# Efficient Parallel Ear Decomposition of Graphs with Application to Betweenness-Centrality

Charudatt Pachorkar, Meher Chaitanya, Kishore Kothapalli
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India 500 032
Email: {charudatt.p@research.,meher.c@research.,kkishore@}iiit.ac.in
Debajyoti Bera
Indraprastha Institute of Information Technology, Delhi
Okhla Industrial Estate, Phase III, New Delhi, India 110 020
Email: dbera@iiitd.ac.in

***Abstract*—Parallel graph algorithms continue to attract a lot of research attention given their applications to several fields of sciences and engineering. Efficient design and implementation of graph algorithms on modern manycore accelerators has to however contend with a host of challenges including not being able to reach full memory system throughput and irregularity. Of late, focusing on real-world graphs, researchers are addressing these challenges by using decomposition and preprocessing techniques guided by the structural properties of such graphs.**

**In this direction, we present a new GPU algorithm for obtaining an ear decomposition of a graph. Our implementation of the proposed algorithm on an NVidia Tesla K40c improves the state-of-the-art by a factor of 2.3x on average on a collection of real-world and synthetic graphs. The improved performance of our algorithm is due to our proposed characterization that identifies edges of the graph as *redundant* for the purposes of an ear decomposition.**

**We then study an application of the ear decomposition of a graph in computing the betweenness-centrality values of nodes in the graph. We use an ear decomposition of the input graph to systematically remove nodes of degree two. The actual computation of betweenness-centrality is done on the remaining nodes and the results are extended to nodes removed in the previous step. We show that this approach improves the state-of-the-art for computing betweenness-centrality on an NVidia K40c GPU by a factor of 1.9x on average over a collection of real-world graphs.**

## I. Introduction

Graph algorithms on modern many- and multi-core architectures are fundamental to computer science with applications to several domains. However, architectural peculiarities in modern many- and multi-core architectures impose significant constraints on the practical performance of parallel graph algorithms. For instance, the irregular memory accesses that are induced by parallel graph algorithms on modern computer architectures imposes a huge performance penalty on their performance. To alleviate these performance bottlenecks, researchers relied on optimizations with respect to data structures [8], memory layouts, and SIMD processing [27].

A recent trend to improve the practical efficiency of parallel graph algorithms is to, in addition, focus on real-world graphs and look for structural properties of such graphs and their impact on the algorithm design and implementation process. Examples include algorithmic enhancements tailored to real-world graphs for biconnectivity [7], strong-connectivity of directed graphs [13], preprocessing to reduce the size of the graph [2], decomposition into subgraphs based on biconnected components for shortest paths and betweenness-centrality [4], [28], decomposition based on Metis and ParMetis for shortest paths [12], and the like.

One such property that we focus in this paper is that real-world graphs, being sparse in nature, tend to have a reasonably good number of nodes of degree at most two. Removing such nodes can be helpful in problems such as computing the betweenness-centrality of a given graph. However, care is required to ensure that the centrality values of nodes removed from the graph can be computed efficiently using the corresponding values at nodes of degree greater than two. To this end, we use the ear decomposition of a graph that helps us to systematically identify and bookkeep details of nodes of degree at most two. An ear decomposition of a graph is a partitioning of a graph into simple paths that overlap only at end points. See Figure 1 for an example.

This paper hence studies the ear decomposition of a graph along with an application to compute the betweenness-centrality of the nodes in a graph. Formally, an ear decomposition of a graph is an ordered partition of the edges of the graph into simple paths $P_0, P_1, \cdots$, such that $P_0$ is an edge, $P_0 \cup P_1$ is a cycle, and for $i \geq 1$, $P_i$ has no common nodes with other lower indexed paths except the end points. An ear decomposition of a graph finds applications to other graph problems such as triconnectivity and planarity testing (cf. [26]).

In this paper, we show that the existing parallel algorithm for obtaining an ear decomposition as presented by Ramachandran [26] offers scope for improvement. To this end, we identify certain edges to be redundant for obtaining an ear decomposition and therefore *remove* these edges from consideration. (See Lemma 2.1). As a result of this characterization, the algorithm of Ramachandran et al. [26] needs to be applied only on a sparse subgraph of the original graph. Given the sparse nature of the input graph, we incorporate suitable changes in the computationally heavy steps of the algorithm of [26]. With these techniques, our implementation outperforms

existing approaches by a factor of 2.3x on real-world instances from the UFL dataset [1].

As an application of ear decomposition we show how to compute the betweenness centrality of nodes in a graph $G$. This measure indicates the relative importance of each node $v$ of $G$ by considering number of shortest path passing through $v$. We start with biconnected graphs and show that one can use an ear decomposition of a biconnected graph to improve the practical efficiency of computing the betweenness centrality values of nodes in the graph. In particular, we use the ear decomposition of a graph to remove nodes of degree two, perform the computation of betweenness-centrality on the remaining graph, and use a post-processing step to compute the betweenness-centrality of nodes that were removed.

Having a post-processing step in our algorithm means that unlike existing approaches [20], [21], [2], we need to retain the results from the processing phase to the post-processing phase. Doing so would require a large amount of space that far exceeds the space available on current generation GPUs. To address this problem, we interleave execution of the processing and the post-processing steps along with identifying redundant information that need not be stored and an orchestration of nodes in the processing step. Using these techniques our implementation outperforms existing approaches by a factor of 1.51x speedup on a collection real-world graphs from the UFL dataset [1].

Finally, using ideas from Sariyuce et al. [28] and Wang et al. [31] we show how to extend our approach to graphs that are not biconnected. Our approach achieves a speedup of 1.9x on a collection of real world graphs from [1]. We note that any improvements to GPU-based algorithms for breadth first search (BFS) can improve our results too.

### A. Related Work

Decomposition of graphs into subgraphs is a technique in parallel graph algorithms that is gaining significant research attention in recent years. Metis [17] decompose a graph into a given number $k$ of subgraphs such that the number of edges that cross a partition is minimized. This decomposition is being used parallel graph algorithms lately as a subroutine [12]. However, for path-based problems such as shortest paths and betweenness-centrality, decomposition via Metis may not be ideal due to the presence of cycles that go across the partitions induced by a Metis decomposition. These cycles mean that computing shortest paths between nodes in two different partitions is non-trivial.

Parallel algorithms in the PRAM style for obtaining an ear decomposition are presented by Ramachandran [26] along with applications to problems such as planarity testing and tri-connectivity. Bader et al. [3] show the results of implementing algorithm [26] on NPACI Sun E10K machines.

Computing the betweenness-centrality values of nodes in a graph has seen lot of interest in recent years in parallel computing research. Most papers [2], [28], [20], [21] use the algorithmic approach of Brandes [6]. Sariyuce et al. [28] use a biconnected component decomposition of a graph, compute the betweenness-centrality of a node local to its biconnected component, followed by a post-processing step for computing the betweenness-centrality values with respect to the entire graph. In a sequential computing model, they show that such a technique results in a speedup of 3.8x compared to Brandes [6]. Wang et al. [31] provided optimizations similar to [28] and achieve considerable speedup over existing multicore implementations including that of the Ligra framework [19].

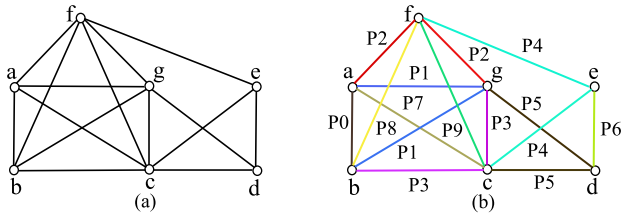


Fig. 1. An example of an ear decomposition. The labels on edges in part (b) of the figure indicate the ear number they belong to.

Optimizing BFS on a GPU with applications to betweenness-centrality in unweighted graphs is studied by Sariyuce et al. [2], and by Bader and Mc. Laughlin [20], [21], [32]. Bader and Mc. Laughlin [20] improved the BFS implementation of Jia et al.[16] and Shi and Zhang [29]. Bader and McLaughlin perform a BFS with each node of the graph as the source of the BFS. Each such BFS is run in parallel on a SMX (SM) of the GPU. The information obtained from the BFS from $v$ is used to compute the betweenness-centrality of $v$. Improvements to GPU BFS shown by Bader and Mc. Laughlin [21] lead to direct improvements computing betweenness-centrality values over their results from [20].

### B. Organization of the Paper

The rest of the paper is organized as follows. In Section II, we show our algorithm and its implementation for obtaining an ear decomposition of a graph. In Section III, we show how to use the ear decomposition of a graph to compute the betweenness-centrality values of nodes in a given graph. The paper ends with concluding remarks in Section IV.

## II. Ear Decomposition

An ear decomposition of a biconnected graph $G = (V, E)$ is an ordered partitioning of the edges of $G$ into simple paths (ears) $P_0, P_1, \cdots$ as follows (see also [26]).

- $P_0$ is an edge $uv$,
- $P_0 \cup P_1$ is a simple cycle, and
- The end points of path $P_i$, for $i \geq 2$, are on the paths $P_0, P_1, \cdots, P_{i-1}$, and path $P_i$ has no other nodes common with the nodes on the paths $\cup_{j=0}^{i-1} P_j$.

An example is shown in Figure 1. Notice that an ear decomposition is not unique. The popular parallel algorithm of Ramachandran [26] for obtaining an ear decomposition proceeds as shown in Algorithm 1. The preorder numbers in Line 2 of Algorithm 1 can be computed, for instance, as described in [15]. For an edge $e = uv$, let $\text{lca}(e)$ be the least common ancestor (LCA) of the nodes $u$ and $v$.

In a PRAM style of analysis [15], the algorithm has a runtime of $O(\log n)$ and uses $O(m+n)$ work. However, in a

**Algorithm 1** EarDecompose$(G)(from[26])$

1: $T$ = SPANNINGTREE$(G)$
2: Root $T$ at a node $r$, and label each node in $T$ with its preorder number
3: **for** each non-tree edge $e = uv$ in $G'$ in parallel **do**
4: Label the edge $e$ with $\text{lca}(e)$
5: **end for**
6: Sort the labels of non-tree edges in increasing order as $1, 2, \cdots$
7: **for** each tree edge $f = (\text{parent}(x), x)$ of $T'$ in parallel **do**
8: Label $f$ with the label of the nontree edge with the smallest label whose fundamental cycle contains $f$.
9: **end for**
10: Edges with label $i$ form the ear $P_i$ for $i \geq 1$.
11: Relabel the nontree edge with label 1 to have label 0.

practical setting, operations indicated in the algorithm usually suffer from drawbacks mentioned below.

- Computing the preorder numbers of the nodes according to $T$ requires one to use the Euler tour technique [15]. This computation on pointer-based data structures such as linked lists involves a lot of uncolasced memory accesses that result in poor performance on GPUs.
- To identify the labels of the non-tree edges, one needs to compute the LCA of the end points of every non-tree edge. To achieve an $O(\log n)$ parallel runtime, the algorithm suggests an $O(\log n)$ time and $O(n)$ work preprocessing based on range minima algorithms for LCA queries. When one considers sparse graphs where the number of LCA queries are small owing fewer non-tree edges, such algorithms can increase the overhead on the computation.
- The labeling of tree edges also has practical difficulties similar to those mentioned above.

### A. Our Approach for Ear Decomposition

Let $G = (V, E)$ be a biconnected graph. In our work, we start by identifying certain edges of $G$ as redundant for the purposes of obtaining an ear decomposition. These redundant edges are removed from $G$ to get a subgraph $G'$. The graph $G'$ will have $n$ nodes and at most $2n - 2$ edges, making $G'$ a sparse graph. We show that an ear decomposition of $G'$ can be easily extended to an ear decomposition of $G$. To obtain an ear decomposition of $G'$, we exploit its sparsity to improve on the practical performance of the algorithm of Ramachandran [26].

*1) Identifying Redundant Edges:* We consider an edge of $G$ as redundant for obtaining an ear decomposition if $e$ can be included as an ear containing just the edge $e$. We call such an ear as a *trivial* ear. (See also [26]). A characterization for identifying redundant edges with respect to biconnectivity of a graph is presented by Cong and Bader [9]. A necessary and sufficient condition for a graph to have an ear decomposition is that the graph should be biconnected. Intuitively, edges that are redundant in maintaining the biconnected nature of a graph can also be possibly redundant for obtaining an ear decomposition.

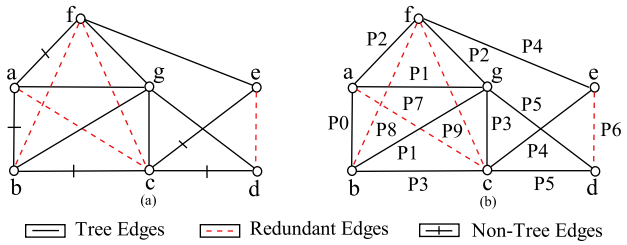


Fig. 2. An example of using Lemma 2.1. In the graph on the left, we note that edges shown in dashed lines and red color are redundant. An ear decomposition, as ears $P_0$ through $P_5$, of the graph with the rest of the edges is shown in the right part of the figure. As the lemma shows, ears $P_6$ through $P_9$ correspond to trivial ears of the redundant edges.

Indeed, as we show in the following lemma, biconnectivity and ear decomposition share the same notion of redundant edges.

*Lemma 2.1:* Let $T$ be a rooted BFS tree of a biconnected graph $G$ and $F$ be a spanning forest of the graph $G \setminus T$. Then, edges in $G \setminus (T \cup F)$ are redundant for the purposes of an ear decomposition.

*Proof:* Consider the graph $T \cup F$. According to the characterization of Cong and Bader [9], if the graph $G$ is biconnected, so is the graph $T \cup F$. Therefore, if $G$ is biconnected, then $T \cup F$ has an ear decomposition. Let $(P_0, P_1, \cdots, P_s)$ be an ear decomposition of the graph $T \cup F$.

We claim that, $(P_0, P_1, \cdots, P_s, Q_{s+1}, Q_{s+2}, \cdots, Q_{s+k})$ is an ear decomposition of $G$ where $Q_{s+i}$ is the edge $e_i$ in the graph $G \setminus (T \cup F)$ with $k = |E(G \setminus (T \cup F))|$.

To this end, notice that a single edge can also be an ear in itself, which we call as a trivial ear. Hence, each $Q_{s+i}$, for $1 \leq i \leq k$, is a valid ear. The endpoints of $Q_{s+i}$, for $1 \leq i \leq k$, belong to the nodes in $\cup_{j=1}^{s} P_j$. Finally, it can be noticed that $E(G) = (\cup_{i=0}^{s} P_i) \cup (\cup_{j=s+1}^{k} Q_j)$. Therefore, $(P_0, P_1, \cdots, P_s, Q_{s+1}, Q_{s+2}, \cdots, Q_{s+k})$ is an ear decomposition of $G$. Notice that the numbering of edges (ears) in $G \setminus (T \cup F)$ can be done in an arbitary manner. ■

An example is illustrated in Figure 2. Since $T$ contains $n - 1$ edges and $F$ has at most $n - 1$ edges, the above lemma indicates that on a graph $G$ of $n$ nodes and $m$ edges, the number of redundant edges is at least $m - 2n + 2$. The remaining graph has thus at most $2n - 2$ edges.

*2) Algorithm for Ear Decomposition:* The results of the earlier section indicate that obtaining an ear decomposition of a graph $G$ can be achieved by obtaining an ear decomposition of a sparse sub-graph $G'$. In this section, we make use of properties induced by the sparsity of $G'$ to arrive at an ear decomposition of $G'$ in an efficient manner. Our algorithm uses a preprocessing step followed by employing the algorithm of Ramachandran [26] with some changes, followed by a post-processing step. Our algorithm is presented as Algorithm 2 with a brief description in the subsequent paragraphs.

*a) Phase I:* In Phase I, the pruning step requires a BFS of $G$ and another spanning forest computation on the graph $G \setminus T$. For this, we use the optimized BFS implementation from [22]. As mentioned in Steps 2–4 of Algorithm 2, we first compute a BFS tree, $T$, of the graph $G$ and a spanning forest $F$ of the graph $G \setminus T$. As mentioned in Lemma 2.1, we remove all edges in $G \setminus (T \cup F)$ from further consideration.

**Algorithm 2** EarDecompose($G$)

1: **/* Phase I – Pruning */**
2: $T$ = BFS($G$)
3: $F$ = SPANNINGFOREST($G \setminus T$)
4: $G' = G \setminus (T \cup F)$
5: **/* Phase II – Ear Decomposition*/**
6: $T'$ = SPANNINGFOREST($G'$)
7: **for** each non-tree edge $e = uv$ in $G'$ in parallel **do**
8: Label the edge $e$ with $\langle \text{Level}(\text{lca}(e)), \#e \rangle$
9: **end for**
10: **for** each tree edge $f = (\text{parent}(x), x)$ of $T'$ in parallel **do**
11: LABELTREEEDGE($f$)
12: **end for**
13: **/* Phase III – Postprocessing */**
14: **for** each edge $e \in G \setminus (T \cup F)$ in parallel **do**
15: Include $e$ as an ear with just the edge $e$ alone.
16: **end for**

The remaining graph, $G'$, has $n$ nodes and at most $2n - 2$ edges.

*b) Phase II:* Since the graph is sparse and has at most $2n-2$ edges, the number nontree edges would be at most $n-1$. Thus, LCA has to be found for at most $n - 1$ pairs of nodes. Therefore, we note that a preprocessing for LCA queries may not be essential. Instead, the LCA of a pair of nodes $u, v$ can be obtained by using the simple technique of walking along the path from $u$ and $v$ to the root of the tree. This simple technique has the advantage that all the LCA queries can be done in parallel. The one disadvantage of the method is that the time spent for each LCA query will now depend on the distance of the LCA node. However, we show in a later section that most LCA queries traverse a distance that is indeed small. (See Tale I).

*Labeling of Tree Edges:* Notice that the graph for which we obtain an ear decomposition has at most $2n - 2$ edges of which $n - 1$ edges appear as tree edges. Hence, the routine LABELTREEEDGE in Line 11 of Algorithm 2 proceeds as follows. Therefore, $n - 1$ fundamental cycles contain $n - 1$ tree edges. We therefore expect that the number of cycles that pass through any given tree edge is small on average. This is verified also empirically as shown in Table I.

In light of the above observation, we expect that the total length of the fundamental cycles to be small. For this reason, to find the label for the tree edges, we list the edges of each fundamental cycle in an array $A$. Given that we can know the length of each fundamental cycle from Steps 7–9 of our approach, we can reserve space for each fundamental cycle in the array and also calculate the starting index in the array where the edges of each cycle have to be written. In this step, there would be no need for costly synchronization operations.

Each element of the array $A$ is of the form $\langle e, \ell, f \rangle$ where $e$ is the id of a tree edge, $\ell$ is the level number of the LCA of the end-points of the non-tree edge $f$ whose fundamental cycle passes through $e$. We now sort [15] the elements of $A$ according to the first elements, followed by the second element, and followed by the third elements of the tuples in $A$. According to such a grouping, all the tuples corresponding to each tree edge $e$ appear in a contiguous manner. Once such a grouping is achieved, we find the minimum in each group using the segmented prefix operation [15].

*c) Phase III – Postprocessing:* In this phase, edges that were pruned in Phase I will be included in the ear decomposition of $G$ as trivial ears.

### B. Experimental Results

We study the performance of our algorithm for ear decomposition in this section.

*1) Platform:* In this section, we introduce briefly our computing platform that consists of an NVidia Tesla K40c GPU attached to an Intel(R) Xeon(R) E5-2650. The Tesla K40c GPU has 2880 compute cores arranged as 192 cores each in 15 SMXs. It has 12 GB on board memory and 64 KB of on chip memory per each SMX. An L2 cache of 1.5 MB is shared among all SMXs. Each SMX has a hardware scheduler which schedules 32 threads at a time. This group is called a warp and a half-warp is a group of 16 threads that execute in a SIMD fashion. For more details of the Tesla K40c GPU, we refer the reader to [23]. For programming the K40c GPU, we use CUDA Version 7.5 as described in [24].

*2) Datasets:* We use the graphs listed in Table I for our experiments. The graphs include both real-world graphs from [1] and also Erdos-Reyni random graphs [5] generated using the RMAT generator [11]. Since we require the graph to be biconnected to have an ear decomposition, we made the graphs in Table I biconnected by adding additional edges as needed. We also remove self-loops and make all the graph undirected.

| **Graph name** | $\|V\|$ | $\|E\|$ | **Edges Pruned** | **Avg. Dist.** | **ACE** |
|---|---|---|---|---|---|
| roadNet-CA | 2.0 M | 2.7M | 15 K | 10.42 | 8.18 |
| roadNet-TX | 1.4 M | 1.9 M | 11 K | 10.44 | 7.77 |
| soc-Epinions1 | 76K | 508 K | 294 K | 2.17 | 1.89 |
| patents_main | 241K | 560 K | 185 K | 5.07 | 5.59 |
| coAuthorsDBLP | 299 K | 977 K | 447 K | 2.89 | 4.2 |
| soc-Slashdot0902 | 82K | 474 K | 371 K | 2.44 | 2.72 |
| caidaRouterLevel | 192 K | 609 K | 284 K | 3.4 | 4.3 |
| scircuit | 171 K | 479 K | 83 K | 3.12 | 4.74 |
| soc-sign-epinions | 131 K | 841 K | 527 K | 2.52 | 1.95 |
| p2p-Gnutella31 | 62 K | 147 K | 52 K | 4.16 | 4.17 |
| **Random Graphs $\mathcal{G}(n,p)$ [5]** | | | | | |
| $\mathcal{G}(1M, 10 \times 10^{-6})$ | 1 M | 10 M | 8 M | 4.86 | 9.31 |
| $\mathcal{G}(1M, 20 \times 10^{-6})$ | 1 M | 20 M | 18 M | 3.99 | 7.79 |
| $\mathcal{G}(1M, 40 \times 10^{-6})$ | 1 M | 40 M | 38 M | 3.68 | 6.86 |
| $\mathcal{G}(1M, 80 \times 10^{-6})$ | 1 M | 80 M | 77 M | 2.99 | 5.92 |
| $\mathcal{G}(2M, 10 \times 10^{-6})$ | 2 M | 10 M | 6 M | 5.15 | 7.6 |
| $\mathcal{G}(2M, 20 \times 10^{-6})$ | 2 M | 20 M | 16 M | 4.98 | 9.66 |
| $\mathcal{G}(2M, 40 \times 10^{-6})$ | 2 M | 40 M | 36 M | 4.13 | 8.03 |
| $\mathcal{G}(2M, 80 \times 10^{-6})$ | 2 M | 80 M | 76 M | 3.87 | 7.3 |

TABLE I
LIST OF SPARSE GRAPHS THAT WE USE IN OUR EXPERIMENTS. THE NUMBER OF NODES AND THE EDGES ARE ROUNDED TO THE NEAREST THOUSAND (K) OR THE NEAREST MILLION (M). THE NOTATION $\mathcal{G}(n,p)$ REFERS TO A RANDOM GRAPH ([5]) WITH $n$ NODES AND AN EDGE PROBABILITY OF $p$. THE NUMBER IN COLUMN LABELED "EDGES PRUNED" SHOWS THE NUMBER OF EDGES THAT ARE REDUNDANT ACCORDING TO LEMMA 2.1. THE COLUMN LABELED "AVG. DIST." SHOWS THE AVERAGE NUMBER OF TREE EDGES TRAVERSED TO FIND THE LCA OF THE END POINTS OF A NONTREE EDGE. THE NUMBER IN THE COLUMN LABELED "ACE" INDICATES THE AVERAGE NUMBER OF FUNDAMENTAL CYCLES ACCORDING TO A BFS TREE THAT PASS THROUGH A TREE EDGE.

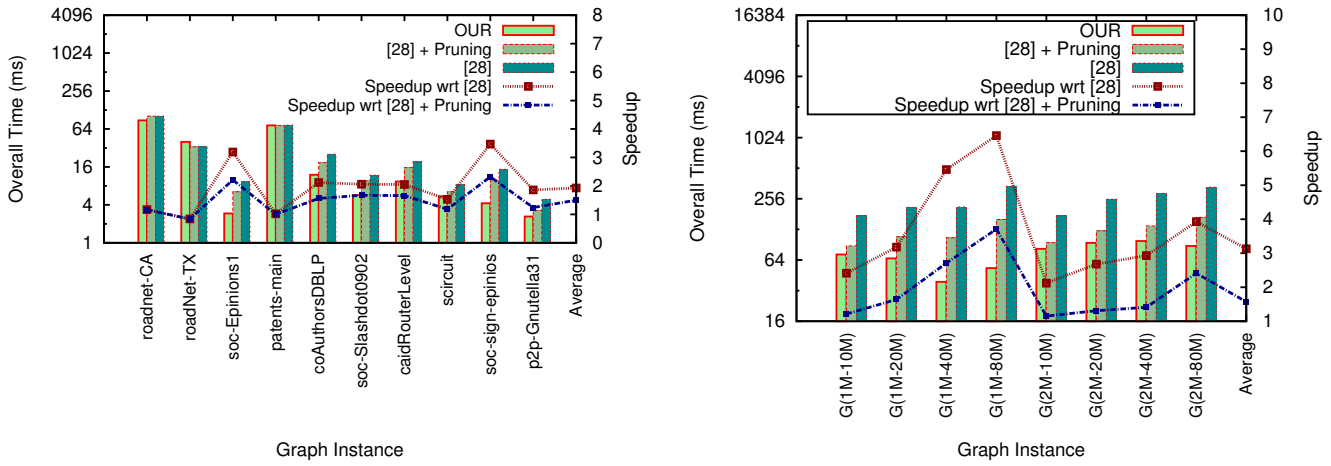


Fig. 3. Performance of our ear decomposition algorithm on real-world graph (a) and on random graphs (b). The last label "Average" indicates the average speedup on the dataset from Table I.

*3) Results:* We study the performance of our algorithm with respect to that of [26], labeled as "[26]" in Figure 3(a). For a better understanding of the performance of our algorithm labeled as $OUR$, we also add the pruning step from Algorithm 2 to the algorithm of Ramachandran [26]. This modification is labeled as "[26] + Pruning" in the plot in Figure 3(a). The "[26] + Pruning" approach uses the algorithm of [26] on the graph $G'$ as described in Algorithm 1. Since there are no implementations of the algorithm of [26] reported on GPUs, we implemented the algorithm of [26] on GPUs. In our implementations we have set number of threads per CUDA block to 512 as this number was observed to provide the best results for all implementations. Other GPU specific considerations such as memory coalescing were also employed to the extent possible in all implementations.

Figure 3(a) shows the absolute runtime as well as the speedup of our algorithm with respect to that of [26]. The speedup is shown on the secondary Y-axis. Note that the Y-axes are on a logarithmic scale. It can be noticed that our algorithm performs 2.3x better than the algorithm of [26]. If we add the pruning step to the algorithm of [26], our algorithm outperforms this variation by a factor of 1.54. This indicates that our performance gains are due to both the pruning step and other algorithmic enhancements to that of [26].

As the number of edges increase, Phase I of our algorithm removes a bigger number of edges thereby reducing the work in the latter phases resulting in a better speedup. This observation is supported by our experiments on random graphs in Figure 3(b) where we keep the number of nodes fixed at 1 M and 2 M nodes and increase the number of edges.

## III. Application to Betweenness-Centrality

An important computation on graphs is to find the betweenness-centrality value of the nodes in the graph. Betweenness centrality as a measure finds applications in several areas of graph analytics such as those in social networks [14] and biological graphs [18]. In a graph $G = (V, E)$ the betweenness centrality (BC) of a node $v \in V$ is a measure of the number of shortest paths that pass through $v$. Equation 1 (see also [6]) captures the above formally where $\sigma_{st}$ denotes number of shortest paths between $s$ and $t$, and $\sigma_{st}(v)$ denotes number of shortest paths between $s$ and $t$ that also pass through $v$.

$$bc(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (1)$$

Before we present our algorithmic approach for computing the betweenness-centrality values of nodes in a given graph, we briefly review the algorithm of Brandes [6] that has been the algorithm of choice [28], [20], [21], [2] for computing betweenness-centrality in parallel.

### A. Brandes Algorithm

Brandes introduced an algorithm to compute betweenness centrality that runs in $O(n+m)$ space and $O(nm)$ sequential time. The algorithm of Brandes works in two stages: a *forward propagation* stage and an *accumulation* stage. In the *forward propagation* stage, from each node $v \in V$ we first obtain the sequence $S_v$ in which nodes are visited according to the BFS algorithm with source node as $v$. During this step, we also store the number of shortest paths from $v$ to other nodes in $V$ as $\sigma_v$ and the parent of each node $u$ in the shortest path tree rooted at $v$, denoted as $P_v(u)$. This information is used in the *accumulation stage* to compute the partial betweenness centrality of nodes in $P_v$ by using the dependency relation $\delta(u) = \sum_{w:u \in P_v(w)} \frac{\sigma_{vu}}{\sigma_{vw}}(1 + \delta(w))$, where $u$ is the parent of $w$ in the shortest path tree rooted at $v$ and $\delta(w)$ denotes the partial betweenness centrality of a node $w$. The algorithm also uses an array $D_v()$ that contains the length of the shortest path from $v$ to all other nodes.

### B. The Approach of Bader et al. [20], [21]

The main idea of the works of Bader and Mc. Laughlin [20], [21] is to target GPU specific optimizations to perform multiple BFS operations, one from each node of the graph as a source node. Bader and Mc. Laughlin [20] use SMX level parallelism and batch the $n$ BFS operations on the SMXs. Other techniques introduced in [20] include memory

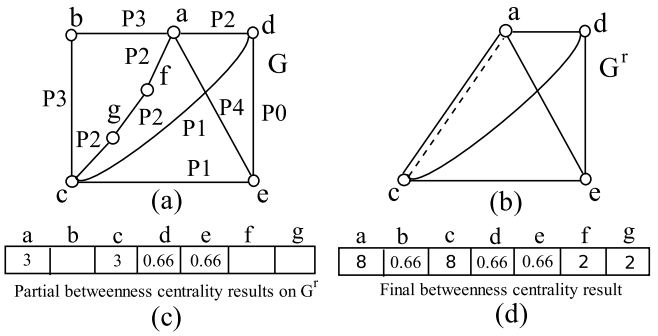


Fig. 4. Figure (a) shows the input graph $G$ with an ear decomposition where the number on the edges indicates the ear they belong to. Figure (b) shows the reduced graph $G^r$. A parallel edge in the reduced graph between nodes $a$ and $c$ is shown as a dotted line for illustration purposes. Figure (c) shows the *partial* betweenness-centrality values of nodes in $G^r$ computed in the processing phase of our algorithm. Figure (d) shows the final betweenness-centrality values for *all* nodes obtained at the end of the post-processing phase.

usage optimization, reduction in atomic operations, and load balancing based on the structure of the graph.

Bader and Mc. Laughlin [21] introduce further optimizations such as warp level parallelism and warp-level load balancing using dynamic scheduling. These optimizations result in an improved BFS performance and a direct improvement over [20] for computing the betweenness centrality values of nodes in unweighted graphs.

### C. Our Approach

Our algorithmic approach to compute betweenness-centrality of nodes in a sparse graph uses the following outline. We start by considering graphs that are biconnected. Such a graph will have an ear decomposition as shown by [26, Lemma 2.1]. In a preprocessing step, we obtain an ear decomposition of the graph. Using the ear decomposition to perform the necessary book-keeping, we remove nodes of degree two from the graph. Once the betweenness-centrality values of the remaining nodes are computed, in a post-processing step we compute the betweenness-centrality values for nodes removed during the preprocessing step. Finally, we show how to extend our approach to graphs that are not biconnected.

An illustration of our approach for biconnected graphs is shown in Figure 4. In the following, we present a pseudocode of our algorithm as Algorithm 3 and provide details of the steps in our algorithm in Sections III-C1–III-C3.

*1) Preprocessing:* Let $G = (V, E)$ be a biconnected graph. The REDUCE($G$) routine starts by obtaining an ear decomposition of $G$ using Algorithm 2. In such a decomposition, nodes of degree two, except possibly those on ear $P_0$, appear on exactly one ear. The resulting reduced graph $G^r = (V^r, E^r)$ is defined as follows. The nodes of $G^r$ are the nodes of $G$ that have a degree at least three. Two nodes $v$ and $w$ in $G^r$ are neighbors if and only if $v$ and $w$ belong to a common ear $P$ of $G$ and have no nodes of degree three or more between them on the ear $P$. Figure 4(a)–(d) shows an example. For a node $x$ of degree two on ear $P = (a_1 a_2 \cdots a_k)$ in $G$, we define functions *left()* and *right()* of $x$ in $G^r$, denoted $\text{left}(x)$ and $\text{right}(x)$, as the nodes of degree at least three on $P$ that are closest to $x$ towards $a_1$ and $a_k$ respectively. For instance, in the example in Figure 4(b), $\text{left}(f) = a$ and $\text{right}(f) = c$. (The terms left and right are only mnemonic in nature.)

**Algorithm 3** Algorithm BetweennessCentrality($G$)

```
1: /* Phase I : Preprocessing */
2:  G^r = REDUCE(G)
3: /* Phase II : Processing */
4: for each v in G^r do in parallel do
5:       (S_v, D_v, σ_v) = FWDSTAGE(v, G)
6:       ACCUMULATE_PARTIALBC(S_v, D_v, σ_v)
7: end for
8: /* Phase III : Post Processing */
9: for each v ∈ G \ G^r do in parallel do
10:      l_x ← left(v) , r_x ← right(v)
11:      (S_v, D_v, σ_v) = SIM_FWDSTAGE(v, l_x, r_x)
12:      SIM_ACCUMULATION(v, l_x, r_x, S_v, D_v, σ_v)
13:      Update the BC values to the nodes in G^r
14: end for
```

Notice that during the construction of the reduced graph, there could be multiple edges between nodes in the reduced graph. In this case, since we are interested in shortest paths, we retain the edge with the shortest length and discard the remaining edges. An example shown in Figure 4(b) for purposes of illustration.

*2) Processing:* In the processing phase, we compute betweenness-centrality values of nodes in $G^r$. We use the BFS routine from [21] in our processing step and perform a BFS in $G$ from each node in $G^r$ as the source node. Along with each BFS, we perform the forward propagation stage and the accumulation stage of the algorithm of Brandes. In the FWDSTAGE routine, for each source node $v$, arrays $S_v$, $D_v$ and $\sigma_v$ are recorded as the result of a BFS with $v$ as the source node as is done in the forward propagation phase. As $G$ is unweighted, $P_v$ is not computed or stored explicitly and is simulated using the $D_v$ array. In the ACCUMULATE_PARTIALBC routine, we use the arrays $S_v, D_v$, and $\sigma_v$ for each $v \in G^r$ computed in the forward stage to compute betweenness-centrality values of nodes in $G^r$. However, these computed betweenness-centrality values of nodes in $G^r$ can change in the post-processing step as the accumulation stage of nodes in $G \setminus G^r$ is performed. Therefore, we call these values as the *partial* betweenness-centrality values as shown in Figure 4(c).

*3) Post-processing:* In this phase, we compute betweenness centrality for nodes in $G \setminus G^r$ and also make updates to the partial betweenness-centrality values for nodes in $G^r$.

The routine SIM_FWDSTAGE works as follows. Let $x$ be a node in $G \setminus G^r$ with $\text{left}(x) = \ell_x$ and $\text{right}(x) = r_x$. We simulate the actions of executing the forward stage of Brandes algorithm [6] for node $x$ as follows. We need to obtain arrays $S_x, D_x$ and $\sigma_x$ as part of the forward stage. For obtaining the array $S_x$, we start by merging sequences $S_{\ell_x}$ and $S_{r_x}$ as follows. Let $v$ and $w$ denote the first node in $S_{\ell_x}$ and $S_{r_x}$ respectively. We now compare $D_{\ell_x}(x) + D_{\ell_x}(v)$ and $D_{r_x}(x) + D_{r_x}(w)$. If the former is smaller, then we add $v$ to $S_x$. Otherwise, we add $w$ to $S_x$. Nodes $v$ and $w$ are incremented to be the next node in $S_{\ell_x}$ and $S_{r_x}$ depending on which of $v$ or $w$ is added to $S_x$ in this step. (Alike the

procedure Merge in Merge Sort [10]). In a similar fashion, we can also obtain arrays $D_x$, and $\sigma_x$ from the respective arrays of nodes $\ell_x$ and $r_x$.

Once these arrays are obtained for node $x$, the accumulation stage (i.e SIM_ACCUMULATION routine) of Brandes algorithm can be simulated as described in Section III-A. During this stage, the partial betweenness-centrality values of nodes in $G^r$ will be updated as needed. At the end of the post-processing phase, we therefore have the final betweenness-centrality values of all nodes in $G$ as shown in Figure 4(d).

### D. Implementation Details

In this section, we mention some of the important implementation details of our algorithm. Since we use the implementation from [21] in Phase II of Algorithm 3, we stand to benefit from all the GPU specific optimimzations that are included in the implementation of [21].

The preprocessing step in our algorithm necessitates a post-processing step unlike other algorithms [20], [21], [32]. To run the post-processing step, as described in our algorithm, we need $O(n)$ Bytes of information per node of $G^r$ amounting to $O(n \cdot n^r)$ Bytes where $n^r = |V(G^r)|$. For even moderate value of $n$, this amount of space far exceeds the amount of space available on current generation GPUs.

To alleviate this problem, we run the processing and the post-processing steps in an interleaved manner. Doing so naively will not result in any improvement in the space utilization. However, we introduce two novel techniques in our implementation that help us in the following way. Firstly, in Section III-D1, we identify information computed in the processing phase that is not needed in the post-processing phase. Secondly, in Section III-D2, we orchestrate the nodes in $G^r$ as to when the processing step corresponding to a node is performed and how long the information thus generated has to be kept in the memory. This allows us to reuse the limited space effectively.

*1) Classifying Nodes in $G^r$:* We observe that some nodes in $G^r$ do not correspond to the $\mathrm{left}()$ and $\mathrm{right}()$ of any node in $G \setminus G^r$. Thus, nodes in $G^r$ can be partitioned into two subsets, $V^f$ and $V^a$. Nodes in $V^f$, which we call as *free nodes*, are such that their $S, D$, and $\sigma$ arrays are not required by any other node in $G \setminus G^r$ during post-processing. On the other hand, nodes $v \in V^a$, which we call as *active nodes*, are such that arrays $S_v, D_v$, and $\sigma_v$ are required during post-processing. Our storage requirement corresponds to storing the arrays for nodes in $V^a$. Information computed in the processing phase with respect to nodes in $V^f$ need not be retained for the post-processing phase.

*2) Orchestrating Nodes in the Processing Phase:* Our technique here involves ordering the nodes in the processing phase so that we can associate with every node $v \in V^a$ a lifetime during which the arrays $S_v, D_v$, and $\sigma_v$ are required in memory for post-processing. Once the lifetime of a node ends, the space used by its arrays can be reclaimed.

To this end, let $F$ denote the subgraph of $G^r$ induced by $V^a$. We now find the connected components of $F$ using standard parallel algorithms such as those presented in [30]. We also order the connected components of $F$ in some order, say $F_1, F_2, \cdots$,. Consider a connected component $H$ of $F$ and define $\mathrm{Dep}(H) := \{x | x \in G \setminus G^r, \mathrm{left}(x) \in V(H)$ or $\mathrm{right}(x) \in V(H)\}$. Once nodes in $\mathrm{Dep}(H)$ finish their post-processing, the information with respect to nodes in $H$ is no longer required to be in memory and the associated space can be reused.

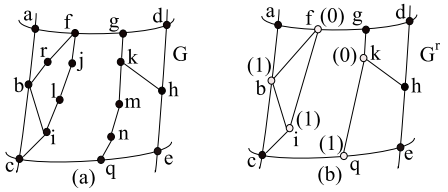


Fig. 5. Figure (a) shows the input graph $G$. Figure (b) shows the reduced graph $G^r$. In Figure (b), nodes filled in black color indicate *free* nodes and the other nodes are *active* nodes. The numbers on the active nodes in Figure (b) indicate the BFS level number with $f$ and $k$ as the source nodes.

Further, we can seek an order of the nodes within $H$ also as follows. Consider a subset $S \subseteq V(H)$ and $\mathrm{Dep}(S)$. Once the post-processing of nodes in $\mathrm{Dep}(S)$ finishes, the arrays with respect to nodes in $S$ are no longer needed in memory. We now observe the following with respect to $S$ and $\mathrm{Dep}(S)$.

For a node $x \in \mathrm{Dep}(S)$, the nodes $v := \mathrm{left}(x)$ and $w := \mathrm{right}(x)$ are neighbors in $H$. Therefore, it follows that $v$ and $w$ appear in either the same level or in consecutive levels of a BFS of $H$. These observations allows us to define a order on the nodes of $H$ so that we can choose appropriate subsets $S$ that reduce the amount of storage required by our algorithm.

To this end, we perform a BFS in $H$ and arrange the nodes of $H$ into sets $L_0, L_1, \cdots$, such that nodes in $L_i$ for $i \geq 0$ are at a distance of exactly $i$ from the source node $s \in H$ of the BFS. (The choice of $s$ is immaterial to our discussion.) We can start with $S_1 = L_0 \cup L_1$ and compute $\mathrm{Dep}(S_1)$ as defined. Once the post-processing of nodes in $\mathrm{Dep}(S_1)$ finishes, we define $S_2 = L_1 \cup L_2$ and perform the post-processing of nodes in $\mathrm{Dep}(S_2)$. While doing so, we retain the arrays corresponding to nodes in $L_1$ in memory and remove those corresponding to nodes in $L_0$. In addition, we have to keep the arrays of nodes in $L_2$ in memory. In general, when performing post-processing of nodes in $\mathrm{Dep}(S_i)$, $i \geq 1$, we need arrays for nodes in $L_{i-1} \cup L_i$. Thus, the space required for our implementation is in $O(\max_i |L_{i-1} \cup L_i| \cdot n)$.

The two techniques reduce our space requirement significantly. In most real-world graphs, the technique illustrated in Sections III-D1, III-D2 reduces amount of storage required by 76% and a further 82% on average respectively.

In our implementation, in the post-processing phase, recall that for a node $x \in G \setminus G^r$, we compute the arrays $S_x$ using the arrays $S_{\ell_x}$ and $S_{r_x}$ where $\ell_x = \mathrm{left}(x)$ and $r_x = \mathrm{right}(x)$. To do so, we use one SMX for each node $x$. Threads within an SMX compute the array $S_x$ as explained in Section III-C3. A similar approach is followed for computing the arrays $D_x$, and $\sigma_x$.

### E. Results

*1) Platform:* We use the GPU described in Section II-B1. For comparison studies with respect to libraries running on

multicore CPUs, we use an Intel(R) Xeon(R) E5-2650 CPU with 128 GB RAM and a memory bandwidth of 68 GB/s for our experiments. The E5-2650 CPU is a dual processor where each processor has 10 cores and each core can process two threads using hyper threading. Each core operates at 2.34 GHz which can be boosted to 3 GHz using turbo boost technology. The E5-2650 CPU has 64 KB L1 cache per core, 256 KB L2 cache per core and a shared 25 MB L3 cache.

*2) Datasets:* We experiment with graphs from the dataset of sparse graphs from the University of Florida dataset [1]. Since we require the graph to be biconnected, we run algorithms on the largest biconnected component of the graphs listed Table II. Since graphs in the dataset from Table II have a large biconnected component that spans more than 80% of the edges, as indicated by numbers shown in column labeled **Largest BCC**, the size of the graph that we run our algorithm is not significantly compromised.

| **Graph name** | $\|V\|$ | $\|E\|$ | **Largest BCC** | | |
|---|---|---|---|---|---|
| | | | $\|V\|$ | $\|E\|$ | **%Deg=2** |
| roadNet-CA | 2.0 M | 2.7 M | 1.57 M | 2.34 M | 24.0 |
| roadNet-TX | 1.4 M | 1.9 M | 1.05 M | 1.57 M | 25.0 |
| soc-Epinions1 | 76 K | 508 K | 36 K | 365 K | 27 |
| patents_main | 241 K | 560 K | 151 K | 474 K | 26.1 |
| coAuthorsDBLP | 299 K | 977 K | 198 K | 818 K | 15.4 |
| soc-Slashdot0902 | 82 K | 474 K | 515K | 473K | 23.0 |
| caidaRouterLevel | 192 K | 609 K | 132K | 541 K | 27.3 |
| scircuit | 171 K | 479 K | 135 K | 335 K | 13.5 |
| soc-sign-epinions | 131 K | 841 K | 58 K | 642 K | 27.7 |
| p2p-Gnutella31 | 62 K | 147 K | 33 K | 119 K | 27.7 |

TABLE II
LIST OF GRAPHS THAT WE USE IN OUR EXPERIMENTS. IN THIS TABLE, THE NUMBER OF NODES AND THE NUMBER OF EDGES ARE ROUNDED TO THE NEAREST THOUSAND (K) OR THE NEAREST MILLION (M). THE LAST COLUMN INDICATES THE PERCENTAGE OF NODES THAT ARE ELIMINATED FROM THE LARGEST BCC DURING THE PREPROCESSING STEP.

*3) Results:* We compare the results of our algorithm, labeled as "OUR" in the rest of this section, with a wide range of algorithms listed in the following.

*a) Bader and Mc. Laughlin [21]:* This work is currently one of the fastest for computing betweenness-centrality on a GPU. We use the software from the authors of [21] in our comparison. This result is labeled "BM15" in the rest of the section.

*b) Gunrock Library [32]:* Gunrock is a GPU based library for graph algorithms, which contains a routine for computing betweenness-centrality. We use the software from [32] and label this result as "Gunrock" in the rest of this section.

*c) APGRE [31] and Ligra [19]:* Wang et al. [31] extend the work of Sariyuce et al. [28] to multi-core CPUs. We implement the algorithm of [31] on the CPU described in Section III-E1 with a thread per core and label this result as "APGRE". The Ligra library [19] consists of routines for graph algorithms and runs on multicore CPUs. On our dataset, as we observed that APGRE is consistently faster than Ligra, we show only the results from APGRE [31] [1].

On the graphs from Table II, the overall time taken by the above algorithms on the largest biconnected component is plotted in Figure 6(a)[2]. The Y-axis of Figure 6(a) is on a logarithmic scale. The secondary Y axis of Figure 6(a) shows the speedup of our algorithm over the **best** of above mentioned baseline algorithms.

The relative speedup obtained by our algorithm with respect to individual algorithms is shown in Table III. The columns under the head "Largest BCC" refer to the speedup on each of our instance with respect to the algorithm of Bader and McLaughlin [21], the Gunrock library [32] and the algorithm of Wang et al. [31]. As can be seen from Table III, the average speedup achieved is 1.6x, 2.08x and 1.96x with respect to [21], [32], and [31] respectively.

The throughput achieved by the algorithms under study is shown in Figure 6(b). The throughput of an algorithm for computing the betweenness-centrality on a graph $G$ of $n$ nodes and $m$ edges is measured as $\frac{n \cdot m}{t}$ Traversed Edges Per Second (TEPS) where $t$ is the time taken in seconds by the algorithm. The quantity MTEPS refers to Million TEPS. As can be seen from Figure 6(b), our algorithm achieves a higher MTEPS compared to the other three algorithms.

*Performance on Synthetic Datasets:* To understand how the number of nodes eliminated in the preprocessing step of Algorithm 3 can impact the speedup achieved, we construct a synthetic graph of $n$ nodes with average degree $d$ as follows. A cycle graph on $n$ nodes ensures that the graph will have only one biconnected component. On this cycle graph, we mark an t% of nodes as nodes that will have a degree of two. The degree of the rest of the unmarked nodes is increased by adding edges to pairs of nodes chosen uniformly at random. We ensure that each unmarked node has degree at least three.

We study the results of our approach on synthetic graphs of size ranging from 100 K nodes to 300 K nodes with an average degree of 20 to 30. We vary the percentage of nodes that can be removed from the reduction step from 10% to 50%. We study the speedup of Algorithm 3 with respect to that of Bader and Mc. Laughlin [21]. As shown in Figure 7, for a fixed $n$, as the percentage of nodes of degree two increases, the speedup achieved by our algorithm also increases.

### F. Extending our Approach to General Graphs

So far, we have assumed that our input graph is biconnected. In general, however, most real-world graphs are not biconnected. In this section, we briefly show how to extend our techniques to non-biconnected graphs. We start by reviewing the BADIOS framework of Sariyuce et al. [28] and the APGRE framework of Wang et al. [31] that we use in our solution.

*1) The BADIOS and the APGRE Framework:* The main idea of the BADIOS framework [28], called as APGRE framework in [31], is to decompose a graph $G$ into its biconnected components (BCCs) and use Brandes algorithm [6] on the individual biconnected components.

In the algorithm of Sariyuce et al., each BCC has a copy of the articulation point, called as an *alias* node, which connects it to other neighbouring BCC's in the original graph $G$.

[1]Detailed timings are provided at cstar.iiit.ac.in/∼kkishore/paper.pdf

[2]Note that in [21], absolute time taken as mentioned are normalized to 8192 iterations. Similarly, the timings shown in [32] are normalized to one iteration.

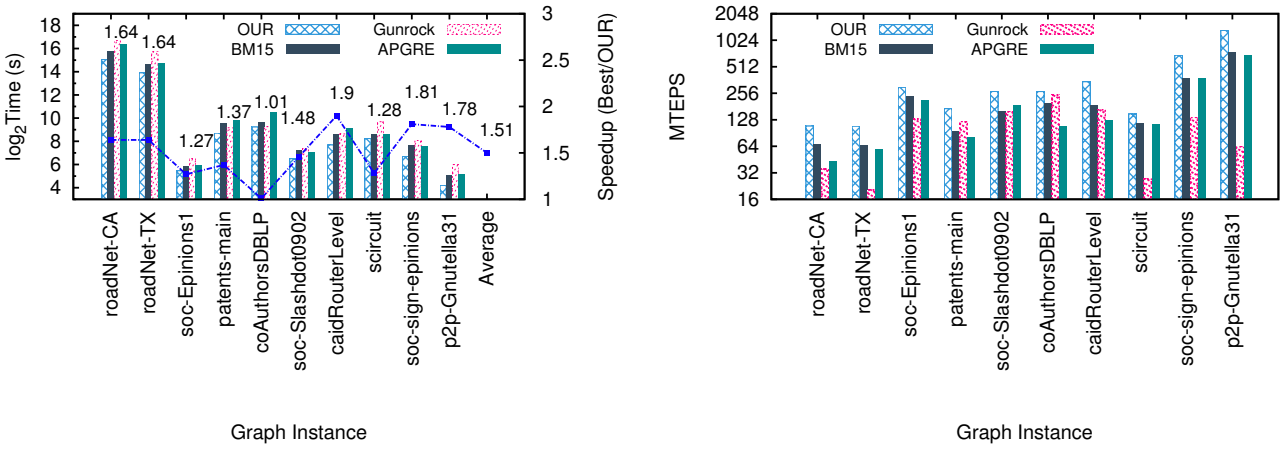


Fig. 6. Comparing the overall performance improvement of Algorithm 3 with respect to that of [21], [32], [31], [19]. The plot on the left (*resp.* right) shows the absolute time (MTEPS) achieved by our algorithm, BM15 [21], Gunrock [32], and APGRE [31], in that order, respectively. The last instance on the X-axis of part (a) of the figure shows the average speedup of our algorithm over the best of the other three algorithms.

| | Speed up with respect to | | | | | |
|---|---|---|---|---|---|---|
| **Graph name** | **Largest BCC** | | | **Entire Graph** | | |
| | **BM15** | **GUNROCK** | **APGRE** | **BM15** | **GUNROCK** | **APGRE** |
| roadNet-CA | 1.62 | 3.06 | 1.61 | 1.81 | 3.56 | 2.62 |
| roadNet-TX | 1.64 | 5.26 | 1.80 | 1.81 | 6.94 | 2.15 |
| soc-Epinions1 | 1.30 | 2.32 | 1.41 | 2.59 | 6.03 | 1.92 |
| patents_main | 1.81 | 1.38 | 2.12 | 1.80 | 1.43 | 1.28 |
| coAuthorsDBLP | 1.35 | 1.09 | 2.46 | 2.05 | 2.02 | 2.88 |
| soc-Slashdot0902 | 1.69 | 1.70 | 1.45 | 2.15 | 2.56 | 1.64 |
| caidaRouterLevel | 1.83 | 2.07 | 2.73 | 2.53 | 2.98 | 2.30 |
| scircuit | 1.29 | 5.41 | 1.29 | 1.50 | 7.07 | 1.57 |
| soc-sign-epinions | 1.81 | 2.63 | 1.84 | 3.47 | 6.24 | 2.09 |
| p2p-Gnutella31 | 1.78 | 3.43 | 1.92 | 3.89 | 10.56 | 2.30 |
| **Average** | **1.60** | **2.08** | **1.96** | **2.06** | **3.44** | **2.04** |

TABLE III
THIS TABLE SHOWS THE RELATIVE PERFORMANCE OF OUR ALGORITHM, LABELED OUR, OVER [21] LABELED "BM15", [32] LABELED "GUNROCK" AND [31] LABELED "APGRE" ON THE LARGEST BCC AND THE ENTIRE GRAPH.

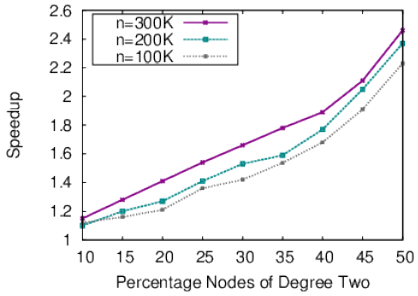


Fig. 7. The relative performance obtained by Algorithm 3 over [21] on synthetic graphs.

A reachability metric is defined for alias nodes as follows. Consider the $i$th BCC $G_i = (V_i, E_i)$ of $G$ let $v' \in V_i$ be an alias node. Consider any node $u \in V_i$ that is different from $v'$. The reachability metric for $v'$, denoted $\text{reach}(v')$, is set to the number of nodes $x \in V \setminus V_i$ such that the path between $u$ and $x$ passes through $v'$. (Note that the choice of $u$ is immaterial in the above.) Reach values for every alias node is computed by a leaf-to-root traversal of the block tree $T$ as described by Puzis et al. [25]. The reachability metric is useful in extending the betweenness centrality values computed on the BCCs of $G$ to the entire graph.

*2) Our Algorithm for General Graphs:* To use the framework of BADIOS [28] or APGRE [31], we start by decomposing the input graph $G$ into its biconnected components, $G_1, G_2, \cdots,$. Since each of these components, $G_i$, $i \geq 1$, are biconnected, they possess an ear decomposition. So, each $G_i$ can be taken as input to Algorithm 3 to compute the betweenness-centrality values of nodes in $G_i$. At this point, once we have the reachability values for alias nodes as is done in [28], [31], it will be possible to extend the betweenness-centrality values of nodes with respect to each $G_i$ to the entire $G$. One can view this approach as having two preprocessing steps: the first step decomposes $G$ into its biconnected components, the second step applies ear decomposition on each component. In the processing step, betweenness-centrality values with respect to each biconnected component is computed similar to the processing phase of Algorithm 3. Finally, we have two post-processing steps: first that is similar to the post-processing phase of Algorithm 3, and the second one similar to that of the corresponding step in [28], [31].

*3) Results:* We reconsider the graphs listed in Table II and apply our approach to compute betweenness-centrality. We use the experimental platform mentioned in Section II-B1. For performance comparison, we consider the algorithms listed in Section III-E3. The results are shown in Figure 8.

Figure 8(a) shows the time taken by our algorithm and the other algorithms on the graphs listed in Table II. The numbers in Figure 8(a) show the speedup achieved by our algorithm compared to the **best** of the other three algorithms. The speedup with respect to each of the algorithms is shown in Table III in the columns labeled "Entire Graph". The speedup achieved in the case of the entire graph is higher than the speedup achieved on the largest biconnected component of the corresponding graph as can be seen from Figures 8(a) and 6(a). The reason for this is that when using the algorithms from [21], [32], every BFS has to run on the entire graph. In our algorithm, and also that of [31], each BFS runs only local

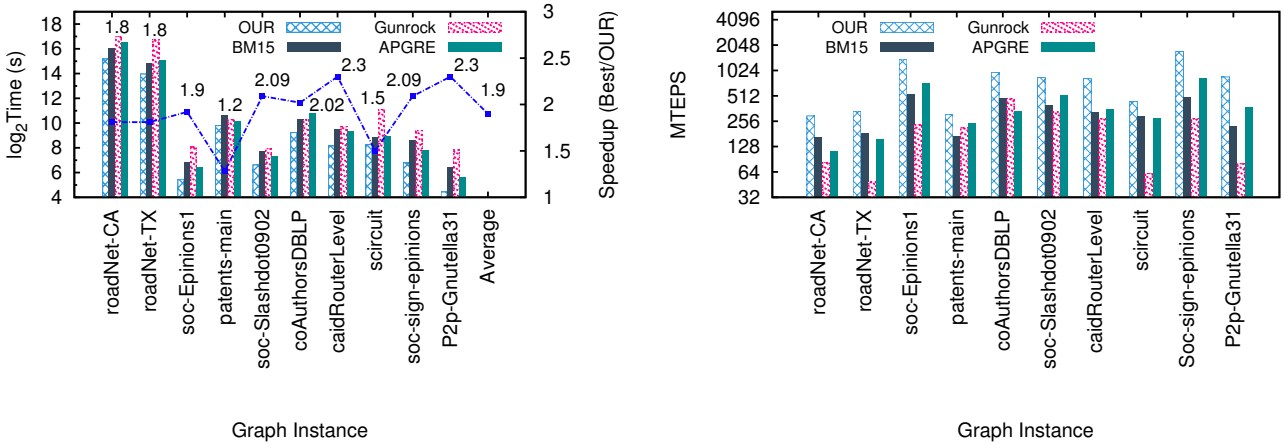


Fig. 8. Comparing the overall performance improvement of Algorithm 3 with respect to that of [21], [32], [31], [19]. The plot on the left (*resp.* right) shows the absolute time (MTEPS) achieved by our algorithm, BM15 [21], Gunrock [32], and APGRE [31], in that order, respectively. The last instance on the X-axis of part (a) of the figure shows the average speedup of our algorithm over the best of the other three algorithms. These results are for the entire graph.

to a biconnected component. The throughput of our algorithm as MTEPS is shown in Figure 8(b) along with the throughput achieved by the other algorithms.

## IV. Conclusions

In this paper, we studied the ear decomposition of a graph and its application to finding the betweenness-centrality of nodes in a graph. Our results indicate that for problems such as betweenness-centrality, using an ear decomposition is effective and practical. We believe that our technique is of independent interest and can be applied to other graph problems.

## References


[1] The University of Florida Sparse Matrix Collection. https://www.cise.ufl.edu/research/sparse/matrices/.

[2] A. E.Saryuce, E.Saule, K., and V.Catalyurek, U. Betweenness centrality on GPUs and heterogeneous architectures. In *Proc. W. GPGPU*, 2013, pp. 76–85.

[3] Bader, D. A., Illendula, A. K., Moret, B. M., and Weisse-Bernstein, N. R. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In *Algorithm Engineering*. Springer, 2001, pp. 129–144.

[4] Banerjee, D. S., Kumar, A., Chaitanya, M., Sharma, S., and Kothapalli, K. Work efficient parallel algorithms for large graph exploration on emerging heterogeneous architectures. *JPDC*, V. 76(2015), pp. 81–93.

[5] Bollobas, B. Random Graphs, Cambridge University Press, 2001.

[6] Brandes, U. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology 25* (2001), 163–177.

[7] Chaitanya, M., and Kothapalli, K. Efficient multicore algorithms for identifying biconnected components. *IJNC* 6:1(2016), pp: 87–106.

[8] Chhugani, J., Satish, N., Kim, C., Sewall, J., and Dubey, P. Fast and efficient graph traversal algorithm for CPUs: Maximizing Single-Node Efficiency. in Proc. IPDPS, pp. 378–387, 2012.

[9] Cong, G., and Bader, D. A. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps). In *Proc. IEEE IPDPS* (2005).

[10] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. Introduction to algorithms, 2001.

[11] Bader, D. A., and Madduri, K. GTgraph: A synthetic graph generator suite. (2006).

[12] Djidjev, H., Thulasidasan, S., Chapuis, G., Andonov, R., and Lavenier, D. Efficient multi-GPU computation of all-pairs shortest paths. In *Proc. of IEEE IPDPS* (2014).

[13] Hong, S., Rodia, N. C., and Olukotun, K. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *in Proc. SC* (2013).

[14] J.-K. Lou, S. D. Lin, K. T. C., and Lei, C. L. What can the temporal social behavior tell us? An estimation of vertex-betweenness using dynamic social information, 2010.

[15] Jaja, J. An introduction to parallel algorithms. Addison-Wesley, 2004.

[16] Jia, Y., Lu, V., Hoberock, J., Garland, M., and Hart, J. C. Edge v. node parallelism for graph centrality metrics. *GPU Computing Gems 2* (2011), 15–30.

[17] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 1 (1998), 359–392.

[18] Koschutzki, D., and Schreiber, F. Centrality analysis methods for biological networks and their application to gene regulatory networks. *Gene Regulation and Systems Biology 2* (2008).

[19] Shun, J., and Blelloch, G. E. Ligra: a lightweight graph processing framework for shared memory. *PPOPP* (2013).

[20] McLaughlin, A., and Bader, D. A. Scalable and high performance betweenness centrality on the GPU. In *Proc. ACM SC*, 2014, pp. 572–583.

[21] McLaughlin, A., and Bader, D. A. Fast execution of simultaneous breadth-first searches on sparse graphs. In *Proc. ICPADS*, 2015, pp. 9–18.

[22] Merrill, D., Garland, M., and Grimshaw, A. Scalable GPU graph traversal. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 117–128.

[23] NVidia Corporation. https://www.nvidia.in/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001_v03.pdf

[24] NVidia Corporation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[25] Puzis, R., Zilberman, P., Elovici, Y., Dolev, S., and Brandes, U. Heuristics for speeding up betweenness centrality computation. In *Proc. SOCIALCOM-PASSAT*, 2012, pp. 302–311.

[26] Ramachandran, V. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. Morgan Kaufmann Publishers Inc., 1993.

[27] Ren, D. Q. Algorithm level power efficiency optimization for CPU–GPU processing element in data intensive SIMD/SPMD computing. *JPDC 71*, 2 (2011), 245–253.

[28] Sariyüce, A. E., Saule, E., Kaya, K., and Çatalyürek, Ü. V. Shattering and compressing networks for betweenness centrality. In *SIAM Data Mining Conference (SDM). SIAM* (2013), SIAM.

[29] Shi, Z., and Zhang, B. Fast network centrality analysis using GPUs. *BMC bioinformatics 12*, 1 (2011).

[30] Soman, J., Kothapalli, K., and Narayanan, P. Some GPU algorithms for graph connected components and spanning tree. *Parallel Processing Letters 20*, 04 (2010), 325–339.

[31] Wang, L., Yang, F., Zhuang, L., Cui, H., Lv, F., and Feng, X. Articulation points guided redundancy elimination for betweenness centrality. In Proc. ACM PPoPP, pp.1–13, 2016.

[32] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Not.* (2015), vol. 50, pp. 265–266.