Distributed Algorithms for the Graph Biconnectivity and Least Common Ancestor Problems

Ian Bogle & George M. Slota
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY
boglei@rpi.edu, slotag@rpi.edu

Graph connectivity analysis is one of the primary ways to analyze the topological structure of social networks. Graph biconnectivity decompositions are of particular interest due to how they identify cut vertices and cut edges in a network. We present the first, to our knowledge, implementation of a distributed-memory parallel biconnectivity algorithm. As part of our algorithm, we also require the computation of least common ancestors (LCAs) of non-tree edge endpoints in a BFS tree. As such, we also propose a novel distributed algorithm for the LCA problem. Using our implementations, we observe up to a 14.8× speedup from 1 to 128 MPI ranks for computing a biconnectivity decomposition.

I. Introduction

The general graph biconnectivity decomposition problem seeks to identify all maximal biconnected subgraphs (subgraphs that can't be disconnected with the removal of a single vertex) as well as all cut vertices (or *articulation points*) and cut edges (or *bridges*) within some graph.

Biconnectivity is a well studied problem, with a work optimal depth-first search (DFS) algorithm originally proposed by Hopcroft and Tarjan [1]. As DFS lacks good parallelisms, Tarjan and Vishkin [2] later presented a parallel algorithm for finding biconnected components in a concurrent-read, concurrent-write PRAM model. This parallel algorithm reduces the problem of biconnectivity to the problem of connectivity in an auxiliary graph. The auxiliary graph can be constructed without using DFS and its connectivity can then be found efficiently in shared memory.

More recently, Slota and Madduri presented shared-memory parallel algorithms for biconnectivity by utilizing simple graph subroutines such as breadth-first search (BFS) and color propagation [3]. Most recently, LCA-BiCC was proposed by Chaitanya and Kothapalli [4]. The use a BFS spanning tree along with lowest common ancestor (LCA) vertices of all non-tree edges to identify all bridges and then perform a recursive algorithm on the components resulting from the removal of these bridges. The *lowest common ancestor* w of two vertices u, v in a rooted tree is the *lowest* (i.e., farthest from the root) vertex that has both u and v as descendants.

Our Contribution: In this work, we present the first parallel biconnectivity algorithm for a distributed graph structure. As with the algorithm of Chaitanya and Kothapalli [4], we require the computation of LCA vertices. We therefore also propose the first fully distributed algorithm for computing LCAs. We observe strong scaling with our algorithms up to 128 MPI ranks with speedups up to 14.8× across a selection of large test instances.

II. Distributed Biconnectivity

Our algorithm builds upon the *degenerate mesh features* algorithm of Bogle et al. [5]. That algorithm specifically considered ice sheet meshes and degenerate features (e.g., *hinges* of ice connected by a single mesh vertex to an ice sheet) that prevented climate simulation convergence. The Bogle et al. algorithm used the notion of *two paths* – any non-degenerate mesh vertex will have at least two paths to the known stable part of the mesh. We generalize this basic approach to the biconnectivity via Whitney's 1932 Theorem, which shows that for all u, v vertices in a 2-connected subgraph there exists two edge-disjoint u, v-paths.

Algorithm 1 Generalized BiCC Algorithm

```
1: procedure BCC-Overview(G = (V, E))
2:     parents, levels ← Dist-BFS(G)
3:     potential_artpts ← LCA-Heuristic(G,parents, levels)
4:     BCC-LR(G, levels, potential_artpts)
```

Our overall approach is given in Algorithm 1. First, we do a distributed BFS using subroutines from the HPCGraph Framework [6] to get a rooted spanning tree of the global graph defined by *parents* and *levels*. Then, an LCA-based heuristic uses the BFS tree to identify a set of potential articulation points (*potential_artpts*). This set is guaranteed to contain all actual articulation points [4]. Finally, our label propagation algorithm uses the set of potential articulation points and the BFS tree to uniquely label each biconnected component in the global graph.

A. LCA Heuristic

The LCA Heuristic algorithm starts *LCA traversals* for the endpoint pair of vertices of each non-tree edge of the BFS tree. Without explicit list construction, non-tree edges are easily identified using *parent* information. An LCA traversal replaces the lower-level (farthest from root) vertex with its parent, until both vertices in the traversal reach the same ancestor. In a

distributed-memory context, we need to communicate traversal progress across processor boundaries. Additionally, we keep track of any edge that we have traversed, as the endpoints of unvisited edges are within our potential articulation point set.

In order to facilitate the distributed memory LCA traversals, we use three entries for each vertex in each queue *package* to track each LCA traversal: the global ID of the vertices, the global ID of the parents of the vertices, and the processes that owns the vertices. If the lower level vertex is owned by a remote process, all six values are sent to it in order for it to process the package. We process a single step of each traversal before communicating our queues in an all-to-all fashion. Note that after we flag a vertex as a potential articulation point, we also check to see if the vertex is owned on another process, ensuring that all flags are consistent across processes. After this procedure, the endpoints of any unvisited edge plus all LCA vertices make up the set of potential articulation points.

While there is likely room for further optimization in our procedure, this stage is a small portion of the total execution time. In addition, we claim that this is the first distributed LCA algorithm for labeling all non-tree vertex pairs. Distributed algorithms for LCA vertex-pair queries exist, but these (e.g., [7]) generally require extensive pre-processing in shared memory to construct some labeling that enables distributed LCA queries for some u, v vertex pair.

B. Label Propagation and Reduction

After we have the set of potential articulation points and the BFS tree, we can proceed with our label propagation algorithm. Our distributed memory approach is given in Algorithm 2. Our label propagation procedure uses two types of label, denoted as *LCA* and *Low*. LCA labels contain only global IDs of LCA vertices in the global graph. Low labels hold the lowest-level vertex that has a particular LCA label. As both of these labels can propagate across processor boundaries, they require special considerations in our communication function.

Initially, every vertex in the local graph starts with no LCA vertex, and its own ID as a Low label. Each local potential articulation point propagates its own ID as an LCA label to its lower-level neighbors. Propagation of the LCA vertices ensures that they only move down the tree, and when a vertex gets two or more LCA labels we *reduce* the labels. Label reduction involves applying a variation of LCA traversal to the multiple LCA labels. Algorithm 3 shows this procedure. We take the lowest-level LCA label from the set of multiple labels, and attempt to replace it, if possible. We abort this traversal if the lowest-level label has an *irreducible label*. A label set is irreducible for the current propagation sweep if it contains multiple labels and at least one is currently *irreducible*, meaning that the label traversal eventually leads to a remote LCA for which this process has not received any LCA label information.

Algorithm 3 shows the procedure for reducing labels. Note that LCA labels are stored in a set, so once traversals converge on a common LCA label, the size of the LCA label set for that vertex will decrease.

The procedure that propagates Low labels is given in Algorithm 4. We pass Low labels between vertices that have the same LCA label, such that each vertex retains the Low label representing the vertex at the lowest level in the BFS

Algorithm 2 BiCC Label Propagation

```
1:  procedure BCC-LR(G, levels, potential_artpts)
2:      for all v ∈ V do
3:          LCA_labels[v] ← null
4:          low_labels[v] ← v
5:      queue ← {all IDs of local potential_artpts}
6:      verts_to_send ← ∅
7:      labels_to_send ← ∅
8:      LCA_procs_to_send ← ∅
9:      while queue is not empty on some process do
10:         irreducible_queue ← ∅
11:         while queue is not empty on this process do
12:             curr_vtx ← queue.pop()
13:             if LCA_labels[curr_vtx].size() > 1 then
14:                 Reduce-Labels(...)
15:                 if LCA_labels[curr_vtx].size() = 1 then
16:                     for all neighbor n of curr_vtx do
17:                         Push-Low-Labels(...)
18:                 if LCA_labels[curr_vtx] or
19:                         low_labels[curr_vtx] changed then
20:                     verts_to_send.push(curr_vtx)
21:                     labels_to_send.push(LCA_labels[curr_vtx])
22:                     labels_to_send.push(low_labels[curr_vtx])
23:             for all neighbor n of curr_vtx do
24:                 Pass-Labels(curr_vtx, n, ...)
25:         Label-Comm(...)
26:         queue ← irreducible_queue
27:         irreducible_queue ← ∅
```

Algorithm 3 Label Reduction Procedure

```
1:  procedure REDUCE-LABELS(
        G, curr_vtx, levels, LCA_labels, queue, irreducible_queue)
2:      if irreducible_queue contains curr then
3:          return
4:      while LCA_labels[curr_vtx].size() ≠ 1 do
5:          low = lowest level ID in LCA_labels[curr_vtx]
6:          if LCA_labels[low].size() = 1 then
7:              LCA_labels[curr_vtx].erase(low)
8:              LCA_labels[curr_vtx].insert(LCA_labels[low][0])
9:          else
10:             if irreducible_queue contains low then
11:                 irreducible_queue.push(curr_vtx)
12:             else if LCA_labels[low].size() = 0 then
13:                 irreducible_queue.push(curr_vtx)
14:             else
15:                 queue.push(curr_vtx)
```

tree with the lowest vertex ID. We also prevent Low labels from propagating up through potential articulation vertices.

Algorithm 4 Low Label Propagation

```
1: procedure PUSH-LOW-LABELS(
   curr_vtx, n, potential_artpts, levels, LCA_labels, Low)
2:     if potential_artpts[curr_vtx] = true then
3:         if LCA_labels[curr_vtx] = LCA_labels[n] and
4:                                 (levels[n] ≤ levels[curr_vtx] or
5:                             LCA_labels[n][0] ≠ curr_vtx) then
6:             curr_low_lvl ← level[Low[curr_vtx]]
7:             n_low_lvl ← level[Low[n]]
8:             if (curr_low_lvl > n_low_lvl) or
9:                                 (curr_low_lvl = n_low_lvl and
10:                                Low[curr_vtx] > Low[n]) then
11:                Low[n] = Low[curr_vtx]
12:    else
13:        if LCA_labels[curr_vtx] = LCA_labels[n] then
14:            curr_low_lvl ← level[Low[curr_vtx]]
15:            n_low_lvl ← level[Low[n]]
16:            if curr_low_lvl > n_low_lvl or
17:                                (curr_low_lvl = n_low_lvl and
18:                                Low[curr_vtx] > Low[n]) then
19:                Low[n] = Low[curr_vtx]
```

Once the label propagation and reduction completes successfully, the sole LCA label for a vertex will point to an articulation vertex. If we perform reductions only when a vertex has two different labels, it has two vertex-disjoint paths to the final vertex represented by the LCA label. However, due to the parallel and distributed nature of our implementation, we need to restrict this baseline label propagation in order to prevent LCA labelings from forming cycles. These propagation rules are described in Algorithm 5.

Importantly, both potential articulation points and regular vertices propagate a given LCA label only to vertices lower in the tree than the label. This restriction prevents cyclical behavior in propagation of LCA labels. Additionally, we also start collecting information needed to send to remote processes during the propagation. A standard communication pattern of communicating owned vertex values to ghosted copies is not sufficient in this algorithm. We also need to send the set of LCA labels that the owned vertex has, so that remote processes do not need to wait to be able to compute local label reductions.

III. Results

A. Experimental Setup

Experiments for our distributed biconnectivity algorithm were performed on Rensselaer Polytechnic Institute's DRP machine. The DRP cluster has 64 nodes, each with two eight-core 2.6 GHz Intel Xeon E5-2650 processors and 256GB of system memory, connected via 56 Gb FDR Infiniband. Our experiments allocate a maximum of 16 ranks per node, with our largest experiments using 128 MPI ranks. We use edge-block

Algorithm 5 Label Propagation Rules

```
1: procedure PASS-LABELS(G, curr_vtx, n, LCA_labels,
   low, levels, potential_artpts, queue, verts_to_send, la-
   bels_to_send, LCA_procs_to_send)
2:     if potential_artpts[curr_vtx] = true then
3:         if (levels[n] ≤ levels[curr_vtx] and
4:                 curr_vtx was recently reduced) or
5:                 (levels[n] = levels[curr_vtx] and
6:                         curr has a single label) then
7:             pass LCA labels that curr_vtx has to n
8:         else if levels[n] > levels[curr_vtx]
9:             pass ID of curr_vtx to n as an LCA label
10:    else
11:        if level[n] > level[curr_vtx] and
12:                curr_vtx has a reduced label then
13:            pass LCA labels that curr_vtx has to n
14:    Push-Low-Labels(curr_vtx, n, ...)
15:    if n's labels changed and n is ghosted then
16:        verts_to_send ← n
17:        labels_to_send ← LCA_labels[n], low[n]
18:        LCA_procs_to_send ← ranks with copy of n
19:        queue ← n
```

partitioning, where we assign to each rank approximately $\frac{|E|}{R}$ edges, where R is the number of ranks.

Graph	Type	# Verts	# Edges	d_{max}	BiCCs
com-LiveJournal	Social	3.9M	69.3M	14.8K	594k
wiki-Talk	Social	2.3M	5M	100K	34k
roadNet-CA	Road	1.9M	5.5M	12	327k
roadNet-PA	Road	1.0M	3M	9	194k
web-Google	Web	0.9M	5.1M	6K	60k

TABLE I
GRAPHS USED IN OUR EXPERIMENTS FOR BICONNECTIVITY.

Table I shows the details for the graphs we used to test our distributed biconnectivity algorithm. We selected social, web, and road networks from the SNAP database[1], as these are representative of the social networks and other graphs generally considering in connectivity analysis applications.

As it is typical for evaluation of k-connectivity algorithms [3], [4], [8] to run on the largest $(k-1)$-connected component of an input, we run our algorithm on the largest connected component of each graph to allow for relative comparison (trivially, any k-connectivity algorithm could be used on graphs with multiple $(k-1)$-connected components by first calling a $(k-1)$-connectivity algorithm and running on the extracted components in serial or parallel). We attempted to implement the Tarjan-Vishkin algorithm [2] in distributed memory for comparison; however, the necessary tree preordering and auxiliary graph construction phases of algorithm resulted in execution times slower than our code by orders-of-magnitude.

[1] http://snap.stanford.edu/snap/

B. Biconnectivity Algorithm Performance

Our algorithm has three main parts: the BFS, the LCA heuristic, and the label propagation algorithm. Figure 1 shows the strong scaling of each part of our distributed biconnectivity algorithm on each of our test graphs. Due to the irregularity of the degree distribution of the wiki-Talk graph, 128 ranks railed due to memory errors. Explicit partitioning to balance vertices and edges would address this issue, but optimizing partitioning approaches is beyond the scope of this work. The overall approach is able to scale on the other graphs up to 128 ranks, and scales on wiki-Talk upto 64 ranks. In terms of overall speedup, we see the highest speedups of 12.8x and 14.8x on roadNet-CA and roadNet-PA, respectively. We see a 3.6x speedup on wiki-Talk, 3.1x speedup on web-Google, and a 2.8x speedup on com-LiveJournal.

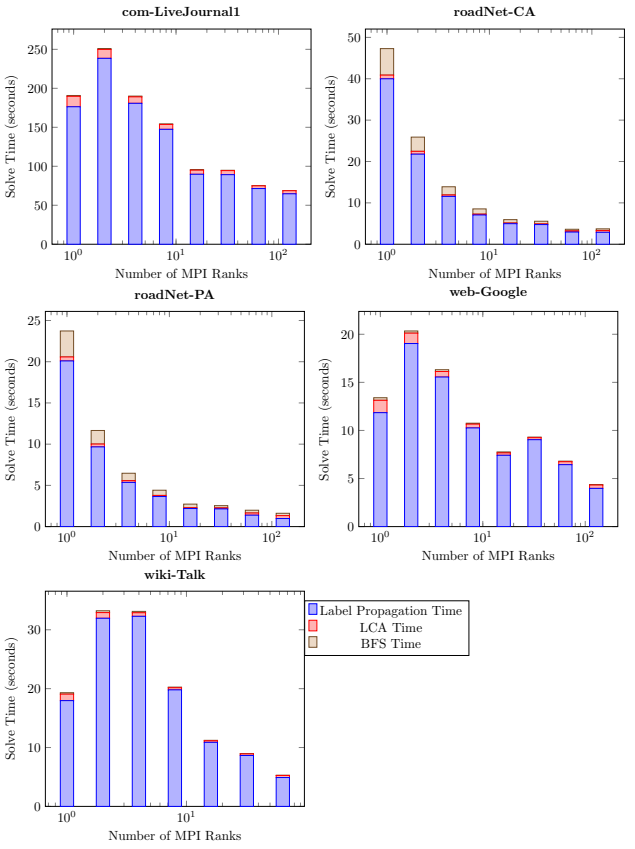

Fig. 1. Strong scaling of our BiCC algorithm from 1 to 128 ranks.

From figure 1, we see that label propagation is currently the most time-consuming part of this approach. Though it takes the most time, the label propagation scaling does not tend to taper off in most of the graphs that we consider. Label propagation spends a majority of the time doing computation, with cumulative communication times never exceeding half a second. Our communication pattern is aimed to reduce both the amount of communication and maximize the number of label reductions which can make progress. However, the traversals that determine what to send can be expensive during initial processing of the algorithm. Additionally, at higher numbers of ranks, a greater number of reductions will often get serialized via our propagation rules, limiting scalability to a degree.

The label reductions used in the label propagation algorithm are nearly identical to the traversals done in the LCA algorithm. An optimization we will pursue in future work is converting the current communication pattern to one that is more like the LCA Heuristic. This should greatly accelerate label propagation, by essentially trading a small increase in communication overhead with a large improvement in load balance.

IV. Conclusions and Future Work

We have proposed novel algorithms for performing a biconnectivity decomposition of a distributed graph and for computing LCA vertices in a distributed tree. Our ongoing work is further optimizing distributed-memory algorithms for these problems to enable further scalability to massive graph datasets. In addition, we have considered using a similar approach to identify triconnected components in graphs. This extension might be quite promising, as efficient distributed algorithms for triconnectivity are not yet present in the literature.

Acknowledgments

We thank the Center for Computational Innovations at RPI for supplying and maintaining computational resources used in this work. This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute under Contract No. DE-AC02-05CH11231 at Rensselaer Polytechnic Institute and Sandia National Laboratories and through the SciDAC ProSPect project at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973.

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

[3] G. Slota and K. Madduri, "Simple parallel biconnectivity algorithms for multicore platforms," in *Int. Conf. High Performance Computing (HiPC)*, 2014.

[4] M. Chaitanya and K. Kothapalli, "Efficient multicore algorithms for identifying biconnected components," *Int. J. Networking & Computing*, vol. 6, pp. 87–106, 2016.

[5] 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 *International Conference on Parallel Processing (ICPP)*, 2019.

[6] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: Implementation and optimization," in *International Parallel & Distributed Processing Symposium (IPDPS)*, 2016.

[7] I. D. Scherson and C.-K. Chien, "Least common ancestor networks," *VLSI Design*, vol. 2, no. 4, pp. 353–364, 1995.

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