency was given. A more recent version of GBBS uses the `UF-Async` algorithm in [32] to compute CC. To achieve efficiency both in theory and in practice, FAST-BCC uses the ***LDD-UF-JTB*** algorithm from [32] and we provide a new analysis for this algorithm to prove its theoretical efficiency.

LDD-UF-JTB consists of two steps. It first runs a low-diameter decomposition (LDD) algorithm [55] to find a decomposition (partition of vertices) of the graph such that each component has a low diameter and the number of edges crossing different components is bounded. The second step is to use a union-find structure by Jayanti et al. [49] to union components connected by cross-component edges. We now show the bounds of this algorithm.

Theorem 5.1. *The LDD-UF-JTB algorithm computes the CCs of a graph G with n vertices and m edges using $O(n+m)$ expected work and $O(\log^3 n)$ span* whp.

Therefore, using LDD-UF-JTB for CC preserves the cost bounds in Thm. 4.13. We prove Thm. 5.1 in the full version of this paper.

We optimized LDD-UF-JTB using the hash bag and local search techniques proposed from [69]. These optimizations are only used in computing CCs in our algorithm, and we do not claim them as contributions of this paper. In our tests, using these optimizations improves the performance of FAST-BCC by 1.5× on average (up to 5×). Some results are shown in the full version of this paper. We note that among all 232 CC algorithms in [32], no one is constantly faster, and the relative performance is decided by the input graph properties. In FAST-BCC, we currently use the same CC algorithm for all graphs, and we acknowledge that using the fastest CC algorithm on each graph can further improve the performance of FAST-BCC. We choose LDD-UF-JTB mainly because it is theoretically-efficient, and also can generate CC as a by-product efficiently.

Spanning Forest. The spanning forest of G is obtained as a by-product of Step 1, which saves all edges to form the CCs. We then re-order the vertices in the compressed sparse row (CSR) format to let each CC be contiguous.

Euler Tour Technique (ETT). We use the standard ETT to root the spanning trees (see Sec. 2). We replicate each undirected edge in T into two directed edges and semisort them [44], so edges with the same first endpoint are contiguous. Then we construct a circular linked list as the Euler circuit. Assume a vertex v has k in-coming neighbors u_1, u_2, $\cdots$, u_k. For every incoming edge of v except for the last one, we link it to its next outgoing edge (i.e., u_i–v is linked to v–u_{i+1} for $1 \le i < k$). For the last incoming edge, we link it to the first outgoing edge of v (i.e., u_k–v is linked to v–u_1).

After we obtain the Euler circuit of the tree, we flatten the linked list to an array by list ranking, and acquire the Euler tour order of each vertex. For list ranking, we coarsen the base cases by sampling $\sqrt{n}$ nodes. We start from these nodes in parallel, with each node sequentially following the pointers until it visits the next sample. Then we compute the offsets of each sample by prefix sum, pass the offsets to other nodes by chasing the pointers from the samples, and scatter all nodes into a contiguous array.

Computing Tags. We use several tags w_1, w_2, *first*, *last*, *low*, and *high* for each vertex, defined the same as Tarjan-Vishkin [65] (see Sec. 3). We use `CAS` operations to compute *first* and *last* as they represent the first and last appearances of a vertex in the Euler tour order. For each tree edge (u, v), if $first[u] < first[v]$, we set $p(v) = u$, or vice versa. Computing *low* and *high* are similar, so we only discuss *low* here. We first initialize $w_1[v]$ with $first[v]$ for each $v \in V$. Then it traverses all non-tree edges u–v and updates $w_1[u]$ and $w_1[v]$ with the minimum of $first[u]$ and $first[v]$. We build a parallel sparse table [15] on w_1 to support range minimum queries. Note that $first[v]$ and $last[v]$ reflect the range of v's subtree in the Euler tour order. Thus, $low[v]$ can be computed by finding the minimum element in $w_1[\cdot]$ in the range between $first[v]$ and $last[v]$. $high[\cdot]$ can be computed similarly.

6 Experiments

Setup. We run our experiments on a 96-core (192 hyper-threads) machine with four Intel Xeon Gold 6252 CPUs, and 1.5 TB of main memory. We implemented all algorithms in C++ using ParlayLib [12] for fork-join parallelism and some parallel primitives (e.g., sorting). We use `numactl -i all` in experiments with more than one thread to spread the memory pages across CPUs in a round-robin fashion. We run each test for 10 times and report the median.

We tested on 27 graphs, including social networks, web graphs, road graphs, k-NN graphs, and synthetic graphs.

		n	m	D	#BCC	$\lvert BCC_1\rvert$%	Ours par.	Ours seq.	Ours spd.	GBBS par.	GBBS seq.	GBBS spd.	SM'14	SEQ	T_{best}/ours	Notes
Social	YT	1.13M	5.98M	23	673,661	39.83%	0.030	0.465	15.6	0.040	0.435	10.8	0.059	0.175	1.35	com-youtube [72]
	OK	3.07M	234M	9	68,117	97.76%	0.103	3.08	30.0	0.158	4.86	30.8	0.297	3.14	1.53	com-orkut [72]
	LJ	4.85M	85.7M	19	1,133,883	75.61%	0.104	3.02	28.9	0.159	3.34	21.0	n	1.87	1.52	soc-LiveJournal1 [9]
	TW	41.7M	2.41B	23	1,936,001	95.33%	1.44	52.9	36.7	2.83	95.2	33.7	20.5*	49.2	1.96	Twitter [51]
	FT	65.6M	3.61B	37	14,039,045	78.50%	3.10	129	41.6	6.44	260	40.5	10.9	122	2.07	Friendster [72]
Web	GG	876K	8.64M	24	175,274	73.31%	0.029	0.534	18.7	0.045	0.530	11.8	n	0.255	1.58	web-Google [53]
	SD	89.2M	3.88B	35	16,189,065	80.36%	3.11	134	43.2	5.61	213	38.0	n	92.3	1.81	sd_arc [54]
	CW	978M	74.7B	254	81,809,602	86.48%	22.9	1464	64.0	39.7	1526	38.4	n	695	1.73	ClueWeb [54]
	HL14	1.72B	124B	366	124,406,075	83.25%	31.1	2057	66.0	50.7	2113	41.7	n	1011	1.63	Hyperlink14 [54]
	HL12	3.56B	226B	650	410,853,262	80.63%	89.1	5435	61.0	104	5985	57.6	n	3027	1.17	Hyperlink12 [54]
Road	CA	1.97M	5.53M	857	381,366	79.55%	0.040	0.824	20.6	0.372	1.05	2.82	n	0.206	5.15	roadnet-CA [53]
	USA	23.9M	57.7M	8,263	7,390,330	66.90%	0.336	12.1	36.0	4.64	15.1	3.25	3.73*	2.25	6.69	RoadUSA [1]
	GE	12.3M	32.3M	2,240	2,482,488	78.67%	0.267	11.1	41.6	2.02	11.4	5.66	1.14*	2.88	7.54	Germany [1]
k-NN	HH5	2.05M	13.0M	1,859	17,408	62.55%	0.073	1.60	22.0	0.447	1.52	3.41	n	0.509	6.16	Household [38, 70], k=5
	CH5	4.21M	29.7M	14,479	299	15.41%	0.128	2.85	22.2	1.44	2.38	1.66	n	0.528	4.11	CHEM [41, 70], k=5
	GL2	24.9M	65.4M	13,333	10,940,922	0.03%	0.402	13.8	34.5	1.53	16.9	11.0	n	2.51	3.80	GeoLife [70, 73], k=2
	GL5	24.9M	157M	21,600	1,009,434	30.07%	0.472	19.1	40.5	2.80	19.4	6.92	n	4.03	5.93	GeoLife [70, 73], k=5
	GL10	24.9M	305M	3,824	51,465	86.38%	0.668	29.2	43.8	1.64	23.5	14.3	n	7.07	2.46	GeoLife [70, 73], k=10
	GL15	24.9M	453M	3,664	23,149	91.11%	0.751	34.4	45.8	1.51	25.9	17.1	n	8.92	2.01	GeoLife [70, 73], k=15
	GL20	24.9M	602M	2,805	13,619	93.96%	0.861	39.2	45.6	1.48	28.6	19.3	n	10.2	1.72	GeoLife [70, 73], k=20
	COS5	321M	1.96B	1,180	85,283	99.74%	8.46	382	45.2	17.5	392	22.4	n	120	2.07	Cosmo50 [52, 70], k=5
Synthetic	SQR	100M	400M	10,000	1	100.00%	1.32	43.4	32.9	15.4	44.2	2.87	20.3*	24.4	11.7	2D grid $10^4 \times 10^4$
	REC	100M	240M	50,500	1	100.00%	1.35	43.6	32.4	47.0	34.6	0.735	13.1*	16.8	12.5	2D grid $10^3 \times 10^5$
	SQR'	100M	400M	10,256	23,836,580	70.65%	1.31	50.1	38.1	12.5	60.9	4.88	n	10.6	8.06	sampled SQR
	REC'	100M	240M	69,014	23,826,514	70.66%	1.37	46.8	34.3	22.4	58.9	2.63	n	10.7	7.81	sampled REC
	Chn7	10M	20M	10^7-1	10^7-1	0.00%	0.278	13.1	46.9	81.6	19.7	0.241	40.5*	3.33	12.0	Chain of size 10^7
	Chn8	100M	200M	10^8-1	10^8-1	0.00%	3.25	152	46.9	957	307	0.320	703*	38.9	12.0	Chain of size 10^8

Table 2. Graph information, running times (in seconds), and speedups. $T_{best}/ours$ (highlighted in yellow) is the **fastest time of the other implementations / our time, both using all cores**. "n" = number of vertices. "m" = number of edges. "D" = approximate diameter. "#BCC" = number of BCCs. "$\lvert BCC_1\rvert$%" = percentage of the largest BCCs. "GBBS" = GBBS's implementation [31]. "SM'14" = Slota and Madduri's algorithm [64] (the faster of the two proposed algorithms). Since SM'14 has scalability issues (see Fig. 4), we report the 16-core time if it is faster, and denote as (*). "SEQ" = Hopcroft-Tarjan BCC algorithm [45]. Details about the baselines are introduced in Sec. 6. The fastest runtime for each graph is underlined. Red numbers are parallel runtime *slower than the sequential algorithm*. "par." = parallel running time (on 192 hyper-threads). "seq." = sequential running time (on 1 thread). "spd." = self-relative speedup. "n" = no support, because SM'14 only works on connected graphs.

The information of the graphs is given in Tab. 2. In addition to commonly-used benchmarks of social, web and, road graphs, we also use k-NN graphs and synthetic graphs. k-NN graphs are widely used in machine learning algorithms (see discussions in [70]). In k-NN graphs, each vertex is a multi-dimensional data point and has k edges pointing to its k-nearest neighbors (excluding itself). We also create six synthetic graphs, including two grids (SQR and REC), two sampled grids (SQR' and REC', each edge is created with probability 0.6), and two chains (Chn7 and Chn8). SQR and SQR' have sizes $10^4 \times 10^4$. REC and REC' have sizes $10^3 \times 10^5$. Each row and column in grid graphs are circular. Chn7 and Chn8 have sizes 10^7 and 10^8. The tested graphs cover a wide range of sizes and edge distributions.

For directed graphs, we symmetrize them to test BCC. We call the social and web graphs *low-diameter graphs* as they have diameters mostly within a few hundreds. We call the road, k-NN, and synthetic graphs *large-diameter graphs* as their diameters are mostly more than a thousand. When comparing the *average* running times across multiple graphs, we always take the ***geometric mean*** of the numbers.

Baseline Algorithms. We call all existing algorithms that we compare to the ***baselines***. We implement sequential Hopcroft-Tarjan [45] algorithm for comparison, referred to as SEQ. We compare the number of BCCs reported by each algorithm with SEQ to verify correctness.

We also compare to two most recent available BCC implementations GBBS [31], and Slota and Madduri [64]. We use SM'14 to denote the *better* of the two BCC algorithms in Slota and Madduri [64]. On many graphs, we observe that SM'14 is faster on 16 threads than using all 192 threads, in which case we report the lower time of 16 and 192 threads.

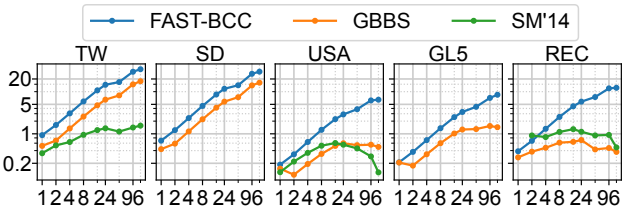

Figure 4. Scalability curves for different BCC algorithms. In each plot, x-axis is core counts (last data point is 96 core with hyperthreading) and y-axis is speedups normalized to SEQ (the sequential Hopcroft-Tarjan algorithm). Higher is better. SEQ is 1.

Through correspondence with the authors, we understand that SM'14 requires the input graph to be connected, so we only report the running time when it gives the correct answers. As few graphs we tested are entirely connected, we focus on comparisons with GBBS and SEQ. We also compare our breakdown and sequential running times with GBBS since GBBS can process most of the tested graphs[2].

Unfortunately, we cannot find existing implementations for Tarjan-Vishkin to compare with. We are aware of two papers that implemented Tarjan-Vishkin [29, 39]. Edwards and Vishkin's implementation [39] is on the XMT architecture and they did not release their code. Cong and Bader's code [29] is released, but it was written in 2005 and uses some system functions that are no longer supported on our machine. For a full comparison, we implemented a faithful Tarjan-Vishkin from the original paper [65]. As engineering Tarjan-Vishkin is not the main focus of this paper, we provide the details in the full paper.

We note that both GBBS and SM'14 exclude the postprocessing to compute the actual BCCs, but only report the number of BCCs at the end of the algorithm. We include this step in FAST-BCC, although this postprocessing only takes at most 2% of the total running time in all our tests.

6.1 Overall Performance

We present the running time of all algorithms in Tab. 2. Our FAST-BCC is ***faster than all baselines on all graphs***, mainly due to the theoretical efficiency—work- and space-efficiency enables competitive sequential times over the Hopcroft-Tarjan sequential algorithm, and polylogarithmic span ensures good speedup for all graphs.

Sequential Running Time. We first compare the *sequential* running time of SEQ, GBBS, and FAST-BCC. SEQ and FAST-BCC use $O(n+m)$ work. To enable parallelism, both FAST-BCC and GBBS traverse all edges multiple times (running CC twice in Steps 1 and 4, and computing low/high for the skeleton in Step 3). We describe more details about GBBS implementation in Sec. 6.2. On average, our sequential time is 2.8× slower than SEQ, but is 10% faster than GBBS.

[2]GBBS updated a new version after this paper was accepted, so we also updated the numbers using their latest version (Nov. 2022). Some new features in the latest version greatly improved their BCC performance.

Scalability and Parallelism. We report the scalability curves for FAST-BCC, GBBS and SM'14 on five graphs (Fig. 4). For fair comparison, the speedup numbers in Fig. 4 are normalized to the running time of SEQ. On these graphs, FAST-BCC is the only algorithm that scales to all processors. It outperforms GBBS and SM'14 on all graphs with all numbers of threads (except REC on 2 cores). We noticed that SM'14 suffers from scalability issues, and the best performance can be achieved at around 16 threads. Hence, we report SM'14's better running time of 16 and 192 threads in Tab. 2. GBBS has similar issues on a few graphs. However, as GBBS's performance does not drop significantly as core count increases, we consistently report GBBS's time on 192 threads in Tab. 2.

Our average self-relative speedups on both low-diameter graphs and large-diameter graphs are 36×. On large-scale low-diameter graphs with sufficient parallelism, the self-relative speedup can be up to 66×. Even on large-diameter graphs, FAST-BCC achieves up to 47× self-relative speedup. In comparison, the self-relative speedup of GBBS's BFS-based algorithm is 29× on low-diameter graphs and 3.7× on large-diameter graphs. This makes GBBS only 11% faster than SEQ on large-diameter graphs (and can be slower on some graphs), while ours is 5.1–18.5× better. Overall, our parallel running time is 10× faster on large-diameter graphs and 1.6× faster on low-diameter graphs than GBBS. On some graphs, SM'14 achieves better performance than GBBS, but FAST-BCC is 1.7–11.1× faster than SM'14 on all the graphs.

To verify that GBBS's performance is bottlenecked by BFS, we created k-NN graphs GL2–20 from the set of points but with different values of k. When increasing k over 5, the graphs have more edges but smaller diameters. For both FAST-BCC and SEQ, the running times increase when k grows due to more edges (and thus more work), but the trend of GBBS's running time is decreasing. This indicates that the BFS is the dominating part of running time for GBBS, and the performance on GBBS is bottlenecked by the $O(\textsc{Diam}(G)\log n)$ span.

6.2 Performance Breakdown

To understand the performance gain of FAST-BCC over prior parallel BFS-based BCC algorithms, we compare our performance breakdown with GBBS in Fig. 5. We choose GBBS because it can process all graphs. Since GBBS is also in the skeleton-connectivity framework, we use the same four step names for GBBS as in FAST-BCC, but there are a few differences. (1) For *First-CC*, FAST-BCC generates a spanning forest while GBBS only finds all CCs. (2) For *Rooting*, FAST-BCC uses ETT to root the tree while GBBS applies BFS on all CCs to find the spanning trees. (3) The task for *Tagging* is almost the same, but GBBS computes fewer tags than FAST-BCC since it is based on BFS trees. FAST-BCC uses 1D RMQ queries that are theoretically-efficient, while GBBS uses a bottom-up traversal on the BFS tree. (4) For *Last-CC*, both algorithms run CC algorithms on the skeletons to find BCCs.

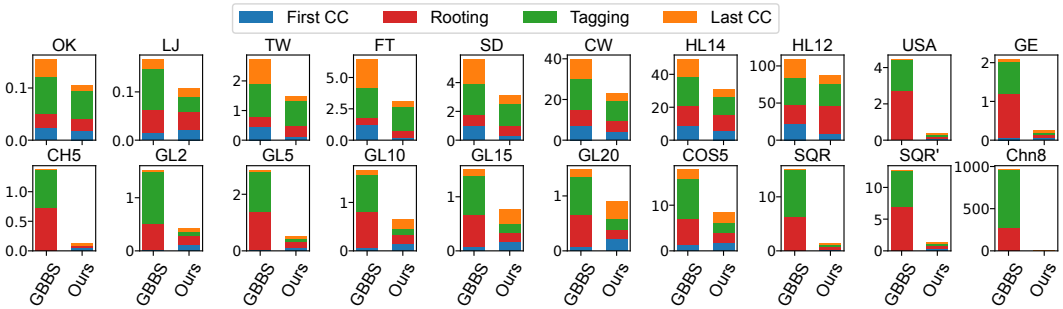

Figure 5. BCC breakdown. y-axis is the running time in seconds. The results for all the 27 graphs are in the full paper.

We first discuss the two steps *First-CC* and *Last-CC* that use connectivity. GBBS can be faster than FAST-BCC in *First-CC* on some graphs. The reason is that our algorithm also constructs the spanning forest in *First-CC*, while GBBS has to run BFS in *Rooting* to generate the BFS spanning forest. In *Last-CC*, the two algorithms achieve similar performance, and in many cases, FAST-BCC is faster. We note that the CC algorithm is independent with the BCC algorithm itself. Both the CC algorithm used in our implementation and GBBS is based on algorithms in an existing paper [32]. As mentioned, based on the results in [32], the "best" CC algorithm can be very different for different types of graphs. One can also plug in any CC algorithms to FAST-BCC or GBBS BCC algorithm to achieve better performance for specific input graphs.

In the *Rooting* step (*generate rooted spanning trees*, the red bar), FAST-BCC is significantly faster than GBBS. GBBS is based on a BFS tree, and after computing the CCs of input graph G, it has to run BFS on G again, which results in $O(m+n)$ work and $O(\textsc{Diam}(G)\log n)$ span. In comparison, FAST-BCC obtains the spanning trees from the *First-CC* step, and only uses ETT in the *Rooting* step with $O(n)$ expected work and $O(\log n)$ span *whp*. As shown in Fig. 5, this step for GBBS is the dominating cost for large-diameter graphs, and this is likely the case for other parallel BCC algorithms using BFS-based skeletons. FAST-BCC almost entirely saves the cost in this step (13× faster on average on large-diameter graphs). For low-diameter graphs, the two algorithms perform similarly—FAST-BCC is about 1.1× faster in this step.

In the *Tagging* step (the green bars), both FAST-BCC and GBBS compute the tags such as *low* and *high*. Since FAST-BCC uses an AST, the values of the arrays are computed using 1D range-minimum query (see Sec. 4.1) with $O(\log n)$ span. GBBS computes them by a bottom-up traversal on the BFS tree, with $O(\textsc{Diam}(G)\log n)$ span. Hence, on large-diameter graphs, GBBS also consumes much time on this step, and FAST-BCC is 1.2–830× faster than GBBS. On low-diameter graphs, GBBS also gets sufficient parallelism, and the performance for both algorithms is similar.

In summary, on all graphs, FAST-BCC is faster than GBBS mainly due to the efficiency in the *Rooting* and *Tagging* step, and the reason is that our algorithm has polylogarithmic span, while GBBS relies on the BFS spanning tree and requires $O(\textsc{Diam}(G)\log n)$ span.

6.3 The Tarjan-Vishkin Algorithm

Although engineering Tarjan-Vishkin (TV) [65] is not the focus of this paper, for completeness, we also implemented a faithful TV algorithm. Due to space limit, we give more details and report the numbers in the full paper, and summarize our findings here. Due to space inefficiency, our TV implementation cannot run on the three largest graphs (CW, HL14, and HL12) on our machines with 1.5TB memory. We note that the smallest among them (CW) only takes about 300GB to store the graph, and our algorithm uses 572GB memory to process it. On all graphs, TV uses 1.2-10.8× more space than FAST-BCC. GBBS is about 20% more space-efficient than FAST-BCC. The reason is that they need to compute fewer number of tags than FAST-BCC.

Regarding running time, we report the running time of TV on all graphs in the full paper, and summarize the results here. Due to the cost of explicitly constructing the skeleton, TV performs slowly on small-diameter graphs, and is slower than GBBS even on k-NN graphs. On all these graphs, the speedup for TV on 96 cores over SEQ is only 1.4–3×. This is consistent with the findings in prior papers [29, 64]. TV works well on road and synthetic graphs due to small edge-to-vertex ratio, so the $O(m)$ work and space for generating the skeleton does not dominate the running time. In this case, polylogarithmic span allows TV to perform consistently better than GBBS. TV is faster than SEQ on 96 cores on all graphs, but slower than FAST-BCC.

7 Conclusion

In this paper, we propose the FAST-BCC (Fencing on Arbitrary Spanning Tree) algorithm for parallel biconnectivity. FAST-BCC has $O(m+n)$ expected optimal work, polylogarithmic span (high parallelism), and uses $O(n)$ auxiliary space (space-efficient). The theoretical efficiency also enables high performance. On our machine with 96 cores and a variety of graph types, FAST-BCC outperforms all existing BCC implementations on all tested graphs.

Acknowledgement

This work is supported by NSF grants CCF-2103483 and IIS-2227669, and UCR Regents Faculty Fellowships. We thank anonymous reviewers for the useful feedbacks.

References

[1] 2010. OpenStreetMap © OpenStreetMap contributors. https://www.openstreetmap.org/.

[2] Alok Aggarwal, Bernard Chazelle, Leo Guibas, Colm Ó'Dúnlaing, and Chee Yap. 1988. Parallel computational geometry. *Algorithmica* 3, 1 (1988), 293–327.

[3] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. 2014. Provably Good Scheduling for Parallel Programs That Use Data Structures Through Implicit Batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[4] Daniel Anderson, Guy E Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), V2. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 445–447.

[5] Lars Arge, Michael Bender, Erik Demaine, Bryan Holland-Minkley, and Ian Munro. 2002. Cache-oblivious priority queue and graph algorithm applications. In *ACM Symposium on Theory of Computing (STOC)*. 268–276.

[6] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (01 Apr 2001).

[7] Giorgio Ausiello, Donatella Firmani, and Luigi Laura. 2011. Real-time anomalies detection and analysis of network structure, with application to the Autonomous System network. In *International Wireless Communications and Mobile Computing Conference*. IEEE, 1575–1579.

[8] Christian Bachmaier, Franz J Brandenburg, and Michael Forster. 2005. Radial level planarity testing and embedding in linear time. In *J. Graph Algorithms and Applications*. Citeseer.

[9] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 44–54.

[10] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2016. Parallel Algorithms for Asymmetric Read-Write Costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[11] Naama Ben-David, Guy E. Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. Implicit Decomposition for Write-Efficient Connectivity Algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[12] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib-a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

[13] Guy E. Blelloch, Rezaul Alam Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. 2008. Provably good multicore cache performance for divide-and-conquer algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[14] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling Irregular Parallel Computations on Hierarchical Caches. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[15] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[16] Guy E. Blelloch and Phillip B. Gibbons. 2004. Effectively sharing a cache among threads. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[17] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[18] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2018. Parallel Write-Efficient Algorithms and Data Structures for Computational Geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[19] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Randomized Incremental Convex Hull is Highly Parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[20] Guy E. Blelloch and Margaret Reid-Miller. 1998. Fast Set Operations Using Treaps. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[21] Guy E. Blelloch and Margaret Reid-Miller. 1999. Pipelining with futures. *Theory of Computing Systems (TOCS)* 32, 3 (1999).

[22] Guy E. Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Parallel and I/O efficient set covering algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[23] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998).

[24] John M Boyer and Wendy J Myrvold. 2006. Simplified $o(n)$ planarity by edge addition. *J. Graph Algorithms and Applications* 5 (2006), 241.

[25] Meher Chaitanya and Kishore Kothapalli. 2015. A simple parallel algorithm for biconnected components in sparse graphs. In *International Parallel and Distributed Processing Symposium (IPDPS) Workshop*. IEEE, 395–404.

[26] Meher Chaitanya and Kishore Kothapalli. 2016. Efficient multicore algorithms for identifying biconnected components. *International Journal of Networking and Computing* 6, 1 (2016), 87–106.

[27] Joseph Cheriyan and Ramakrishna Thurimella. 1991. Algorithms for parallel k-vertex connectivity and sparse certificates. In *ACM Symposium on Theory of Computing (STOC)*. 391–401.

[28] Yi-Jen Chiang, Michael Goodrich, Edward Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Vitter. 1995. External-Memory Graph Algorithms.. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Vol. 95. 139–149.

[29] Guojing Cong and David Bader. 2005. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition)*. MIT Press.

[31] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)* 8, 1 (2021), 1–70.

[32] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: a framework for static and incremental parallel graph connectivity algorithms. *Proceedings of the VLDB Endowment (PVLDB)* 14, 4 (2020), 653–667.

[33] Laxman Dhulipala, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E Blelloch, Phillip B Gibbons, and Julian Shun. 2020. Semi-Asymmetric Parallel Graph Algorithms for NVRAMs. *Proceedings of the VLDB Endowment (PVLDB)* 13, 9 (2020).

[34] David Dinh, Harsha Vardhan Simhadri, and Yuan Tang. 2016. Extending the nested parallel model to the nested dataflow model with provably efficient schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 49–60.

[35] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[36] Xiaojun Dong, Letong Wang, Yan Gu, and Yihan Sun. 2022. FAST-BCC: A Parallel Implementation for Graph Biconnectivity. https://github.com/ucrparlay/FAST-BCC.

[37] Xiaojun Dong, Letong Wang, Yan Gu, and Yihan Sun. 2023. Provably Fast and Space-Efficient Parallel Biconnectivity. *arXiv preprint:2301.01356* (2023).

[38] Dheeru Dua and Casey Graf. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/.

[39] James A Edwards and Uzi Vishkin. 2012. Better speedups using simpler parallel programming for graph connectivity and biconnectivity. In *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. 103–114.

[40] Xing Feng, Lijun Chang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Long Yuan. 2018. Distributed computing connected components with linear communication cost. *Distributed and Parallel Databases* 36, 3 (2018), 555–592.

[41] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.

[42] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel Cover Trees and their Applications. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 259–272.

[43] Yan Gu, Omar Obeya, and Julian Shun. 2021. Parallel In-Place Algorithms: Theory and Practice. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 114–128.

[44] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 24–34.

[45] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM* 16, 6 (1973), 372–378.

[46] John Hopcroft and Robert Tarjan. 1974. Efficient planarity testing. *J. ACM* 21, 4 (1974), 549–568.

[47] Joseph JáJá. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[48] Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. 2017. Parallel algorithm for incremental betweenness centrality on large graphs. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2017), 659–672.

[49] Siddhartha Jayanti, Robert E Tarjan, and Enric Boix-Adserà. 2019. Randomized concurrent set union and generalized wake-up. In *ACM Symposium on Principles of Distributed Computing (PODC)*. 187–196.

[50] Yuede Ji and H Howie Huang. 2020. Aquila: Adaptive parallel computation of graph connectivity queries. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 149–160.

[51] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *International World Wide Web Conference (WWW)*. 591–600.

[52] YongChul Kwon, Dylan Nunley, Jeffrey P Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. 2010. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *International Conference on Scientific and Statistical Database Management*. Springer, 132–150.

[53] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[54] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2014. Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph.

[55] Gary L Miller, Richard Peng, and Shen Chen Xu. 2013. Parallel graph decompositions using random shifts. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[56] MEJ Newman and Gourab Ghoshal. 2008. Bicomponents and the robustness of networks to failure. *Physical review letters* 100, 13 (2008), 138701.

[57] John H Reif. 1985. Depth-first search is inherently sequential. *Inform. Process. Lett.* 20, 5 (1985), 229–234.

[58] John H. Reif. 1993. *Synthesis of Parallel Algorithms*. Morgan Kaufmann.

[59] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit Çatalyirek. 2013. Incremental algorithms for closeness centrality. In *IEEE International Conference on Big Data*. IEEE, 487–492.

[60] Ahmet Erdem Sariyüce, Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. 2013. Shattering and compressing networks for betweenness centrality. In *Proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM, 686–694.

[61] Carla Savage and Joseph JáJá. 1981. Fast, efficient parallel algorithms for some graph problems. *SIAM J. on Computing* 10, 4 (1981), 682–691.

[62] Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. 2022. Many Sequential Iterative Algorithms Can Be Parallel and (Nearly) Work-efficient. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[63] Julian Shun, Laxman Dhulipala, and Guy Blelloch. 2014. A Simple and Practical Linear-work Parallel Algorithm for Connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[64] George M Slota and Kamesh Madduri. 2014. Simple parallel biconnectivity algorithms for multicore platforms. In *IEEE International Conference on High Performance Computing (HiPC)*. IEEE, 1–10.

[65] Robert E Tarjan and Uzi Vishkin. 1985. An efficient parallel biconnectivity algorithm. *SIAM J. on Computing* 14, 4 (1985).

[66] Yung H Tsin and Francis Y Chin. 1984. Efficient parallel algorithms for a class of graph theoretic problems. *SIAM J. on Computing* 13, 3 (1984), 580–599.

[67] Uzi Vishkin. 1985. On efficient parallel strong orientation. *Inform. Process. Lett.* 20, 5 (1985), 235–240.

[68] Mihir Wadwekar and Kishore Kothapalli. 2017. A fast GPU algorithm for biconnected components. In *International Conference on Contemporary Computing (IC3)*. IEEE, 1–6.

[69] Letong Wang, Xiaojun Dong, Yan Gu, and Yihan Sun. 2022. Parallel Strong Connectivity Based on Faster Reachability. In *manuscript*.

[70] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2021. GeoGraph: A Framework for Graph Processing on Geometric Data. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 38–46.

[71] Yifan Xu, Anchengcheng Zhou, Grace Q Yin, Kunal Agrawal, I-Ting Angelina Lee, and Tao B Schardl. 2022. Efficient Access History for Race Detection. In *Algorithm Engineering and Experiments (ALENEX)*. SIAM, 117–130.

[72] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[73] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw gps data for geographic applications on the web. In *International World Wide Web Conference (WWW)*. 247–256.

A Artifact Description

A.1 Availability

Our artifact is available on Zenodo: https://doi.org/10.5281/zenodo.7445964. Our code is also released on GitHub: https://github.com/ucrparlay/FAST-BCC. Scripts for downloading the dataset and running the code, including a README file on how to use them are given on the GitHub repo.

A.2 Requirements

- Hardware: any modern (2010+) x86-based multi-core (ideally more than 32 physical cores) Intel machines. 64GB memory is required for the basic datasets (without the three largest graphs). Running the complete dataset requires about 1.5TB main memory.
- Software: Linux machines with `gcc` or `clang` supporting `C++17` features.

A.3 Getting the Artifact

Our code is also publicly available on GitHub: https://github.com/ucrparlay/FAST-BCC. Our source code can be acquired using:

```
git clone --recurse-submodules https://github.com/
    ucrparlay/FAST-BCC.git
```

A.4 Download the Dataset

We provide the basic datasets on our Google Drive: https://tinyurl.com/FAST-BCC-Dataset. We also provide a script to download the graphs.

```
$ cd scripts/
$ ./download_dataset.sh
```

Our complete dataset contains some large graphs, which is about 2TB data in total. Due to storage limit, we cannot provide the largest three graphs tested in the paper. They are available on Web Data Commons [54].

A.5 Running the benchmark

To run the FAST-BCC and Hopcroft-Tarjan, simply run:

```
$ cd scripts/
$ ./run_fastbcc.sh
$ ./run_fastbcc_sequential.sh
$ ./run_hopcroft_tarjan.sh
$ ./run_tarjan_vishkin.sh
```

The scripts will generate the results in CSV format in the **results/** folder. The running times and the number of biconnected components are reported in these files.

A.6 Graph Formats

The application can auto-detect the format of the input graph based on the suffix of the filename. Here is a list of supported graph formats:

- `.bin`: The binary graph format from GBBS. It uses compressed sparse row (CSR) format and organizes as follows:
  - `n`: the number of vertices (64-bit variable);
  - `m`: the number of edges (64-bit variable);
  - `size`: the size of this file in bytes, which equals to $3 \times 8 + (n-1) \times 8 + m \times 4$ (64-bit variable);
  - `offset[]`: `offset[i]` (inclusive) and `offset[i+1]` (exclusive) represents the range of neighbors list of the i-th vertex in the edges array (64-bit array of length $n+1$);
  - `edges[]`: the edges list (32-bit array of length m) and `edges[i]` is a vertex id representing the other endpoint of an edge.
- `.adj` The adjacency graph format from Problem Based Benchmark Suite (PBBS) [4].

Some graphs in binary format can be found in our Google Drive folder. See more details in Section A.4. The input graphs need to be undirected, i.e., each edge should appear twice in the input in both directions.