Achieving Speedups for Distributed Graph Biconnectivity

Ian Bogle
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York, USA
boglei@rpi.edu

George M. Slota
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York, USA
slotag@rpi.edu

***Abstract*—As data scales continue to increase, studying the porting and implementation of shared memory parallel algorithms for distributed memory architectures becomes increasingly important. We consider the problem of biconnectivity for this current study, which identifies cut vertices and cut edges in a graph. As part of our study, we implemented and optimized a shared memory biconnectivity algorithm based on color propagation within a distributed memory context. This algorithm is neither work nor time efficient. However, when we compare to distributed implementations of theoretically efficient algorithms, we find that simple non-optimal algorithms can greatly outperform time-efficient algorithms in practice when implemented for real distributed-memory environments and real data. Overall, our distributed implementation for computing graph biconnectivity demonstrates an average strong scaling speedup of 15× across 64 MPI ranks on a suite of irregular real-world inputs. We also note an average of 11× and 7.3× speedup relative to the optimal serial algorithm and fastest shared-memory implementation for the biconnectivity problem, respectively.**

***Index Terms*—parallel algorithms, graph algorithms, biconnectivity**

I. Introduction

The large scale of modern datasets representing real-world graphs and meshes for scientific computations often necessitates the use of distributed memory processing. Likewise, efficiently solving basic graph problems such as a graph biconnectivity decomposition remains important to multiple applications, including climate modeling [1], network resilience [2], social network analysis [3], and other scientific computing applications [4]. However, many graph algorithms, including biconnectivity, are not widely studied in distributed memory, and many do not have in-practice performant distributed implementations described in the existing literature.

Graph biconnectivity is often described as identifying *cut vertices* in a connected graph – vertices that, when removed, will disconnect the graph. This can be considered a specific instance of a more general connectivity problem, k-vertex connectivity, where k corresponds to the minimum number of vertices that need to be removed from a graph in order to disconnect it. We note that there exists time-efficient shared memory algorithms for biconnectivity (i.e., 1-vertex connectivity), but as the value of k increases, algorithm complexity correspondingly does as well. Algorithms for general k-vertex connectivity often use flow-based approaches [5], while more specific algorithms for $k \leq 3$ connectivity use a wide variety of subroutines.

We restrict this current study to graph biconnectivity in the context of two algorithms, both of which were originally proposed for shared memory computation. First, we consider the well-known time-optimal Tarjan and Vishkin algorithm [6] and its Cong and Bader extension (TV-Filter) [7]. Other recent work [8, 9], as well as our own substantial preliminary work, has attempted to distribute the Tarjan-Vishkin algorithm. However, none of these attempts have produced an implementation that offers much, if any, speedup relative to the serial algorithm running on a single thread of a consumer laptop. As such, we primarily consider the color-propagation algorithm (Color-BiCC) of Slota and Madduri [10]. This algorithm was original proposed as a "simple" alternative to Tarjan-Vishkin, and it is neither time-optimal nor work-optimal. Regardless, as our results will show, this simplicity can in practice result in a scalable state-of-the-art distributed implementation.

This idea of tradeoffs in terms of implementation simplicity versus theoretical optimality, as well as the general idea of studying how to efficiently port an algorithm from shared to distributed memory, are key motivators for this current work.

A. Our Contributions

Our primary contribution is the distributed implementation of the Slota-Madduri Color-BiCC algorithm in distributed-memory. In addition, we also implement the edge filtering technique of Cheriyan and Thurimella [11] to greatly reduce the necessary number of edges needed to determine cut vertices and edges in some input graph. Our specific contributions are noted below:

1) We port the Slota-Madduri Color-BiCC algorithm and the Cheriyan-Thurimella edge filtering algorithms to distributed memory. These are the **first distributed-memory implementations** of these algorithms presented in the literature.
2) Overall, our distributed implementation of the Color-BiCC algorithm demonstrates average speedups of 11× and 7.3× relative to the optimal serial Hopcroft-Tarjan algorithm [12] and the shared memory Color-BiCC

algorithm, respectively, on 64 MPI ranks. Our implementation is also the **first to demonstrate considerable speedups relative to the serial algorithm** for distributed biconnectivity processing.

II. Background

Here we will discuss general distributed graph processing, explicitly define the graph biconnectivity problem, and present several prior and relevant works.

A. Distributed Graph Processing

We consider a graph $G = (V, E)$, where V is the vertex set and E is the edge set. In our discussions, we will use $n = |V|$ and $m = |E|$. For our distributed implementation, we explicitly consider a 1-dimensional distributed graph processing model, where our input graph $G = (V, E)$ is broken into some number of vertex-disjoint subgraphs $G = (\{V_1, V_2, \ldots V_p\}, \{E_1, E_2, \ldots E_p\})$ that are distributed to p computational ranks. Each of these subgraphs contain a set of *owned* vertices and all of their incident edges. Consequently, these subgraphs also include all non-owned vertices (i.e., *ghosted vertices*) within the 1-hop neighborhood of owned vertices. Each rank holds vertex state information for both owned and ghosted vertices.

For all of our implementations, we enforce a $O(\frac{n+m}{p})$ memory bound. In other words, no single rank (assuming $p > 1$) can view *all* vertex or edge information or states. The assumption being, under this distributed model, we can theoretically strong scale any input graph or process any input of arbitrarily large scale given a sufficient number of compute nodes. Note however, in practice with a 1D distribution, each V_i, V_j and E_i, E_j are not going to be perfectly balanced, so there is a practical limit.

B. Biconnectivity Definitions

We consider the general problems of k-connectivity and the specific problem of biconnectivity. A graph G is said to be k-connected if the minimum size of a vertex separator (a subset of vertices $S \subseteq V$ such that $G - S$ is disconnected) is equal to k; i.e., $|S| = k$. Most computational formulations for k-connectivity seek to identify all such minimum separators.

Many graph biconnectivity algorithms are described as performing a *biconnectivity decomposition*, which is an edge-labeling of some connected G such that all maximal 2-connected subgraphs have edges with the same label. Biconnectivity algorithms often in practice simply seek to identify all edges e such that $G - e$ is disconnected (cut edges or *bridges*) and all vertices v such that $G - v$ is disconnected (cut vertices or *articulation vertices*). With articulation vertices and bridges denoted, producing the desired edge labeling is relatively trivial.

C. Least Common Ancestor Traversal

Many connectivity algorithms, including one we implement in this paper, use least (or *lowest*) common ancestors (LCA) of a rooted spanning tree. Given two vertices u, v in a rooted tree, the least common ancestor is the first vertex that is contained within both of u and v's paths to the root (the root itself could be that ancestor). If u and v are joined by a tree-edge, then the parent of the pair is the LCA.

D. Prior Biconnectivity Work

Due to their wide applicability across a broad number of fields, biconnectivity algorithms have been studied for many decades. Hopcroft and Tarjan provided the first optimal serial algorithm [12], which is based on depth-first search and requires a single traversal with work $O(n + m)$.

Later, Tarjan and Vishkin [6] described the first time-optimal parallel algorithm for biconnectivity. Their algorithm runs in $O(\log n)$ time on $O(n + m)$ processors, but includes several non-trivial stages of computation, including spanning tree construction, preorder labeling, subtree size enumeration, and connectivity checking. Cong and Bader [7] incorporated Cheriyan and Thurimella [11] edge filtering as a preprocessing step to provide an optimized implementation of the Tarjan-Vishkin algorithm, *TV-Filter*. Edge filtering identifies which edges in some input graph are actually necessary to identify the k-separators of the graph, and it can speedup k-connectivity algorithms by eliminating processing of a majority of edges. Cong and Bader showed their TV-Filter implementation achieved a $2\times$ speedup over an optimized parallel implementation of baseline Tarjan-Vishkin.

Slota and Madduri [10] studied simple shared memory approaches for biconnectivity by developing algorithms using breadth-first search (BFS) and color propagation as primary subroutines. These subroutines have been extensively studied in practice. Among many other works in the literature, Buluç et al. [13] examined BFS implementations for massive graphs in distributed memory, while Stergiou et al. [14] considered time-optimal distributed color propagation for connectivity[1]. More recently, Chaitanya et al. [15] further developed the Slota-Madduri biconnectivity ideas by using LCA traversals in shared memory and a more complex parallelization scheme.

Distributed Algorithms: There have been a small number of works that have considered the computation of graph biconnectivity in a distributed memory setting. Ahmadi and Stone [16] and Kazmierczak and Radhakrishnan [17] are two such works, though their proposed algorithms run in linear time. Bogle et al. [1] presented a specialized algorithm that performed a decomposition similar to biconnectivity on ice sheet meshes and then later [18] generalized that approach for the actual biconnectivity problem. Another avenue of prior work [8, 9] has considered algorithms for use within a vertex-centric processing system, such as Pregel. These approaches have considered optimized implementations of the Tarjan-Vishkin algorithm as well as proposed algorithms that use variations of ear decompositions. However, the common theme with all of the current distributed biconnectivity implementations in the literature is that **they lack consistent speedups relative to the optimal serial algorithm**.

[1]This is among many other works, as BFS and connectivity are possibly the two most-studied problems in graph processing.

III. COLOR-BICC IMPLEMENTATION

As noted, our primary effort in this work was distributing the Color-BiCC algorithm of Slota and Madduri [10]. We also considered the Slota-Madduri BFS-based algorithm from the same reference and the algorithm of Chaitanya and Kothapalli [15]. However, the Slota-Madduri BFS algorithm utilizes many concurrent thread-owned traversals, which would be costly to distribute, and the Chaitanya-Kothapalli algorithm uses task-based parallelism, which also does not lend itself to an easy distributed formulation.

Algorithm 1 Slota-Madduri Color-BiCC Algorithm

```
procedure COLOR-BICC($G = (V, E)$)
    $T$ = BFS($G$)                          ▷ Compute spanning tree via BFS
    $High, Low$ = LCA($G, T$)                ▷ Initialize labels
    $A$ = Color($G, High, Low, T$)           ▷ Propagate labels
    $B$ = LabelEdges($G, A$)
    return $B$
```

Given in Algorithm 1, the Color-BiCC algorithm has three primary phases. The first phase computes a rooted spanning tree. The second phase initializes per-vertex $High$ and Low label values using LCA traversals of the tree. The final phase propagates these labels such that, at the conclusion of propagation, each vertex v will have a $High$ label with a value of the nearest articulation vertex that will disconnect v from the root (or, the label of the root itself). The algorithm uses these labels to determine and then return a set of articulation vertices A. These articulation vertices are then used to label the edges of G for a biconnectivity decomposition.

A. BFS and LCA Implementations

We use an efficient BFS implementation modified and updated from our prior work [19] to get the rooted spanning tree T. T is implicitly created via $Parents$ and $Levels$ arrays that defines each vertex's parent and its distance from the root. The LCA phase uses these along with traversals up the tree to initialize $High$ and Low values. Specifically, each vertex will have their $High$ value initialized to the lowest-level ancestor that it has in common with one of its non-tree neighbors. Here, "lowest level" refers to being closer to the root. Low values for each vertex are set to the lowest numeric vertex identifier among that vertex and all of its non-tree neighbors.

Our distributed LCA implementation is given in Algorithm 2. We use a distributed queue-based approach for communication. This queue is comprised of *packages* that contain the two vertices that the LCA traversal originated from (u and v) and the current location of this traversal (stored as $high$ and low along with the current levels). During each superstep, packages are processed with traversals progressing by following the values in $Parents$ arrays. When $high$ and low are equal, it indicates a least common ancestor has been found. $High(u)$ and $High(v)$ can then be updated to these values if the current level for $high$ is closer to the root than current vertices indicated in $High(u)$, $High(v)$. This update can happen immediately if u, v are local to the processing rank; otherwise, a final communication occurs to transmit the result to the owning ranks.

Algorithm 2 Distributed Memory LCA Phase of Color-BiCC

```
1:  procedure LCA($G, T = (Parents, Levels)$)
2:      $Q \leftarrow \emptyset$, $Q_n \leftarrow \emptyset$                    ▷ Traversal queues
3:      for all $(v, u) \notin T$ do in parallel           ▷ Non-tree edges
4:          $Q \leftarrow \{Parents(u), u, Levels(u), Parents(v), v, Levels(v)\}$
5:      for all $v \in V(G)$ do in parallel
6:          $High(v) \leftarrow Parents(v)$
7:          $Low(v) \leftarrow$ lowest ID $u \in N(v)$, where$(v, u) \notin T$
8:      while $Q$ is not empty on some process do
9:          for all $package \in Q$ do in parallel
10:             $high \leftarrow$ higher level $Parents$ in $package$
11:             $low \leftarrow$ lower or equal level $Parents$ in $package$
12:             $u, v \leftarrow$ vertices in $package$
13:             $level_u, level_v \leftarrow$ levels in $package$
14:             if $level_u = null$ then              ▷ Received final value
15:                 Update $High(u)$
16:             else if $low = high$ then   ▷ Common ancestor found
17:                 if $rank(u) \neq this\_rank$ then
18:                     $package \leftarrow \{u, high, null, \ldots\}$
19:                     $Q_n$.insert($package$)
20:                 else           ▷ Can set final value without comm.
21:                     Update $High(u)$
22:                 if $rank(v) \neq this\_rank$ then
23:                     $package \leftarrow \{v, high, null, \ldots\}$
24:                     $Q_n$.insert($package$)
25:                 else
26:                     Update $High(v)$
27:             else                         ▷ Take next step in traversal
28:                 $package \leftarrow \{Parents(high), u, level_u - 1,$
29:                                 $Parents(low), v, level_v - 1\}$
30:                 $Q_n$.insert($package$)
31:         Exchange $Q_n$
32:         Swap($Q, Q_n$), $Q_n \leftarrow \emptyset$
33:     return $High$, $Low$
```

B. Color Propagation Implementation

The final phase of Color-BiCC is the iterative propagation of $High$ and Low values following a set of propagation rules, demonstrated in Algorithm 3. We slightly modify the original implementation of Color-BiCC, which utilized a "push" style of propagation, where vertex v would modify its neighbor u's values. Within a distributed context and having much wider parallelism, we use a "pull" style of propagation, where vertex v only modifies its own values based on its neighbors values. Otherwise, our propagation rules are consistent with the original algorithm.

We utilize a queue-based strategy with a boundary exchange on each iteration for communication of label values. We track which vertices are currently in the queue to avoid adding the same vertex multiple times. For our boundary exchange, we pass each owned vertex v along with $High(v)$ and $Low(v)$. The receiving ranks of v will have v as a ghost vertex. In addition to updating their local values for v, the receiving rank will also place all owned neighbors of v in the queue for processing. As such, the active set of vertices for processing in a given iteration are only the vertices who have had some

Algorithm 3 Distributed Memory Coloring Phase of Color-BiCC

```
procedure COLOR($G, High, Low, T = (Parents, Levels)$)
    $Q \leftarrow V(G), Q_n \leftarrow \emptyset$
    while $Q$ is not empty on some process do
        for all $v \in Q$ do in parallel
            for all $(v, u) \in E(G)$ do
                if $High(v) = v$ then
                    continue
                if $Levels(High(u)) > Levels(High(v))$ then
                    $High(v) \leftarrow High(u)$
                if $Levels(High(u)) = Levels(High(v))$ then
                    if $High(u) > High(v)$ then
                        $High(v) \leftarrow High(u)$
                if $High(u) = High(v)$ then
                    if $Low(u) < Low(v)$ then
                        $Low(v) \leftarrow Low(u)$
                if $High(v)$ or $Low(v)$ changed and $v \notin Q$ then
                    $Q_n$.insert($u$)
        Exchange $High, Low$ for all $v \in Q_n$
        Swap($Q, Q_n$), $Q_n \leftarrow \emptyset$
        for all $v \in Q$ do in parallel
            for all $v, u \in E(G)$ do
                if $u \notin Q$ then
                    $Q$.insert($u$)
    $A \leftarrow$ all unique values in $High$
    return $A$
```

value change in their neighborhood on the previous iteration. This optimization is particularly important for graphs such as web graphs, which can have a high diameter and long tails of iterations where only a few updates are processed per iteration. The final set of articulation vertices A are all unique vertex values stored in $High(v)$ across all $v \in V(G)$.

Queuing Strategy: As noted, we use a queue-based approach for processing and communication. We maintain an *active* set of vertices – these vertices had their or their neighbor's label update on the prior iteration. Likewise, many of our implementations utilize a multi-level queue to minimize necessary atomics, where each processing thread owns a small queue, which they push to a rank-level queue when it fills, while the rank-level queue is what is used during communication boundary exchanges with other ranks. We do not show this low-level of granularity in our algorithm listings for space considerations, but we note it here in text because it is an important performance optimization.

C. Discussion of Color-BiCC

We note that none of the distributed subroutines we developed for Color-BiCC are theoretically optimal. In fact, all of these three subroutines are dependent on graph diameter within the BSP model. However, as one will be able to observe in our results, this is not restrictive on performance in practice on real graphs. In fact, we observe consistent speedups across all tests for our implementation and very fast performance relative to other distributed implementations and the optimal serial algorithm. This alludes to our primary takeaway for this work: in real-world environments, implementation simplicity can often trump theoretic efficiency by a considerable degree.

IV. Cheriyan-Thurimella Edge Filtering

As mentioned, the Cong-Bader TV-Filter algorithm initially reduces the number of edges needed to determine biconnectivity on some G. We additionally incorporate this optimization into our implementation. This edge filtering is possible via the interesting result of Cheriyan and Thurimella [11], which states that the k-connectivity of some connected graph G is equivalent (in terms of separators) to the k-connectivity of some G' where $G' = T \cup F_1 \cup ... \cup F_k$. Here T is a spanning tree of G and F_i is a spanning forest of $G - T - F_1 - \ldots - F_{i-1}$. As biconnectivity decompositions seek to identify separators of size 1, we can simply construct $G' = T \cup F_1$. We observe that this procedure can greatly reduce the number of edges needing processed during a biconnectivity decomposition. In practice, we got reductions in graph sizes from 3-10$\times$ for the graphs we used for our experiments.

Algorithm 4 Cheriyan-Thurimella Edge Filtering

```
procedure FILTER($G = (V, E)$)
    $T$ = BFS($G$)
    $C$ = ConnectedComponents($G - T$)
    $F$ = BFS($G, C$)
    return $T \cup F$
```

Our approach to distributed edge filtering is given in Algorithm 4. We use two primary subroutines, including a single/multi-source BFS and an implementation of connected components. For BFS, we reuse an efficient implementation that has been updated from our prior work [19]. Our connected components implementation uses label propagation and is given in Algorithm 5. Our connected components algorithm initializes component labels for all vertices to be their global vertex identifier. Until convergence, each vertex iteratively updates their label to the lowest label in its neighborhood. As with the final phase of Color-BiCC, this algorithm relies heavily on the queue-based strategy described above.

A. Discussion of Cheriyan-Thurimella

We also considered two other approaches for implementing the BFS, connectivity, and spanning tree stages of edge filtering. Firstly, we implemented time-optimal algorithms that use techniques such as the distributed-memory equivalent of pointer-jumping. However, we actually observed "worse" performance with these implementation, likely due to the larger work complexity and high communication costs associated with such techniques in distributed memory. Secondly, while source code for optimized algorithms for these routines can be found online, interoperability concerns with our existing codebase would have necessitated implementations to also be from scratch. Since our existing approach is observed to be quite scalable as-is, we decided to keep our approach as given above (and retain our consistent theme of "simplicity").

Algorithm 5 Connected Components

```
procedure ConnectedComponents($G = (V, E)$)
    for all $v \in V(G)$ do in parallel
        $C(v) \leftarrow$ VertexID($v$)
    $Q \leftarrow V(G), Q_n \leftarrow \emptyset$
    while $Q$ is not empty on some rank do
        for all $v \in Q$ do in parallel
            for all $(v, u) \in E(G)$ do
                if $C(v) > C(u)$ then
                    $C(v) \leftarrow C(u)$    ▷ Propagate lower labels
                    $Q_n$.insert($v$)
            for all $(v, u) \in E(G)$ do
                if $C(v)$ changed then
                    $Q_n$.insert($u$)
        Exchange $C(v)$ for all boundary $v \in Q_n$
        Update $C(v)$ for all received values
        Swap($Q, Q_n$), $Q_n \leftarrow \emptyset$
    return $C$
```

V. Implementation Details

We implement all of our methods in C++ using MPI and OpenMP for distributed and shared-memory parallelism. For the graph structure, communication routines, and certain subroutines, we use and modify code available from HPCGraph[2]. We use a 1D vertex distribution, which we determine via a naïve hash-based approach. Our processing methodology closely follows a BSP approach, with computation/communication supersteps. Communications mostly consists of boundary exchanges of queue data and ghost vertex states in an all-to-all fashion.

VI. Experimental Setup

We run our experiments on well-known datasets with a variety of scales and topologies. The datasets are given in Table II. We consider the underlying graphs for any directed graphs. We also preprocess all networks to extract the largest connected component and remove duplicate edges. Our graphs primarily come from the Stanford Large Network Dataset Collection[3], the Network Data Repository[4], and the Koblenz Network Collection[5]. We additionally include a random scale-25 R-MAT graph generated with parameters $\{A = 0.45, B = 0.15, C = 0.15, D = 0.25\}$.

For experimentation, we consider runs on RPI's AiMOS supercomputer. AiMOS has nodes with $2\times$ 20-core 3.15 GHz IBM Power 9 CPUs and 512 GB DDR which are connected by 100 Gb EDR Inifiniband. Our build and execution environment on AiMOS includes Spectrum MPI 10.4 and XL 16.1.1 running on RHEL 8.4. We compare directly to the code openly available for Color-BiCC[6]. For additional comparisons, we implemented our own version of the serial Hopcroft-Tarjan algorithm.

[2]https://github.com/HPCGraphAnalysis/HPCGraph
[3]http://snap.stanford.edu/data/index.html
[4]https://networkrepository.com/index.php
[5]http://konect.cc/
[6]https://www.cs.rpi.edu/~slotag/soft/BiCC-HiPC14.tar

TABLE I
Graphs used for experiments and their properties in terms of the number of vertices $|V|$ and edges $|E|$ after preprocessing as well as the approximate diameter D and number of biconnected components (#BiCCs).

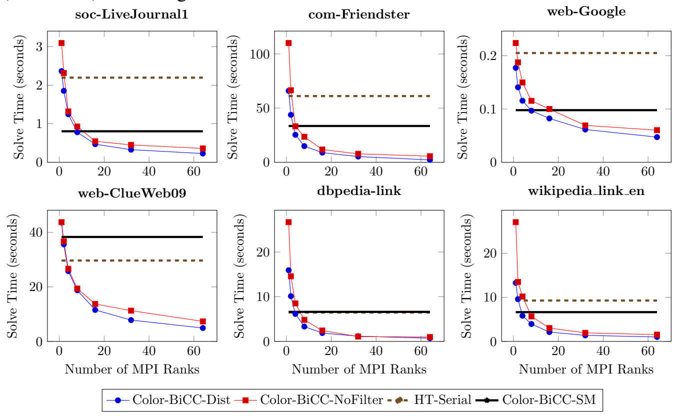
Graph Name	Type	$\|V\|$	$\|E\|$	D	#BiCCs	Ref.
soc-LiveJournal1	Social	4.8 M	43 M	46	76 K	[20]
com-Friendster	Social	52 M	1.1 B	35	5.5 M	[21]
web-Google	Web	855 K	4.3 M	25	60 K	[20]
web-ClueWeb09	Web	225 M	1.0 B	40	15 M	[22]
dbpedia-link	Info.	18 M	127 M	13	2.8 M	[23]
wikipedia_link_en	Info.	14 M	335 M	12	1.9 M	[23]
RMAT_25	Random	34 M	537 M	11	174 K	[24]

To better analyze distributed scaling behavior on a limited number of nodes, we run one MPI rank per socket on AiMOS for our experiments. We have generally observed the fastest execution times with only 1 thread per code for all algorithms, so we fix our thread count per rank equal to the core count per socket for all experiments. As such, we run 2 MPI ranks per each AiMOS node and 20 OpenMP threads per rank.

VII. Results

Figure 1 gives the strong scaling performance of our Color-BiCC implementation from 1 to 64 ranks on AiMOS. We give the summed time of Color-BiCC and edge filtering (Color-BiCC-Dist), the time of Color-BiCC without edge filtering (Color-BiCC-NoFilter), the time of the serial Hopcroft-Tarjan algorithm (HT-Serial), and the time of Slota and Madduri's shared memory code running on a single socket (Color-BiCC-SM). We run on only a single socket (20 threads) to enable relative comparisons to a single rank of the distributed implementation.

Fig. 1. Strong scaling of our Color-BiCC implementation from 1 to 64 ranks on AiMOS with (Color-BiCC-Dist) and without filtering (Color-BiCC-NoFilter) relative to the shared memory Slota-Madduri algorithm (Color-BiCC-SM) on 20 threads and the optimal serial Hopcroft-Tarjan algorithm (HT-Serial) on a single thread.



For Color-BiCC with edge filtering, we measure an average speedup of $15\times$ from 1 to 64 ranks. Relative to the serial and shared memory algorithms, we measure average speedups of $11\times$ and $7.3\times$, respectively. For all inputs except for web-Google, we note that the sum time for Color-BiCC and

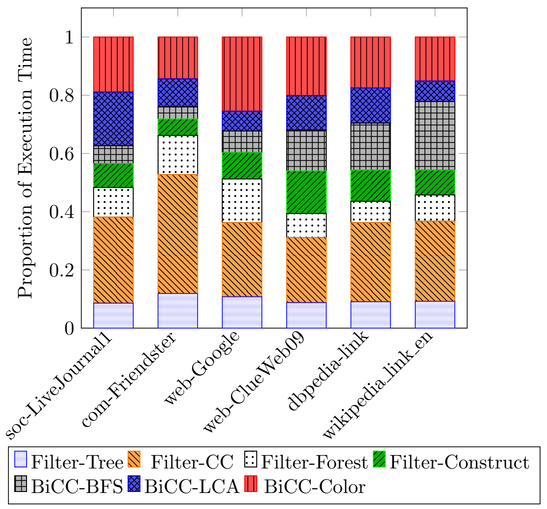
TABLE II

Comparison of Hopcroft-Tarjan (HT) on a single thread, the Slota-Madduri code (SM) on 20 threads, our distributed implementation of Color-BiCC without filtering (CBNF) on 64 ranks, our distributed implementation of Color-BiCC with filtering (CBD) on 64 ranks, and the speedup of Color-BiCC with filtering from 1 to 64 ranks.

Graph	HT	SM	CBNF	CBD	Speedup
soc-LiveJournal1	2.2	0.80	0.36	**0.23**	10×
com-Friendster	61	33	5.6	**2.2**	30×
web-Google	0.21	0.098	**0.047**	0.060	3.7×
web-ClueWeb09	30	38	7.3	**4.9**	8.9×
dbpedia-link	6.5	6.6	0.97	**0.72**	22×
wikipedia_link_en	9.3	6.6	1.5	**1.0**	13×

filtering is lesser than the time for Color-BiCC to process the unfiltered input. On a single rank and 20 threads, we note that our distributed Color-BiCC implementation is 1.9× slower on average than the Slota and Madduri code. However, we believe that this difference is quite reasonable in terms of overhead, considering that the communication routines and queue structures are still being utilized on single rank runs. For the largest inputs of com-Friendster and web-ClueWeb09, we observe scaling past the shared memory performance in as little as 2 or 4 ranks. For a distributed implementation operating on irregular graph inputs, we believe this performance to be in-line with the state-of-the-art. Table II summarizes these performance results. We note in bold the fastest execution time, which is CBD for all inputs except for web-Google, and we also give the speedup of CBD relative to the serial algorithm.

Fig. 2. Proportion of execution total time of BCC-Color-Dist for each of the seven primary stages: Spanning tree, connected components, spanning forest, and graph construction for the filtering algorithm; as well as the BFS, LCA, and coloring stages of the biconnected components algorithm.



For our final analysis of results, we consider the proportional execution time breakdown of Color-BiCC with filtering on 64 ranks of AiMOS. This is given in Figure 2. We note that the connected components routine of the filtering algorithm is on average the largest proportion of execution time, followed by the color propagation phase of Color-BiCC. These results are relatively unsurprising, given that the filtering algorithm is running on the entire unfiltered graph, while the color propagation routine usually requires the greatest number of supersteps out of all of our implemented procedures. We finally note that on 64 ranks, the ratio of communication to computation time is approximately balanced. This is surprisingly consistent for all of our larger inputs (web-Google has a higher relative communication), likely due to our relatively naïve hash-based 1D distribution.

A. Other Comparisons

Here, we compare our performance to the fastest distributed biconnectivity algorithms we have identified in the literature, that of Feng et al. [9]. As their code is not publicly available, a direct comparison is not possible. They run on the 2010 Twitter crawl from the Laboratory for Web Algorithmics[7] and the California Road Network from SNAP[8], using 8 and 60 Pregel workers (threads) spread across 2 and 15 “Amazon EC2 r3.2xlarge instances with enhanced networking”, respectively. For their fastest implementation (GD-BCC), they report times of about 12 seconds for the road network and 100 seconds for the Twitter crawl; their Tarjan-Vishkin code was quite slower. Comparatively, we observe serial Hopcroft-Tarjan times on AiMOS of 0.8 seconds for the road network and 92 seconds for Twitter. Mirroring their distributed experimental setup (4 threads per node, on 2 and 15 nodes) for our Color-BiCC implementation, we observe times of about 0.3 seconds for the road network and 12 seconds for Twitter.

We note the obvious, in that these times cannot be directly compared across different hardware setups. In addition, the overheads of Pregel can be considerable relative to a lightweight HPC-focused framework such as HPCGraph. However, we still point out that these differences by themselves likely do not explain the entire observed differences in execution times.

VIII. Conclusion

We have presented our distributed implementation and port of the shared memory Color-BiCC algorithm for graph biconnectivity computation. In addition, we present the first distributed implementation for Cheriyan-Thurimella edge filtering. We note that our relatively simple and theoretically suboptimal approach scales well past the optimal serial algorithm and appears to be considerably faster than other existing state-of-the-art biconnectivity codes. A straightforward extension of this work would be further optimization of the repeated constituent subroutines in our two implementations, such as spanning tree construction. In addition, the ideas we discussed in this paper might be also applicable towards the design of efficient distributed triconnectivity or general k-connectivity algorithms.

[7]https://law.di.unimi.it/webdata/twitter-2010/

[8]https://snap.stanford.edu/data/roadNet-CA.html

IX. Acknowledgments

We thank the Center for Computational Innovations at RPI for supplying and maintaining computational resources used in this work, including the AiMOS supercomputer supported by the National Science Foundation under Grant No. 1828083.

References

[1] I. Bogle, K. Devine, M. Perego, S. Rajamanickam, and G. M. Slota, "A parallel graph algorithm for detecting mesh singularities in distributed memory ice sheet simulations," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[2] R. E. Moraes and C. C. Ribeiro, "Power optimization in ad hoc wireless network topology control with biconnectivity requirements," *Computers & Operations Research*, vol. 40, no. 12, pp. 3188–3196, 2013.

[3] C. A. R. Pinheiro, *Social network analysis in telecommunications*. John Wiley & Sons, 2011, vol. 37.

[4] D. Day, M. Bhardwaj, G. Reese, and J. Peery, "Mechanism free domain decomposition," *Computer methods in applied mechanics and engineering*, vol. 192, no. 7-8, pp. 763–776, 2003.

[5] A. Frank, "Connectivity and network flows," *Handbook of combinatorics*, vol. 1, pp. 111–177, 1995.

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

[7] G. Cong and D. A. Bader, "An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps)," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 9–pp.

[8] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1821–1832, 2014.

[9] X. Feng, L. Chang, X. Lin, L. Qin, W. Zhang, and L. Yuan, "Distributed computing connected components with linear communication cost," *Distributed and Parallel Databases*, vol. 36, no. 3, pp. 555–592, 2018.

[10] G. M. Slota and K. Madduri, "Simple parallel biconnectivity algorithms for multicore platforms," in *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, 2014, pp. 1–10.

[11] 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*, 1991, pp. 391–401.

[12] J. Hopcroft and R. Tarjan, "Algorithm 447: efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, 1973.

[13] A. Buluç, S. Beamer, K. Madduri, K. Asanovic, and D. Patterson, "Distributed-memory breadth-first search on massive graphs," *arXiv preprint arXiv:1705.04590*, 2017.

[14] S. Stergiou, D. Rughwani, and K. Tsioutsiouliklis, "Shortcutting label propagation for distributed connected components," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, 2018, pp. 540–546.

[15] M. Chaitanya and K. Kothapalli, "Efficient multicore algorithms for identifying biconnected components," *International Journal of Networking and Computing*, vol. 6, no. 1, pp. 87–106, 2016.

[16] M. Ahmadi and P. Stone, "A distributed biconnectivity check," in *Distributed Autonomous Robotic Systems 7*. Springer, 2006, pp. 1–10.

[17] A. Kazmierczak and S. Radhakrishnan, "An optimal distributed ear decomposition algorithm with applications to biconnectivity and outerplanarity testing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 2, pp. 110–118, 2000.

[18] I. Bogle and G. M. Slota, "Distributed algorithms for the graph biconnectivity and least common ancestor problems," in *The 6th IEEE Workshop on Parallel and Distributed Processing for Computational Social Systems*, 2022.

[19] G. M. Slota, S. Rajamanickam, and K. Madduri, "High-performance graph analytics on manycore processors," in *International Parallel & Distributed Processing Symposium (IPDPS)*, 2015.

[20] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[21] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[22] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[23] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd international conference on world wide web*, 2013, pp. 1343–1350.

[24] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.