Parallel Ear Decomposition and Application to Betweenness Centrality.

Thesis submitted in partial fulfillment
of the requirements for the degree of

MS by Research
in
Computer Science & Engineering

by

Charudatt Pachorkar
201307687
charudatt.p@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, India
January 2017

Copyright © Charudatt Pachorkar, 2016
All Rights Reserved

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Parallel Ear Decomposition and Application to Betweenness Centrality" by Charudatt Pachorkar, has been carried out under my supervision and is not submitted elsewhere for a degree.

______________ Date

______________________________ Adviser: Dr. Kishore Kothapalli

Acknowledgments

I would like to express my sincere gratitude to my thesis advisor Dr. Kishore Kothapalli for his excellent guidance, support and motivation. This thesis would not have been possible without his extensive motivation, patience, immense knowledge and positive criticism. He has not only taught me how to tackle a problem, but also some fundamental skills which are extremely useful for a researcher. I take this opportunity to thank him for his guidance.

I am grateful to all faculty members of CSTAR-Lab- Dr. Kannan Srinathan, Dr. Ashok Kumar Das and Dr. Sanjay Rawat for yielding wonderful research center. I would like to thank Dr. Ashok Kumar Das for not only teaching me courses like System and Network Security and Principle of Information Security, but also giving me a wonderful opportunity to be TA for Principle and Information Security.

It was a wonderful opportunity to work with my all lab mates. I especially want to thank Meher for snapping me out of boundless loops and re-instating my self belief. I would like to thank Debarshi and Kiran for being available all the time for both technical and informal discussions. Many thanks to Rajesh, Teja, Spandan, Satwik, Nayyar, Hardik for all the valuable discussions. Special thanks to my friends Mallikarjun, Brij, Anirudh and more for all the memories.

Last but not the least, I want to thank my family for always being supportive.

Abstract

In recent years, the amount of information generated and stored is inconceivable. A plethora representations are available to store such kind of data, with graphs being one of the most efficient approaches, useful for studying and analysing various attributes of data. Considering amount of data to process, parallel computing is the most suitable option but many state of the art algorithms are not efficient enough to take maximum advantage of available architectures. Hence researchers are focusing on development and enhancement of parallel graph algorithms. Parallel graph algorithms continue to attract a lot of attention in research given their applications to several fields of sciences and engineering. However, efficient design and implementations of graph algorithms on modern many-core accelerators has to 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 algorithm for obtaining an ear decomposition of a graph. Our implementation of the proposed algorithm on an NVidia Tesla K40c and Intel Xeon E5-2650 improves the state-of-the-art by a factor of 2.3x and 2.02x on average respectively 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.

Contents

Chapter Page

1 Introduction . . . 1
  1.1 Preliminaries . . . 2
  1.2 Betweenness Centrality . . . 3
    1.2.1 Brandes Algorithm . . . 4
  1.3 Ear Decomposition . . . 5
  1.4 Contributions . . . 6
    1.4.1 Organization of the Thesis . . . 6

2 Ear Decomposition . . . 7
  2.1 Related Work . . . 7
    2.1.1 Vijaya's Algorithm . . . 8
    2.1.2 PRAM Algorithm . . . 10
  2.2 Contribution . . . 10
    2.2.1 Pruning Method . . . 10
    2.2.2 Our Algorithm . . . 13
      2.2.2.1 Phase I . . . 13
      2.2.2.2 Phase II . . . 13
      2.2.2.3 Phase III . . . 14
  2.3 Experimental Results and Analysis . . . 14
    2.3.1 Platform Details . . . 14
    2.3.2 Real World and Synthetic Dataset . . . 15
      2.3.2.1 Results . . . 15

3 Betweenness Centrality . . . 21
  3.1 Related Work . . . 21
    3.1.1 Bader and McLaughlin algorithm (BM15) . . . 22
  3.2 Contribution . . . 22
    3.2.1 Our Algorithm . . . 22
      3.2.1.1 Preprocessing . . . 24
      3.2.1.2 Processing . . . 24
      3.2.1.3 Post-processing . . . 24
    3.2.2 Implementation Details . . . 25
      3.2.2.1 Classifying Nodes in G^r . . . 25
      3.2.2.2 Orchestrating Nodes in the Processing Phase . . . 26
    3.2.3 Our Algorithm Extended to General Graphs . . . 27

3.2.3.1 The BADIOS and the APGRE Framework 27
3.2.3.2 Our Algorithm for General Graphs 28
3.3 Experimental Results 29
3.3.1 Platform Details 29
3.3.2 Dataset 29
3.3.2.1 Results 29
3.3.2.1.1 Bader and Mc. Laughlin [28] 31
3.3.2.1.2 Gunrock Library [43] 31
3.3.2.1.3 APGRE [42] and Ligra [25] 32

4 Conclusions and Future Work 38

Bibliography 40

List of Figures

Figure Page

1.1 An example of Betweenness centrality with centrality values belonging to each vertex. 4

1.2 Partial Betweenness centrality. 5

1.3 An example of an ear decomposition. The labels on edges in part (b) of the figure indicate the ear number they belong to. 5

2.1 An example of ear decomposition. 7

2.2 An example of vijaya's algorithm. Part (b) shows spanning tree T of graph G in part (a). Labels on vertices in part (b) are pre-order number. (c, d), (e, f) and (g, h) are non tree edges. 9

2.3 Ear decomposition result. Each tree edge is labelled with related ear number. 9

2.4 Grafting and pointer jumping operations applied to sample graphs. 11

2.5 An example of using Lemma 2.2.1. 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. . . 12

2.6 GPU 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 2.2. 18

2.7 CPU 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 2.2. 19

2.8 Part (a) shows regular lattice RL and part (b) shows regular triangulation RT graph. . 20

3.1 Figure (a) shows the input graph G with an ear decomposition where the number on the edges indicates the ear number they belong to. Figure (b) shows the reduced graph G^r. A parallel edge in the reduced graph $(a - c)$ is shown as the 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. 23

3.2 An illustration of our approach. 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 other 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. It can be noticed that the active nodes consist of two connected components: one containing nodes f, b, i and the other containing nodes k, q. 26
3.3 The relative performance obtained by Algorithm 2 over [28] on synthetic graphs. . . . 28
3.4 An example of reach value calculation. 29
3.5 Comparing the overall performance improvement of Algorithm 2 with respect to that of [28], [43], [42], [25]. The plot on the left (*resp.* right) shows the absolute time (MTEPS) achieved by our algorithm, BM15 [28], Gunrock [43], and APGRE [42], 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. 30
3.6 Comparing performance of Algorithm 2 with LIGRA on largest BCC. Part (a) shows absolute time of our algorithm with respect to LIGRA. Part (b) shows MTEPS achieved by our algorithm with respect to LIGRA. The last instance on the X-axis of part (a) of the figure shows the average speed-up. 33
3.7 Comparing performance of Algorithm 2 with LIGRA on complete graph. Part (a) shows absolute time of our algorithm with respect to LIGRA. Part (b) shows MTEPS achieved by our algorithm with respect to LIGRA. The last instance on the X-axis of part (a) of the figure shows the average speedup. 34
3.8 Comparing the overall performance improvement of Algorithm 2 with respect to that of [28], [43], [42], [25]. The plot on the left (*resp.* right) shows the absolute time (MTEPS) achieved by our algorithm, BM15 [28], Gunrock [43], and APGRE [42], 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. 35

List of Tables

Table Page

2.1 Fundamental cycle representation formed by non tree edges (e, f), (c, d) and (g, h). . . 8
2.2 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 number in column labeled "Edges Pruned" shows the number of edges that are deemed redundant according to Lemma 2.2.1. The number in the column labeled "Avg. Dist." refers to the average number of tree edges traversed to find the LCA of the end points of a nontree edge. The number in the last column labeled "ACE" indicates the average number of fundamental cycles according to a BFS tree that pass through a tree edge. 16
2.3 This table shows absolute time (in msec) for ear decomposition on GPU. Fist column shows time for OUR approach followed by time for [28] with pruning which is followed only [28]. 17
2.4 This table shows absolute time (in msec) for ear decomposition on CPU. Fist column shows time for OUR approach followed by time for [28] with pruning which is followed only [28]. 17
2.5 This table shows absolute time (in msec) for ear decomposition [4] dataset. Fist column shows time for OUR approach followed by time for PRAM algorithm. Last column indicates average number of tree edges traversed to find the LCA of the end points of a nontree edge 18

3.1 It shows reduction in memory requirement for different stages of our algorithm. 27
3.2 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 near thousand (K) or the nearest million (M). The Last column indicates the percentage of nodes that are eliminated from the largest BCC during our reduction step. 31
3.3 This table shows the relative performance of OUR algorithm, labeled OUR, over [28] labeled "BM15", [43] labeled "GUNROCK" and [42] labeled "APGRE" on the largest BCC and the complete graph. 36
3.4 This table shows absolute time of OUR algorithm, labeled OUR, BM15, GUNROCK, APGRE and LIGRA on the largest BCC. 36
3.5 This table shows absolute time of OUR algorithm, labeled OUR, BM15, GUNROCK, APGRE and LIGRA on the complete graph 37

Chapter 1

Introduction

In the most simplest way a graph can be explained as a set of vertices V and edges E such that vertices $u, v \in V$ are connected by edge $e \in E$ in undirected or directed fashion. Many real world practical problems can be represented in terms of graphs. Sometimes in alternate terminology a graph is referred to as a network in which attributes of the network are associated with nodes of a graph. In computer science, graphs are used to represent social network, communication, organization of data and much more. For instance, connections of a person on social media can be represented by directed graphs where each person represents a node and connections to other persons indicate edges. In similar way, many problems in biology, chemistry, physics, astrology etc. boil down to graphs. Therefore development of graph algorithms has great importance in graph theory.

In recent years, the amount of information that is being generated and stored is highly inconceivable. To analyse such vast amounts of information, graphs are extremely useful. This has started off an entirely new branch in graph theory called $graph\ analytic$. In order to understand the importance of graph analytics one should know how it is different from relational analytics. Simply put, relational analytics deals with one to one or one to many relationships. For example, relational analytics can be used to find a person and his/her friend or friends in a network but it is difficult to find second level of indirect friends. In such situations graph analytics is highly useful because it compares many to many relations. To study complex relations which are not easily visible using relational analytics, graph analytics can be is used.

Graph analytics has numerous applications and approaches. The vast plethora of applications include identifying communities, identifying connections between different nodes, etc. One of such a important approach is to find influencers in the network. In recent years this approach has been used by many researchers because of its tremendous impact on fields such as social media, communication, transport. In formal/technical terms, it is known as betweenness centrality. It is based on number of shortest paths passing through a node in the network.

Betweenness centrality of a node in the network depends on its position. A node can gain more power by occupying advantageous positions in the network. If there is single shortest path between source and destination passing through a node say u then u has high importance. On the other way, if

there are multiple paths between source and destination such that one of them is passing through u then importance of u is comparatively reduced. Unfortunately, the fastest known algorithm for computation of betweenness centrality takes high time complexity. Section 1.2 describes it in detail. Hence there is need of other approaches to reduce computation time for betweenness centrality.

One of the impressive approaches is network decomposition and reduction. It means, divide the network into multiple sub-networks in such a way that each sub-network can be processed independently. Processing of highly independent work can be accelerated further by using modern multi-core and many-core platforms. In recent years, parallel computing has evolved remarkably. As power consumption by processors has become vital issue in recent years, parallel computing became the important paradigm. Nvidia's GPU and Intel's Xeon processors are dominant example of this. From here on out, additional speed will come from additional cores but using these core in efficient and scalable way is a challenging task. To achieve this, exclusive parallel programming languages such as CUDA, OpenMP and MPI are playing a crucial role. They allow the user to focus on design of parallel and data distribution among processors by hiding underlying complexities. In this work, we have used Nvidia's K40 GPU and Intel's Xeon Phi processor. For programming on these many-core and multi-core processors we have used CUDA and OpenMp respectively.

Processing time within decomposed components can be reduced further by applying graph reduction methods. Selection of reduction methods depends on the application at hand. For instance, removal of identical vertices [37] to speedup betweenness centrality computation is useful in unweighted graphs but not in weighted graphs. Another approach is to hide degree two vertices during main processing phase and compute result for them in post processing phase. This approach is highly useful in applications where result for degree two nodes can be computed using adjacent node with degree greater than two. Out of the multiple ways available to hide degree two vertices but the most systematic and efficient way is ear decomposition. It is described in Section 1.3.

In summary, this thesis shows that as a result of using graph decomposition and reduction techniques implemented on modern high performance multi-core and many-core architectures, the processing time of high complexity algorithms such as betweenness centrality can be significantly reduced.

1.1 Preliminaries

This section describes basic definitions.

$G = (V, E)$ where V is set of vertices and E is set of edges called a $directed\ graph$ if it consists edges of form edge $(u, v) \in E$ such that it is directed from u to v i.e. outgoing from u and coming to v. If edges are unordered pair of vertices it is called an $undirected\ graph$. Number of edges incident on vertex indicates the $degree$ of a vertex.

$G' = (V', E')$ is called $subgraph$ of $G = (V, E)$ if $V' \subset V$ and $E' \subset E$. $Induced\ subgraph$ $H = (V', M)$ is sub-graph of G induced by V' where $M = \{(i, j) \in E | i, j \in V'\}$. G is $connected$ in

directed or undirected way if there exist path from each vertex to every other vertex. Maximal induced subgraph of connected G is called $connected\ component$.

A connected graph without cycle is called a $tree$. Let $T = (V, E)$ be a tree rooted at $s \in V$. $T = (V, E, s)$ is called $out - tree$ if all edges of tree are directed away from root node. If they are directed towards root, T is called $in - tree$. Vertex with degree one is called a $leaf$ vertex. Number of edges between root vertex and $v \neq s$ and $v \in V$ gives $level$ of vertex v. If vertex v is descendent of vertex u (i.e. u is ancestor of v) and direct edge exist between u and v then u is called $parent$ of v. It means level difference between $level\ of\ u$ and $level\ of\ v$ is one. $Least\ common\ ancestor\ lca(u, v)$ is a vertex say $p \in V$ which is ancestor of both u and v such that p does not have a child which is an ancestor for both u and v. For $e = (u, v)$ least common ancestor is denoted by $lca(u, v)$ or $lca(e)$. For a spanning tree T $non\ tree\ edge$ is an edge in $G \setminus T$.

Let T be a spanning tree and $e = (u, v) \in E$ is a non tree edge of G. Cycle created by any non tree edge with tree edges of T $(T \cup e)$ is called a $fundamental\ cycle$.

$Cutedge$ or $bridge$ is an edge $e \in E$ of a graph $G = (V, E)$ such that the removal of e splits G into one or more sub-graphs. Similarly $cutvertex$ or $articulation\ vertex$ is a vertex $v \in V$ of G such that removal of v splits G into one or more sub-graphs. A connected graph G is called $2 - edge\ connected$ if it does not contain $cutedge$. Similarly, connected graph G is called $biconnected$ or $2 - vertex\ connected$ if it has no $articulation\ vertex$.

1.2 Betweenness Centrality

Betweenness Centrality indicates importance of node in a graph as shown in Figure 1.1. It is based on number of shortest path from all vertices to all others passing through that node. Let $G = (V, E)$ be a connected graph with $|V|$ number of vertices and $|E|$ number of edges then betweenness centrality of a node $v \in V$ can be shown as below.

$$bc(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1.1}$$

where σ_{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. Assuming communication between nodes follows shortest path, node with high betweenness centrality value has a large influence on communication between nodes. This concept has crucial impact on domains such as social networks, biology, transport etc. It has been used to find best suitable locations for stores in city [32], to investigate growth of AIDS in sexual networks [24] and community detection [41]. A social network is a social structure made up of sets of dyadic ties, set of actors who can be any arbitrary entities like person or organization and set of interactions between actors. Graph theoretic concepts are used to understand and explain social phenomenon to analyse social network. Betweenness centrality is used to rank the entities in a network according to their positions based on number of shortest path passing through them. To

compute betweenness centrality of all nodes in graph, shortest path calculations between all pairs of vertices is required. FloydWarshall [11] algorithm can be used to calculate shortest path between all pairs of vertices. Predecessor matrix is updated while calculating all pair shortest path to calculate betweenness centrality of every node in shortest path. Time and space complexity of this approach on a graph G is $O(V^3)$ and $O(V^2)$ respectively. Johnson's algorithm [11] can be used on sparse graph taking $O(V^2 logV + VE)$ time. Brandes [7] has improved upon these algorithm with $O(V + E)$ space complexity and time complexity $O(VE)$ for unweighted and $O(V^2 logV + VE)$ for weighted graph.

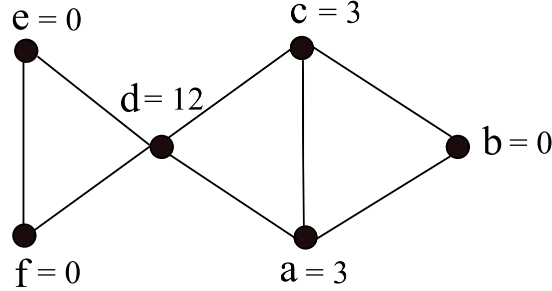


Figure 1.1 An example of Betweenness centrality with centrality values belonging to each vertex.

1.2.1 Brandes Algorithm

Brandes introduced an algorithm to compute betweenness centrality that runs in $O(V+E)$ space and $O(VE)$ sequential time where V and E indicates number of vertices and edges in a graph respectively. 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 breadth first search (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 σ_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 partial betweenness centrality of nodes in P_v. If there is exactly one shortest path from source node v to destination then dependency relation is

$$\delta(u) = \sum_{w:u \in P_v(w)} (1 + \delta(w)) \tag{1.2}$$

Similar relation can be given as follows for general case.

$$\delta(u) = \sum_{w:u \in P_v(w)} \frac{\sigma_{vu}}{\sigma_{vw}} (1 + \delta(w)) \tag{1.3}$$

where u is the parent of w in the shortest path tree rooted at v as shown in Figure 1.2. $\delta(w)$ 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.

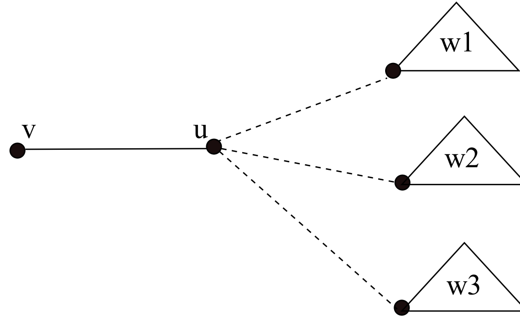


Figure 1.2 Partial Betweenness centrality.

1.3 Ear Decomposition

An ear decomposition of a biconnected graph $G = (V, E)$ is a partitioning of the edges of G into simple paths $P_0, P_1, \cdots$ as follows.

- 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 $\cup_{j=0}^{i-1} P_j$.

Paths $\in P_0, P_1, \cdots, P_{i-1}$ are called ears. Index of lowest numbered ear is given to vertex $v \in V$ and edge $e = (u, v)$ which is denoted by $ear(v)$ and $ear(u, v)$ or $ear(e)$ respectively.

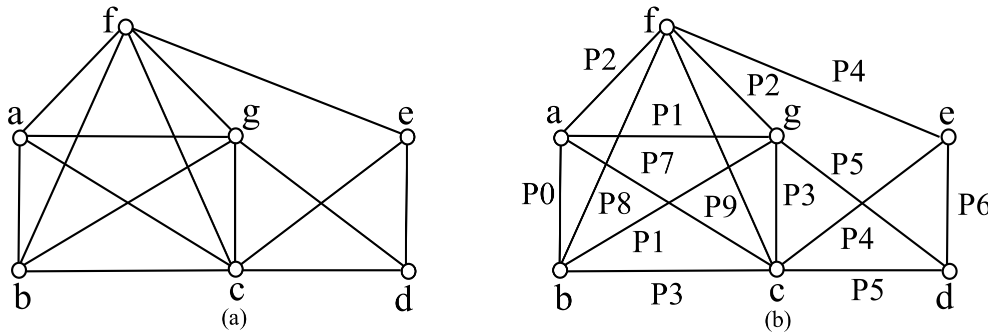


Figure 1.3 An example of an ear decomposition. The labels on edges in part (b) of the figure indicate the ear number they belong to.

An example is shown in Figure 1.3. An ear is called open ear if it does not contain cycle else called as closed ear. An ear with single edge called as trivial ear. In Figure 1.3 ears from $P6 - P9$ are trivial ears as they consist single edge. If all ears of ear decomposition are open, it is known as open ear decomposition. It is used to solve many important problem of graph theory such as st-numbering [26], graph planarity [35] and graph four connectivity [19].

Ear decomposition is also called as Whitney-Robbins synthesis because of their pioneering work. H. Robbins has used ear decomposition of 2-edge connected graph to prove a theorem called as $Robbins$ $theorem$. He proved G is 2-edge connected if and only if it has an ear decomposition. H.Whitney [44] proved that G is biconnected if and only if G has an open ear decomposition.

For directed graphs an ear is a directed path with all internal vertices having out degree and indegree one. A directed graph is strongly connected if and only if it has ear decomposition. It is strongly biconnected if and only if it has open ear decomposition.

If each ear has odd number of edges then it is called odd ear decomposition. In case of factor-critical graphs which has odd number of vertices such that removal of vertex v gives perfect matching L. Lovasz [22] proved that G is a factor- critical graph if and only if G has odd ear decomposition.

1.4 Contributions

In this paper, we show that the existing parallel algorithm for obtaining an ear decomposition 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. As a result of this characterization, the existing algorithm of 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. 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, which is a relative measure of the number of shortest paths that pass through a given node 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.

1.4.1 Organization of the Thesis

The rest of the thesis is organised as follows. In chapter 2 we first discussed some background work of ear decomposition followed by our naive algorithm and its implementations details. Initially chapter 3 explains about previous works to compute betweenness centrality followed by our method to compute betweenness centrality using ear decomposition. The thesis ends with concluding remarks in chapter 4.

Chapter 2

Ear Decomposition

An ear decomposition of an undirected graph G is segregation of its edges into set of ears such that one or two endpoints of an ear are part of earlier ears in sequence such that internal vertices of an ear are not part of any earlier ear as shown in Figure 2.1. Moreover, in most cases first ear in sequence is a cycle.

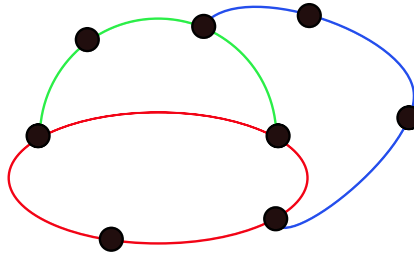


Figure 2.1 An example of ear decomposition.

The rest of the chapter is organised as follows. In Section 2.1 we first discussed some related work of ear decomposition. Section 2.2 describes our contribution followed by pruning method and our algorithm. Chapter ends with section 2.3, which shows our experiment results and analysis.

2.1 Related Work

Decomposition of graphs into subgraphs is a technique in parallel graph algorithms that is gaining significant research attention in recent years. Metis [18] and ParMetis [20] decompose a graph into a given number k of subgraphs such that the number of edges that cross a partition is minimized. This decomposition has been used in several graph algorithms lately as a subroutine [13, 39]. 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 [34] along with applications to problems such as planarity testing and triconnectivity. Bader [4] shows the results of implementing algorithm [34] on NPACI Sun E10K machines.

2.1.1 Vijaya's Algorithm

In this section we described Vijaya's parallel algorithm for finding ear decomposition. It outputs numbering of edges in E by assigning ear numbers. Let $G = (V, E)$ be a graph with $|V| = n$ vertices and $|E| = m$ edges.

1. Select vertex $v \in V$ as a root and find spanning tree T of G. Number vertices of T in pre-order from 1 to n-1.
2. To mark non tree edges in T with ear numbers.
   - Use least common ancestor $lca(e)$ in T to number each non tree edge e in G.
   - Relabel non tree edges number by sorting them in non-decreasing order as $1, 2 \cdots$.
3. To mark tree edges t with smallest number of non tree edge whose fundamental cycle consist t.
   - Mark each vertex with minimum number of non tree edge e incident on it.
   - Mark each tree edge $t = (v, parent(v))$ with minimum label from subtree rooted at v.
4. Non tree edge e numbered as 1 renumbered to 0.

Figure 2.2 shows an example of vijaya's algorithm. Figure 2.2 (a) is a sample graph G. Figure 2.2 (b) is spanning tree T of graph G rooted at vertex a. Vertices in T are labelled with pre-order number. Each non tree edge of T forms fundamental cycle with corresponding LCA vertex. Pre-order number of each LCA vertex is assigned to related non tree edge for sorting. Therefore, non tree edges (e, f), (c, d) and (g, h) are numbered as 1, 2 and 6 respectively. After sorting, these non tree edges are renumbered as $1, 2$ and 3. Table **??** shows fundamental cycle of each non tree edge. First two rows indicate tree edges of T. Third, Fourth and Fifth row represent tree edges present in fundamental cycle formed by non tree edges (e, f), (c, d) and (g, h). Tree edges (b, d) and (i, g) are shared between two fundamental cycles. Figure 2.3 shows ear decomposition where label on tree edge is an ear number it belong to.

a	a	b	d	i	g	e	b	c	i	h
b	i	d	e	g	f	f	c	d	h	g
1	1	1	1	1	1	1				
		2					2	2		
				3					3	3

Table 2.1 Fundamental cycle representation formed by non tree edges (e, f), (c, d) and (g, h).

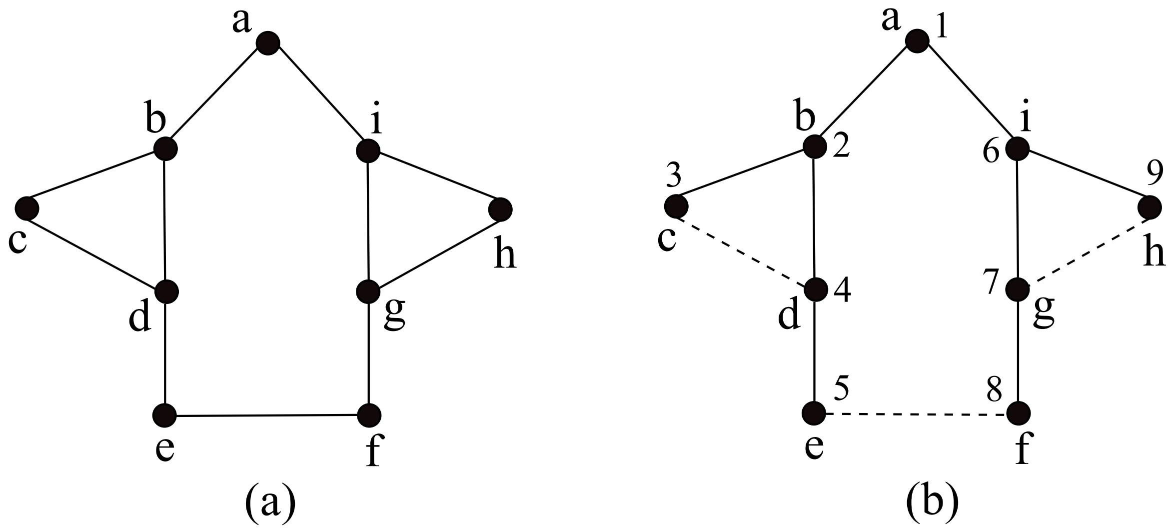


Figure 2.2 An example of vijaya's algorithm. Part (b) shows spanning tree T of graph G in part (a). Labels on vertices in part (b) are pre-order number. (c, d), (e, f) and (g, h) are non tree edges.

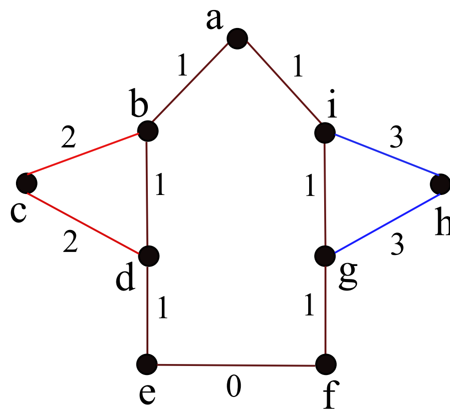


Figure 2.3 Ear decomposition result. Each tree edge is labelled with related ear number.

In a PRAM style of analysis, the algorithm has a runtime of $O(\log n)$ and uses $O(m + n)$ work. However, in a 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 [16]. 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 target a $O(\log n)$ 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 to 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.

2.1.2 PRAM Algorithm

Bader [4] has used similar approach as explained above except the computation of spanning tree. In this part of the algorithm vertices of graph $G = (V, E)$ are divided evenly across all processors. All processors can access entire graph as it is loaded into shared memory. $PF()$ is pseudoforest function which represents collection of trees on vertex set V. To manipulate pseudoforest $Grafting$ and $PointerJumping$ operations are used.

- **Grafting :** Let T_i and T_j be two non overlapping trees in set of trees, v_i is the root of T_i and $u_j \in T_j$ then $PF(v_i) \leftarrow u_j$ operation is called grafting of T_i onto T_j.
- **Pointer Jumping :** Let $v \in T$ then $PF(v) = PF(PF(v))$ is called pointer jumping operation.

Initially $PF(v) = v$ for each $v \in V$. In the next step grafting is used to graft possible rooted trees onto other trees. At every vertex pointer jumping is performed to reduce diameter of the tree. This algorithm ends when all vertices reside in a single tree. An example of grafting and pointer jumping is shown in Figure 2.4.

2.2 Contribution

2.2.1 Pruning Method

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

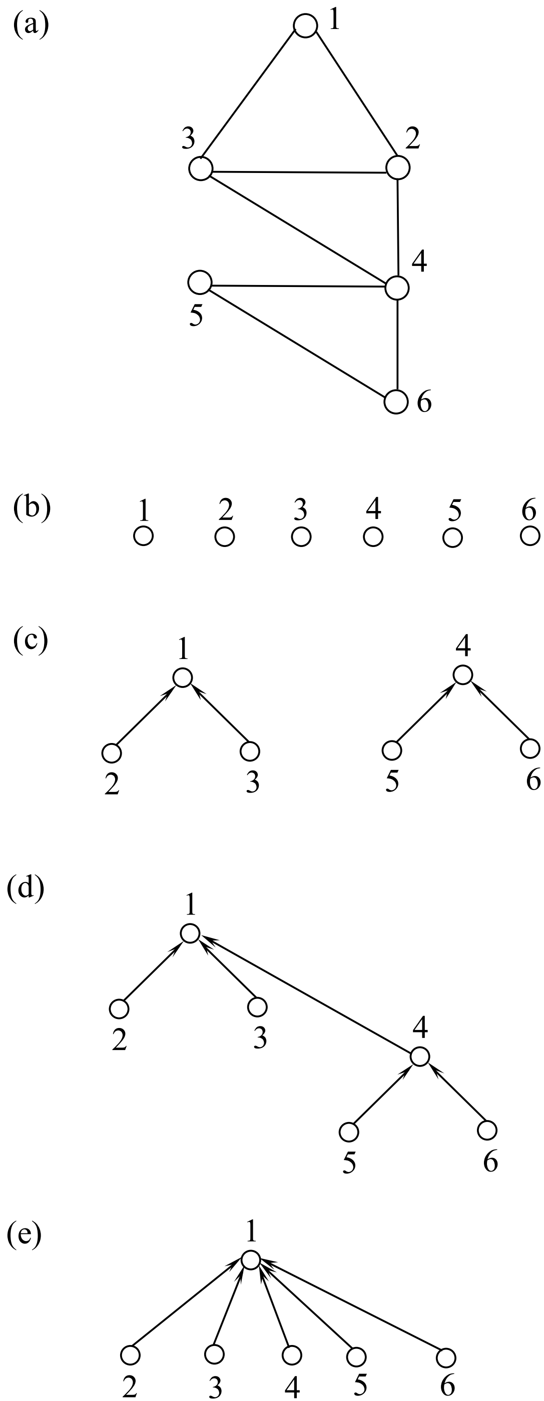


Figure 2.4 Grafting and pointer jumping operations applied to sample graphs.

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 [34].

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 [34]). A characterization for identifying redundant edges with respect to biconnectivity of a graph is presented by Cong and Bader [10]. A necessary and sufficient condition for a graph to have an ear decomposition is that the graph to 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. Indeed, as we show in the following lemma, biconnectivity and ear decomposition share the same notion of redundant edges.

Lemma 2.2.1 *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 [10], 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, $(P_0, P_1, \cdots, P_s, Q_{s+1}, Q_{s+2}, \cdots, Q_{s+r})$ is an ear decomposition of G where Q_{s+i} is the edge e_i in the graph $G \setminus (T \cup F)$ with $r = |E(G \setminus (T \cup F))|$.

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 $1 \leq i \leq r$, is a valid ear. The endpoints of Q_{s+i}, for $1 \leq i \leq r$, belong to the nodes in $\cup_{j=1}^{s} P_j$. Finally, it can be noticed that $E(G) = (\cup_{i=0}^{s} P_i) \cup \left(\cup_{j=s+1}^{r} Q_j \right)$. Therefore, $(P_0, P_1, \cdots, P_s, Q_{s+1}, Q_{s+2}, \cdots, Q_{s+r})$ is an ear decomposition of G.

An example is illustrated in Figure 2.5. 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.

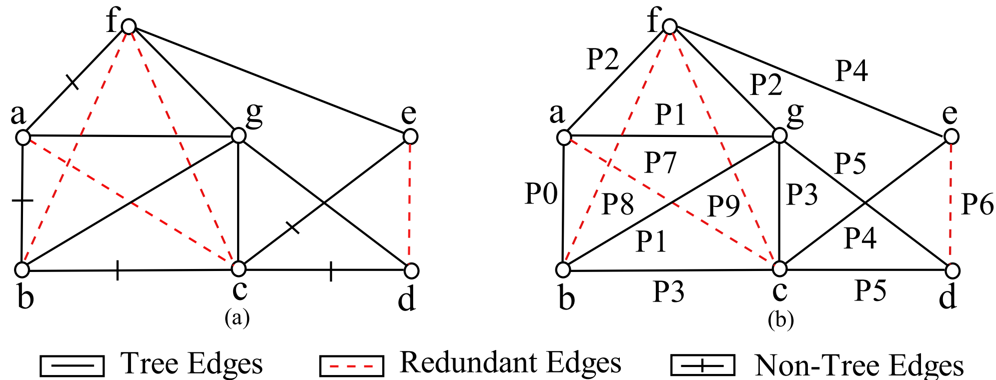


Figure 2.5 An example of using Lemma 2.2.1. In the graph on the left, we note that edges shown in dashed lines and red color are redundant. An ear decomposition, as ears P_0 through P_5, of the graph with the rest of the edges is shown in the right part of the figure. As the lemma shows, ears P_6 through P_9 correspond to trivial ears of the redundant edges.

2.2.2 Our Algorithm

The results of the earlier section indicate that obtaining an ear decomposition for a given input graph G can be achieved by obtaining an ear decomposition of a sparse 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 [34] with some changes, followed by a post-processing step. Our algorithm is presented as Algorithm 1 with a brief description in the subsequent paragraphs.

Algorithm 1 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**

2.2.2.1 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 [29]. As mentioned in Steps 2–4 of Algorithm 1, 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 2.2.1, we remove all edges in $G \setminus (T \cup F)$ from further consideration. The remaining graph, G', has n nodes and at most $2n - 2$ edges.

2.2.2.2 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 2.2).

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. 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 2.2 (in Section 2.3.2).

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, ℓ 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 group 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 [16].

2.2.2.3 Phase III

In this phase, edges that were pruned in Phase I will be included in the ear decomposition of G as trivial ears.

2.3 Experimental Results and Analysis

2.3.1 Platform Details

In this section, we introduce briefly our computing platform that consists of NVidia's 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 and supports PCI Express Gen3. Base clock frequency is 745 MHz and boost clock frequency is 875 MHz. Memory clock frequency is 3 GHz with bandwidth of 288 GB/sec. It has 12 GB on board memory and 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. It's peak double-precision floating point performance is 1.43 Tflops and single-precision floating point

performance is 4.29 Tflops. For more details of the Tesla K40c GPU, we refer the reader to [30]. For programming the K40c GPU, we use CUDA Version 7.5 as described in [31].

Intel(R) Xeon(R) E5-2650 CPU has 128 GB RAM and a memory bandwidth of 68 GB/s. It 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. It has 64 KB L1 cache per core, 256 KB L2 cache per core and a shared 25 MB L3 cache. It is using PCI Express 3.0 with lane combinations x4, x8 and x16 that is used to link processors PCI lanes and PCI devices.

2.3.2 Real World and Synthetic Dataset

We use the graphs listed in Table 2.2 for our experiments. The graphs include both real-world graphs from [1] and also Erdos-Reyni random graphs [6] generated using the RMAT generator [12]. Since we require the graph to be biconnected to have an ear decomposition, we made the graphs in Table 2.2 biconnected by adding additional edges. We also remove self-loops and make all the graph undirected.

2.3.2.1 Results

In Figure 2.6(a) we study the performance of our algorithm implemented with respect to that of [34]. Both algorithms are implemented on GPU. Figure 2.7(a) shows comparison of our algorithm implemented on CPU with Vijaya's [34] CPU implementation. For a better understanding of the performance of our algorithm, we also add the pruning step from Algorithm 1 to the algorithm of Ramachandran [34]. This modification is labeled as " [34] with Pruning" in the plot in Figure 2.6(a) and 2.7(a). We have also provided absolute timings for our, vijaya's and vijaya with pruning algorithm on GPU and CPU in Table2.3 and 2.4 respectively.

Figure 2.6(a) and 2.7(a) show the absolute runtime as well as the speedup of our algorithm with respect to that of [34]. 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 and 2.02x better than the algorithm of [34] on GPU and CPU respectively. If we add the pruning step to the algorithm of [34], our algorithm outperforms this variation by a factor of 1.54 on GPU and 1.74 on CPU. This indicates that our performance gains are due to both the pruning step and other algorithmic enhancements to that of [34].

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 2.6(b) where we keep the number of nodes fixed at 1 M and 2 M nodes and increase the number of edges.

Comparison between our algorithm and PRAM algorithm is shown in Tabel 2.5. We have used similar dataset used by Bader et al. which has two types of graph regular mesh and random planer graphs. Lattice and Triangulation graphs are regular mesh. Remaining graphs are random planer graphs. An example of regular lattice and regular triangulation is shown in Figure 2.8. Graph generator LEDA

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 $\mathcal{G}(n, p)$					
$\mathcal{G}(1M, 10 \times 10^{-6})$	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 2.2 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 number in column labeled "Edges Pruned" shows the number of edges that are deemed redundant according to Lemma 2.2.1. The number in the column labeled "Avg. Dist." refers to the average number of tree edges traversed to find the LCA of the end points of a nontree edge. The number in the last column labeled "ACE" indicates the average number of fundamental cycles according to a BFS tree that pass through a tree edge.

[23] is used to generate regular mesh and random planer graphs. The generator first generates maximal planer graph and then delete all but m i.e. given number of edges. We have used $|V| = 8192$ number of vertices as used in [4] and 8 threads for processing. As shown in 2.5, our algorithm is performing better than PRAM except in case of Regular Lattice. It is because average number of tree edges traversed to find LCA is very high.

Graph name	OUR	Baseline + Pruning	Baseline
roadNet-CA	87.62	102.12	102.33
roadNet-TX	40.06	33.42	33.64
soc-Epinions1	2.92	6.46	9.34
patents_main	72.97	72.89	74.08
coAuthorsDBLP	12.01	18.77	25.46
soc-Slashdot0902	5.74	9.61	11.84
caidaRouterLevel	9.38	15.60	19.26
scircuit	5.47	6.44	8.35
soc-sign-epinions	4.24	9.85	14.7
p2p-Gnutella31	2.61	3.24	4.9
G(1M-10M)	72.57	88.05	175.08
G(1M-20M)	66.34	108.48	211.05
G(1M-40M)	39.08	105.85	213.25
G(1M-80M)	53.21	160.28	339.8
G(2M-10M)	82.43	94.85	174.99
G(2M-20M)	94.53	124.06	253.37
G(2M-40M)	98.40	138.48	288.13
G(2M-80M)	88.06	168.07	333.2

Table 2.3 This table shows absolute time (in msec) for ear decomposition on GPU. Fist column shows time for OUR approach followed by time for [28] with pruning which is followed only [28].

Graph name	OUR	Baseline + Pruning	Baseline
roadNet-CA	683.04	594.12	1040
roadNet-TX	459.4	370.02	381.53
soc-Epinions1	68.45	159.58	174.21
patents_main	291.82	392.84	424.32
coAuthorsDBLP	115.46	272.66	310.89
soc-Slashdot0902	74.46	157.15	168.29
caidaRouterLevel	110.88	267.13	294.7
scircuit	85.98	115.61	142.79
soc-sign-epinions	290.68	584.88	683.20
p2p-Gnutella31	44.04	77.63	97
G(1M-10M)	432.66	846.54	928.68
G(1M-20M)	440.083	1602.75	1751.70
G(1M-40M)	482.57	3070.35	3099.42
G(1M-80M)	550.19	6306.90	6412.11
G(2M-10M)	834.05	949.44	1059.96
G(2M-20M)	888.05	1794.28	1974.80
G(2M-40M)	912.10	3199.46	3322.70
G(2M-80M)	984.10	6642.30	6821.2

Table 2.4 This table shows absolute time (in msec) for ear decomposition on CPU. Fist column shows time for OUR approach followed by time for [28] with pruning which is followed only [28].

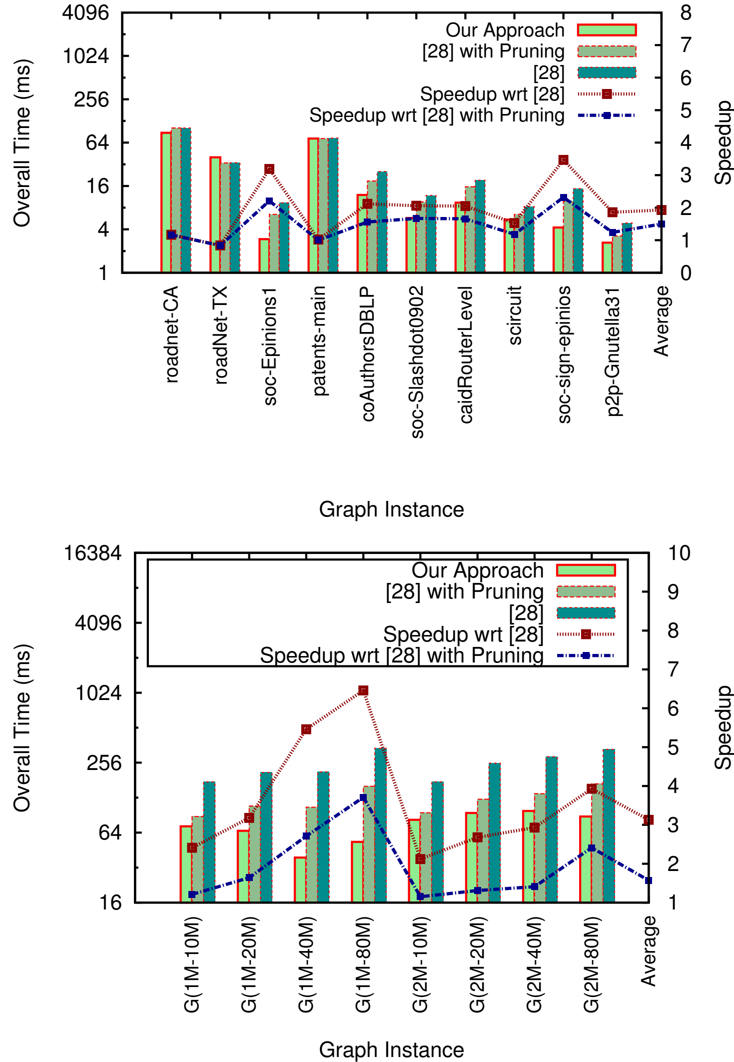


Figure 2.6 GPU Performance of our ear decomposition algorithm on real-world graph. (a) and on random graphs (b). The last label "Average" indicates the average speedup on the dataset from Table 2.2.

Graph name	V	E	OUR	PRAM	Avg Dist.
Random Graph A	5965	7686	0.791	1.9	4.21
Random Graph B	8192	16375	1.030	1.9	3.67
Random Graph C	8192	24570	1.345	1.9	3.39
Random Graph D	8192	24570	1.349	1.9	3.38
Regular Triangulation	8192	24551	1.422	1.9	4.12
Regular Lattice	8192	16199	15.0128	1.9	90

Table 2.5 This table shows absolute time (in msec) for ear decomposition [4] dataset. Fist column shows time for OUR approach followed by time for PRAM algorithm. Last column indicates average number of tree edges traversed to find the LCA of the end points of a nontree edge

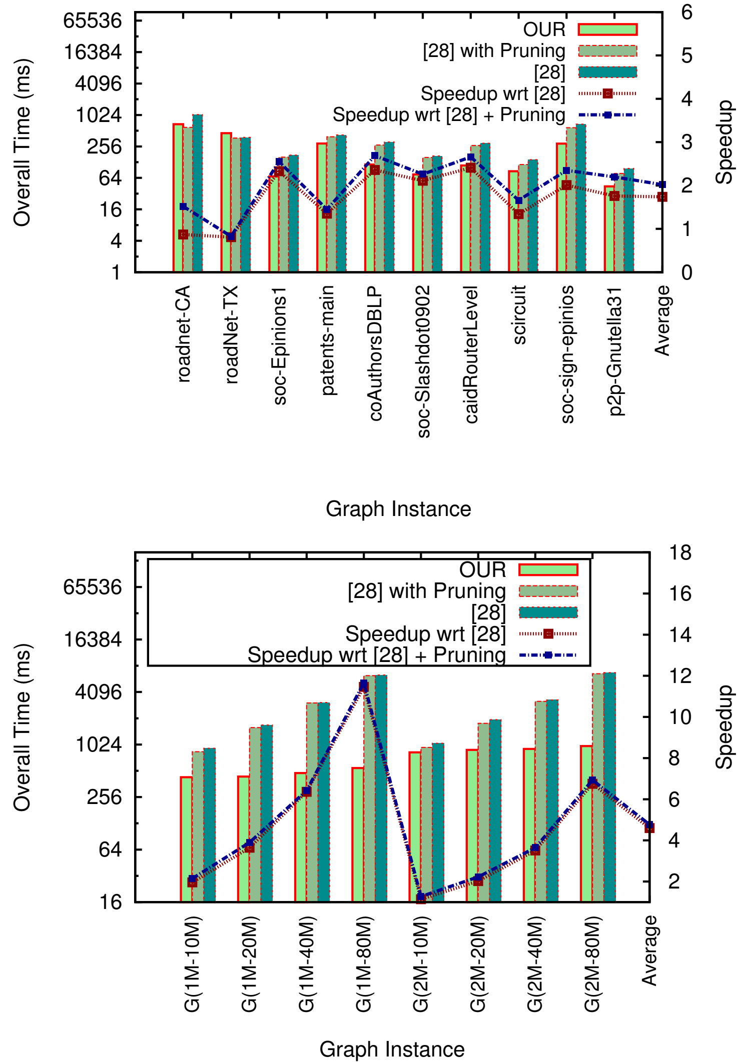
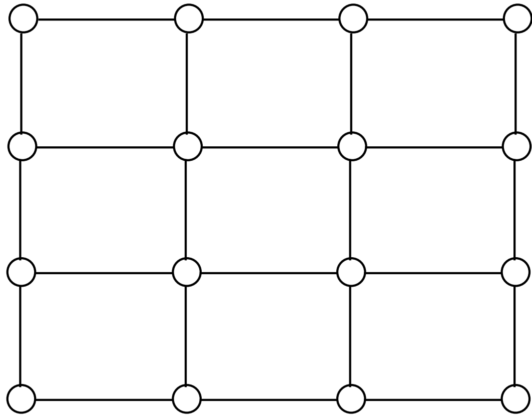


Figure 2.7 CPU Performance of our ear decomposition algorithm on real-world graph (a) and on random graphs (b). The last label "Average" indicates the average speedup on the dataset from Table 2.2.



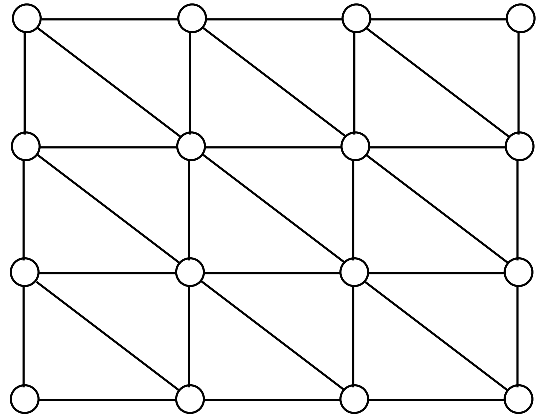


Figure 2.8 Part (a) shows regular lattice RL and part (b) shows regular triangulation RT graph.

Chapter 3

Betweenness Centrality

Betweenness centrality shows importance of node in a network. It is same as number of shortest path from all vertices to all other vertices pass through that vertex. A vertex at critical position in a network has significant impact on a network assuming that transfer of any entity in network follows shortest path. Earlier, mathematician J. Anthonisse proposed this idea but his work was never published. This concept has many applications in fields such as social network, communication, transport and biology.

The rest of the chapter is organised as follows. In Section 3.1 we first discussed some related work on betweenness centrality computation. Section 3.2 describes our contribution followed by our algorithm and implementation details. Chapter ends with section 3.3, which shows our experiment results and analysis.

3.1 Related Work

Computing the betweenness-centrality values of nodes in a graph has seen lot of interest in recent years in parallel computing research. Most papers [3,27,28,37] use the algorithmic approach of Brandes [7] due to improved time complexity from $O(n^3)$ to $O(nm)$ for unweighted graphs. Sariyuce et al. [37] use a biconnected component decomposition of a graph, computing the betweenness-centrality of a node local to its biconnected component, followed by a post-processing step of 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 [7]. Wang et al. [42] provided optimization similar to [37] and achieve considerable speedup on existing multicore implementations including that of the Ligra framework [25].

Optimizing BFS on a GPU with applications to betweenness-centrality in unweighted graphs is studied by Sariyuce et al. [3], and by Bader and Mc. Laughlin [27, 28, 43]. Bader and Mc. Laughlin [27] improved the BFS implementation of Jia et al. [17] and Shi and Zhang [38].

3.1.1 Bader and McLaughlin algorithm (BM15)

The main idea of the works of Bader and Mc. Laughlin [27, 28] 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 [27] batch the n BFS operations on the SMXs.

This algorithm takes advantage of level parallelism. It differs from previous GPU implementations in two ways. First difference is storage of current and next level vertices. It uses two queue's namely $curr_q$ and $next_q$. $curr_q$ stores vertices at current level and next queue stores vertices at next level. A thread is assigned to vertex in current level which explores child nodes of that vertex. After computing S_x, D_x and σ_x arrays (as explained in 1.2.1)for vertices in current level $next_q$ is copied into $curr_q$. This procedure is repeated till the last level. Second is the memory requirement for the predecessor array. This algorithm is not storing the predecessor array. Predecessor nodes are computed during the accumulation phase by using level difference and adjacency array which reduces the memory requirement to $O(n)$. It is computed on the fly by neighbour traversal method. In an accumulation phase of the algorithm predecessor is found by comparing level difference between adjacent elements of a vertex. Level difference of one between current level and previous level of a node indicates parent child relation. This step adds additional computation but overall complexity does not change.

Bader and Mc. Laughlin [28] introduced 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 [27] for computing the betweenness centrality values of nodes in unweighted graphs.

3.2 Contribution

3.2.1 Our Algorithm

Our algorithmic approach to compute betweenness-centrality of nodes in a sparse graph uses the following outline. We first start with considering graphs that are biconnected. Such a graph will have an ear decomposition as shown by [34, 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 3.1. In the following, we present a pseudocode of our algorithm as Algorithm 2 and provide details of the steps in our algorithm in Sections 3.2.1.1–3.2.1.3.

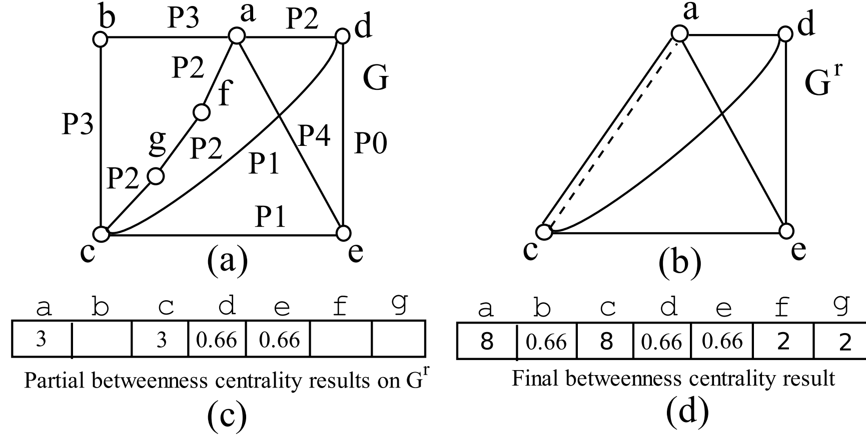


Figure 3.1 Figure (a) shows the input graph G with an ear decomposition where the number on the edges indicates the ear number they belong to. Figure (b) shows the reduced graph G^r. A parallel edge in the reduced graph $(a - c)$ is shown as the dotted line for illustration purposes. Figure (c) shows 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.

Algorithm 2 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: $(S_v, D_v, P_v\ \sigma_v)$ = FWDSTAGE(v, G)
6: ACCUMULATE_PARTIALBC$(S_v, D_v, P_v, \sigma_v)$
7: **end for**
8: /* Phase III : Post Processing */
9: **for** each $v \in G \setminus G^r$ do in parallel **do**
10: $l_x \leftarrow \text{left}(v)$, $r_x \leftarrow \text{right}(v)$
11: $(S_v, D_v, P_v\ \sigma_v)$ = SIM_FWDSTAGE(v, l_x, r_x)
12: SIM_ACCUMULATION$(v, l_x, r_x, S_v, D_v, P_v, \sigma_v)$
13: Update the BC values to the nodes in G^r
14: **end for**

3.2.1.1 Preprocessing

Let $G = (V, E)$ be an input graph that is biconnected. The REDUCE(G) routine starts by obtaining an ear decomposition of G using Algorithm 1. 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 3.1(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 $\mathrm{left}(x_1)$ and $\mathrm{right}(x_1)$, 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 3.1(b), $\mathrm{left}(f) = a$ and $\mathrm{right}(f) = c$.

Notice that during the construction of the reduced graph, there could be multiple edges between nodes in the reduced graph. Figure 3.1(b) shows how multiple edges can appear. In this case, since we are interested in shortest paths, we retain the edge with the shortest length and discard the remaining edges. These edges are shown in Figure 3.1(b) only for purposes of illustration and are not considered in the actual processing.

3.2.1.2 Processing

In the processing phase, we compute the betweenness-centrality values of nodes in G^r. We use the BFS routine from [28] 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, P_v and σ_v are recorded as the result of a BFS with v as the source node as is done in the forward propagation phase. In the ACCUMULATE_PARTIALBC routine, we use the arrays S_v, D_v, P_v, and σ_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 *partial* betweenness-centrality values as shown in Figure 3.1(c).

3.2.1.3 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 computed in the processing phase. The routine SIM_FWDSTAGE works as follows. Let x be a node in $G \setminus G^r$ with $\mathrm{left}(x) = \ell_x$ and $\mathrm{right}(x) = r_x$. We simulate the actions of executing the forward stage of Brandes algorithm [7] for node x as follows. We need to obtain arrays S_x, P_x, D_x and σ_x as part of the forward stage. For obtaining the array S_x, we start by merging sequences S_{ℓ_x} and S_{r_x} as follows. Let v and w denote the first node in S_{ℓ_x} and S_{r_x} respectively. We now compare $D_{\ell_x}(x) + D_{\ell_x}(v)$ and $D_{r_x}(x) + D_{r_x}(w)$. $D_{l_x}(x)$ and $D_{r_x}(x)$ represent distance of x from v and w respectively. $D_{l_x}(v)$ and $D_{r_x}(w)$ indicate distance of v and w from x

respectively. 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_{ℓ_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 [11]). In a similar fasion, we can also obtain arrays D_x, P_x, and σ_x from the respective arrays of nodes ℓ_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 1.2.1. 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 3.1(d).

3.2.2 Implementation Details

In this section, we mention some of the important implementation details of our algorithm. The preprocessing step in our algorithm necessitates a post-processing step unlike other algorithms [27, 28, 43]. 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 3.2.2.1, we identify information computed in the processing phase that is not needed in the post-processing phase which reduces bookkeeping. Secondly, in Section 3.2.2.2, 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.

3.2.2.1 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, P, D, and σ 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, P_v, and σ_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. Figure 3.2(b) illustrates the idea of free and active nodes. In second column of Table 3.2.2.2, we have provided count of active nodes in largest BCC for each graph in our dataset.

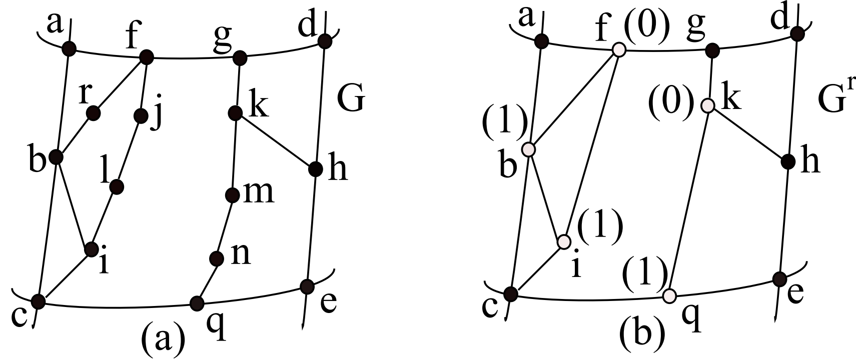


Figure 3.2 An illustration of our approach. Figure (a) shows the input graph G. Figure (b) shows the reduced graph G^r. In Figure (b), nodes filled in black color indicate *free* nodes and other the other nodes are *active* nodes. The numbers on the active nodes in Figure (b) indicate the BFS level number with f and k as the source nodes. It can be noticed that the active nodes consist of two connected components: one containing nodes f, b, i and the other containing nodes k, q.

3.2.2.2 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, P_v, and σ_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 [40]. 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 Dep$(H) := \{x | x \in G \setminus G^r, \text{left}(x) \in V(H) \text{ or } \text{right}(x) \in V(H)\}$. Once nodes in 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. This is shown in Table 3.2.2.2. Third column of Table 3.2.2.2 shows vertices in largest connected component. It shows significant memory reduction as compare to storage of all active vertices, on average 76%.

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

For a node $x \in$ Dep(S), the nodes $v := \text{left}(x)$ and $w := \text{right}(x)$ 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 Dep(S_1) as

defined. Once the post-processing of nodes in Dep(S_1) finishes, we define $S_2 = L_1 \cup L_2$ and perform the post-processing of nodes in 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 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 $O(\max_i |L_{i-1} \cup L_i| \cdot n)$. Third column of Table 3.2.2.2 shows maximum vertices in a level for largest connected component. This technique reduces storage requirement by 82% on average.

The two techniques reduce our space requirement significantly. In most real-world graphs, the technique illustrated in Sections 3.2.2.1, 3.2.2.2 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_{ℓ_x} and S_{r_x} where $\ell_x = \text{left}(x)$ and $r_x = \text{right}(x)$. To do so, we use one SMX for each node x. Threads within an SMX compute the array S_x as explained in Section 3.2.1.3. A similar approach is followed for computing the arrays P_x, D_x, and σ_x.

Graphs	Active nodes in Largest BCC	Nodes in largest connected component	Max. nodes per level
RoadNet-CA	424921	41	6
RoadNet-TX	276461	57	6
Soc-Epinions1	8918	5521	62
patents_main	43937	3841	61
coAuthorsDBLP	36686	4744	99
soc-Slashdot0902	11100	6409	70
caidRouterLevel	32783	183	81
scircuit	15912	58	55
soc-sign-epinios	12617	8861	95
p2p_Gnutella31	10461	5461	26

Table 3.1 It shows reduction in memory requirement for different stages of our algorithm.

3.2.3 Our Algorithm Extended 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. [37] and Wang et al. [42] that we use in our solution.

3.2.3.1 The BADIOS and the APGRE Framework

The main idea of the BADIOS framework [37], called as APGRE framework in [42], is to decompose a graph G into its biconnected components (BCCs) and use Brandes algorithm [7] on the individual biconnected components.

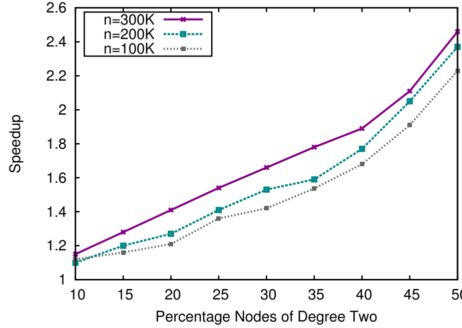


Figure 3.3 The relative performance obtained by Algorithm 2 over [28] on synthetic graphs.

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. A reachability metric is defined for alias nodes as follows. Consider the ith 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.) After the decomposition step, Sariyuce et al. [37] use Brandes algorithm to compute betweenness centrality values within the BCC's. The reachability metric is useful in extending the betweenness centrality values computed on the BCCs of G to the graph G. Reach values for every alias node of articulation point are computed by a leaf-to-root a traversal of the block tree T. Suppose we compute the number of nodes n_i in each the i'th BCC G_i of G. Let G_i and G_j be two BCCs of G such that G_i is a leaf node in T and G_j is the parent of G_i in T with G_i and G_j sharing an articulation point v with alias nodes v" and v' in G_i and G_j respectively as shown in Figure 3.4. We set reach(v"), as the difference between the total number of vertices in the graph G and the number of nodes in $BCC(i)$. In other words, reach(v") =$|V(G)| - n_i - 1$. Further reach(v') is set as the number of nodes in sub tree rooted at G_j. This procedure is extended via a leaf-to-root traversal of T as described by Puzis et al. [33].

3.2.3.2 Our Algorithm for General Graphs

To use the framework of BADIOS [37] or APGRE [42], 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 2 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 [37, 42], 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

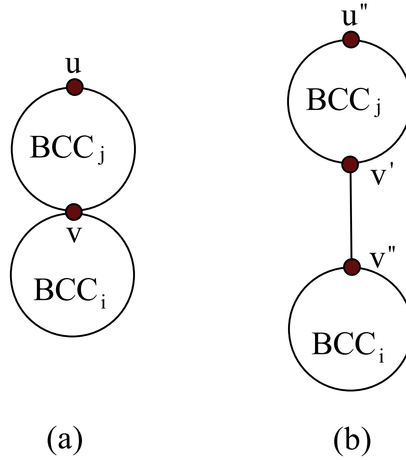


Figure 3.4 An example of reach value calculation.

2. Finally, we have two post-processing steps: first that is similar to the post-processing phase of Algorithm 2, and the second one similar to that of the corresponding step in [37, 42].

3.3 Experimental Results

3.3.1 Platform Details

We have used same platform as described in section 2.3.1.

3.3.2 Dataset

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 3.2. Since graphs in the dataset from Table 3.2 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.

3.3.2.1 Results

We reconsider the graphs listed in Table 3.2 and apply our approach to compute betweenness-centrality. We use the experimental platform mentioned in Section 2.3.1. For performance comparison, we consider the algorithms listed in Section 3.3.2.1. The results are shown in Figure 3.5.

Figure 3.5(a) shows the time taken by our algorithm and the other algorithms on the graphs listed in Table 3.2. The numbers in Figure 3.5(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

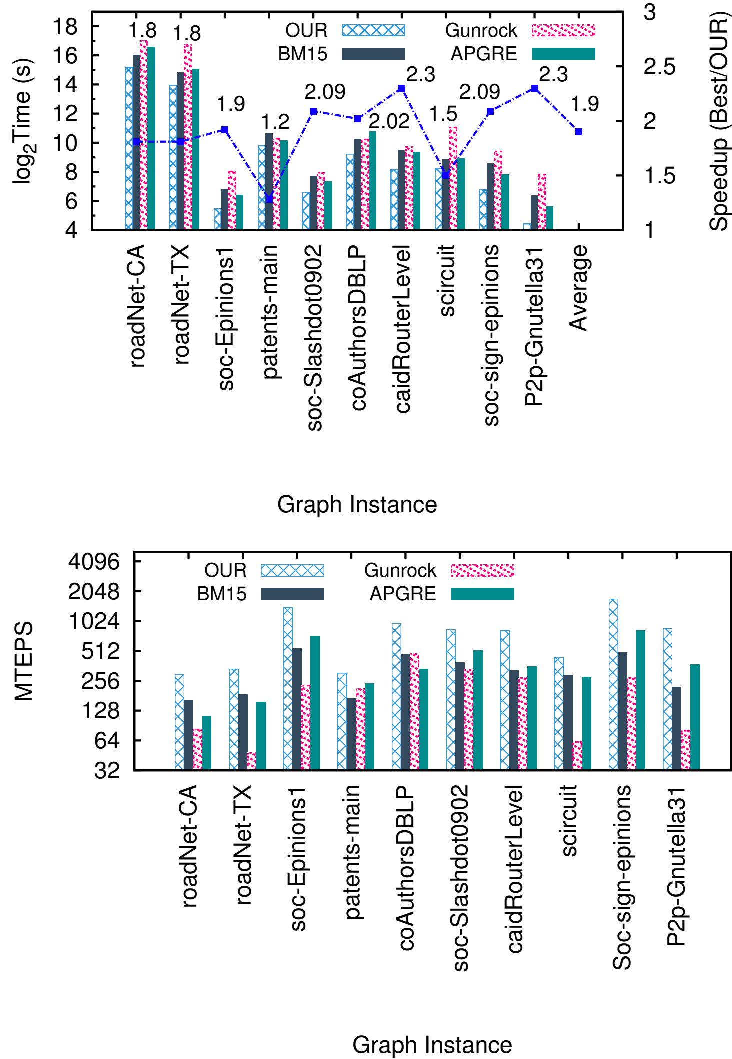


Figure 3.5 Comparing the overall performance improvement of Algorithm 2 with respect to that of [28], [43], [42], [25]. The plot on the left (*resp.* right) shows the absolute time (MTEPS) achieved by our algorithm, BM15 [28], Gunrock [43], and APGRE [42], in that order, respectively. The last instance 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.

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 3.2 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 near thousand (K) or the nearest million (M). The Last column indicates the percentage of nodes that are eliminated from the largest BCC during our reduction step.

3.3 in the columns labeled ”Complete Graph”. The speedup achieved in the case of the complete graph is higher than the speedup achieved on the largest biconnected component of the corresponding graph as can be seen from Figures 3.5(a) and 3.8(a). The reason for this is that when using the algorithms from [28, 43], every BFS has to run on the entire graph. In our algorithm, and also that of [42], each BFS runs only local to a biconnected component. The throughput of our algorithm as MTEPS is shown in Figure 3.5(b) along with the throughput achieved by the other algorithms. Table 3.4 shows absolute time for complete graph and Table 3.5 shows absolute time for betweenness centrality computation in largest BCC.

3.3.2.1.1 Bader and Mc. Laughlin [28] This work is currently one of the fastest for computing betweenness-centrality on a GPU. We use the software from the authors of [28] in our comparison. This result is labeled ”BM15” in the rest of the section.

3.3.2.1.2 Gunrock Library [43] Gunrock is a GPU based library for graph algorithms. It has flexible API's that can express wide range of graph processing primitives at a high level of abstraction. It includes several GPU specific optimizations such as load balancing and memory efficiency, that help to achieve high performance. Instead of focusing on $computation$ steps of an algorithm Gunrock tries to optimize $frontier$ data structure of edges or vertices that represents subset of graph which is currently participating in computation. To manipulate $frontier$ Gunrock has defined three steps namely advance, filter and compute. An advance step generate new frontier from current frontier by visiting neighbouring vertices. Filter is used to select subset of vertices from newly generated frontier vertices according to user define function. Finally, compute step perform user specified operation on all elements (edges or vertices) in current frontier. The latest release of which contains a routine for computing betweenness-centrality. We use the software from [43] and label this result as ”Gunrock” in the rest of this section.

3.3.2.1.3 APGRE [42] and Ligra [25] Wang et al. [42] extends the work of Sariyuce et al. [37] to multi-core CPUs. We implement the algorithm of [42] on the CPU described in Section 3.3.1 with a thread per core and label this result as "APGRE".

The Ligra library [25] consists of routines for graph algorithms and runs on multicore CPUs. It is a light weight graph processing framework for shared memory parallel machines. It has two simple routines called as $Edgemap$ and $Vertexmap$ for mapping over edges and vertices. For betweenness centrality computation, it is maintaining frontier array. It holds unprocessed vertices in next level. Frontier array is computed by traversing neighbours of vertices in previous frontier array in parallel. If adjacent vertex is not visited then it is added to next frontier array. An atomic operation compare and swap CAS is used to mark visited vertex because multiple threads can mark same vertex in parallel. For dependency calculation phase, it is reversing the graph, since edges now needs to point in reverse direction. Then, using partial dependency relation it is updating betweenness centrality of every vertex.

On average our algorithm is outperforming LIGRA by factor of 23.91x on full graph as shown in Figure 3.7 and 17.25x on largest BCC as shown in Figure 3.6. To compare performance of both the approaches we have provided absolute time and average speed-up on each graph.

On the graphs from Table 3.2, the overall time taken by the above algorithms on the largest biconnected component is plotted in Figure 3.8(a)[1]. The Y-axis of Figure 3.8(a) is on a logarithmic scale. The secondary Y2 axis of Figure 3.8(a) shows the speedup of our algorithm over the **best** of above mentioned baseline algorithms.

The relative speed-up obtained by our algorithm with respect to individual algorithms is shown in Table 3.3. 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 [28], labeled as "BM15", the Gunrock library [43], labeled as "GUNROCK", and the algorithm of Wang et al. [42], labeled as "APGRE". As can be seen from Table 3.3, the average speedup achieved is 1.6x, 2.08x and 1.96x with respect to [28], [43], and [42] respectively.

The throughput achieved by the algorithms under study is shown in Figure 3.8(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 on the graph G. The quantity MTEPS refers to Million TEPS. As can be seen from Figure 3.8(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 2 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

[1]Note that in [28], absolute time taken as mentioned are normalized to 8192 iterations. Similarly, the timings shown in [43] are normalized to one iteration.

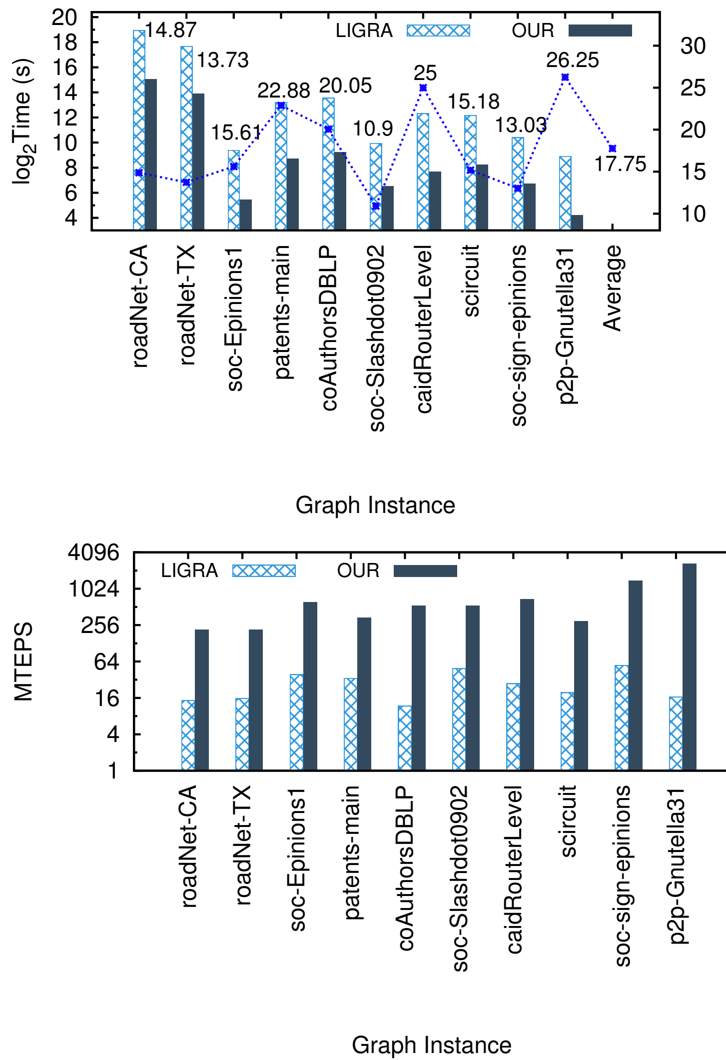


Figure 3.6 Comparing performance of Algorithm 2 with LIGRA on largest BCC. Part (a) shows absolute time of our algorithm with respect to LIGRA. Part (b) shows MTEPS achieved by our algorithm with respect to LIGRA. The last instance on the X-axis of part (a) of the figure shows the average speed-up.

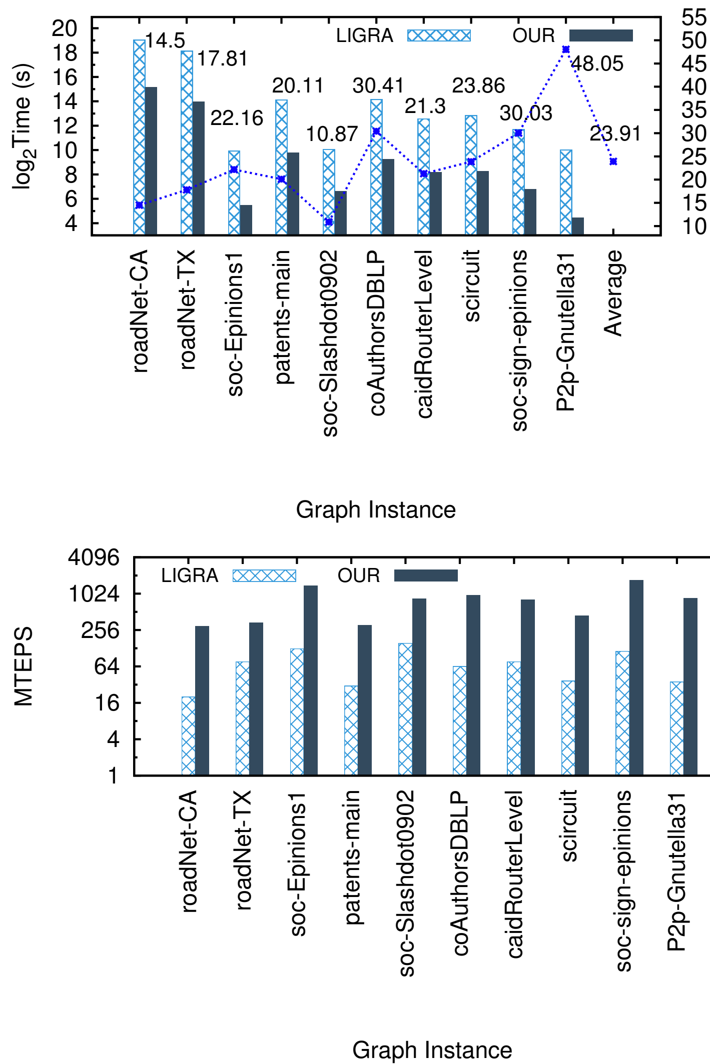


Figure 3.7 Comparing performance of Algorithm 2 with LIGRA on complete graph. Part (a) shows absolute time of our algorithm with respect to LIGRA. Part (b) shows MTEPS achieved by our algorithm with respect to LIGRA. The last instance on the X-axis of part (a) of the figure shows the average speedup.

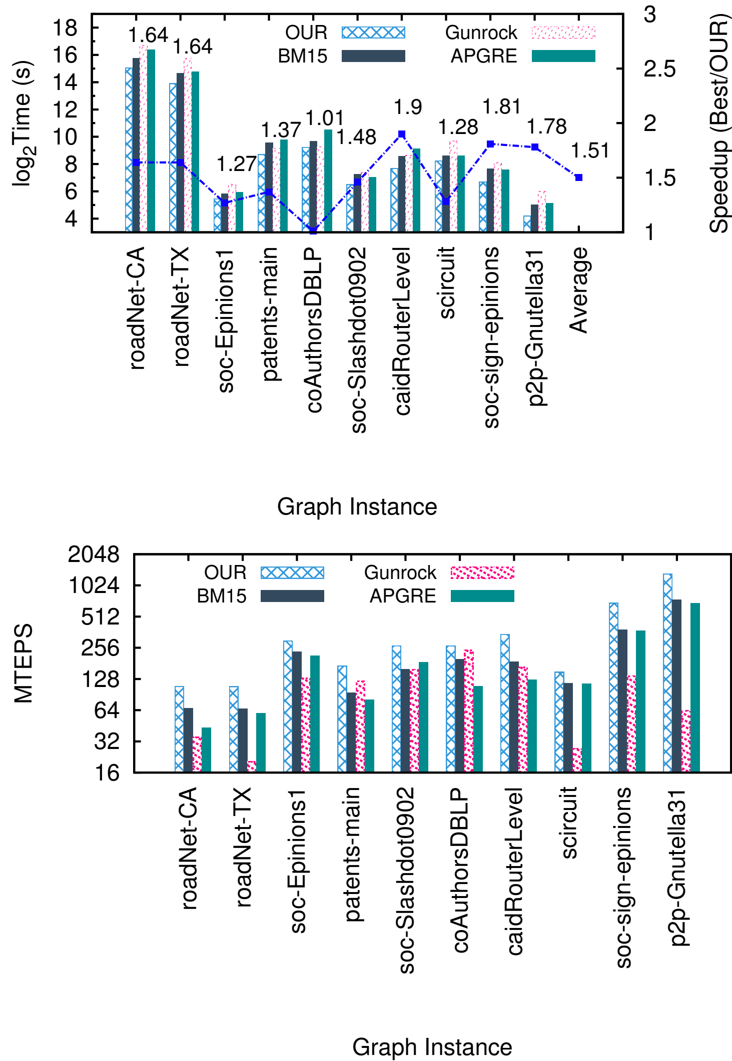


Figure 3.8 Comparing the overall performance improvement of Algorithm 2 with respect to that of [28], [43], [42], [25]. The plot on the left (*resp.* right) shows the absolute time (MTEPS) achieved by our algorithm, BM15 [28], Gunrock [43], and APGRE [42], in that order, respectively. The last instance 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.

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 2 with respect to that of Bader and Mc. Laughlin [28]. As shown in Figure 3.3, for a fixed n, as the percentage of nodes of degree two increases, the speedup achieved by our algorithm also increases.

	Speed up with respect to							
Graph name	**Largest BCC**				**Entire Graph**			
	BM15	**GUNROCK**	**APGRE**	**LIGRA**	**BM15**	**GUNROCK**	**APGRE**	**LIGRA**
roadNet-CA	1.62	3.06	14.50	1.61	1.81	3.56	2.62	14.87
roadNet-TX	1.64	5.26	1.80	17.81	1.81	6.94	2.15	13.74
soc-Epinions1	1.30	2.32	1.41	22.63	2.59	6.03	1.92	15.65
patents_main	1.81	1.38	2.12	36.78	1.80	1.43	1.28	22.88
coAuthorsDBLP	1.35	1.09	2.46	20.63	2.05	2.02	2.88	20.05
soc-Slashdot0902	1.69	1.70	1.45	10.87	2.15	2.56	1.64	10.90
caidaRouterLevel	1.83	2.07	2.73	21.30	2.53	2.98	2.30	25.00
scircuit	1.29	5.41	1.29	23.86	1.50	7.07	1.57	15.18
soc-sign-epinions	1.81	2.63	1.84	30.03	3.47	6.24	2.09	13.03
p2p-Gnutella31	1.78	3.43	1.92	48.05	3.89	10.56	2.30	26.25
Average	**1.60**	**2.08**	**1.96**	**24.4**	**2.06**	**3.44**	**2.04**	**16.96**

Table 3.3 This table shows the relative performance of OUR algorithm, labeled OUR, over [28] labeled ”BM15”, [43] labeled ”GUNROCK” and [42] labeled ”APGRE” on the largest BCC and the complete graph.

Graph name	**OUR**	**BM15**	**GUNROCK**	**APGRE**	**LIGRA**
roadNet-CA	33969.11	55030.21	104237.49	85462.84	538159.71
roadNet-TX	15303.04	25097.02	80520.23	27652.38	285643.52
soc-Epinions1	43.25	56.11	100.76	61.37	978.96
patents_main	416.50	755.54	578.91	885	17872.64
coAuthorsDBLP	602.81	814.99	661.65	1484.41	18332.81
soc-Slashdot0902	90.72	153.03	154.56	132.09	1068.18
caidaRouterLevel	207.03	381.29	428.60	566.5	6113.36
scircuit	304.11	391.01	1646.36	393	7335.77
soc-sign-epinions	103.65	187.14	273.14	190.93	3302.66
p2p-Gnutella31	18.42	32.87	63.34	35.49	1038.93

Table 3.4 This table shows absolute time of OUR algorithm, labeled OUR, BM15, GUNROCK, APGRE and LIGRA on the largest BCC.

Graph name	OUR	BM15	GUNROCK	APGRE	LIGRA
roadNet-CA	37112.31	67247.32	132243.98	97085.49	504965.92
roadNet-TX	16035.76	28965.98	111214.89	34447.92	210086.8
soc-Epinions1	44.17	114.36	266.56	85	675.27
patents_main	888.6	1602.76	1267.11	1133.81	9530.63
coAuthorsDBLP	615.81	1237.38	1215.76	1734.58	12083.81
soc-Slashdot0902	98.24	211.37	251.1	161.21	989.33
caidaRouterLevel	286.94	725.84	854.54	658.56	5175.54
scircuit	307.24	460.11	2173.52	480.86	4615.77
soc-sign-epinions	109.97	381.63	686.42	229.55	1350.72
p2p-Gnutella31	21.62	84.11	228.42	49.76	483.51

Table 3.5 This table shows absolute time of OUR algorithm, labeled OUR, BM15, GUNROCK, APGRE and LIGRA on the complete graph .

Chapter 4

Conclusions and Future Work

In this thesis, we have proposed an efficient approach for ear decomposition. We proved that our algorithm in general outperforms the corresponding best known implementations. Our pruning technique is useful to identify and remove redundant edges which accelerates the ear decomposition computation.

We have also showed that ear decomposition can be used to find the betweenness centrality of nodes in a graph. Ear decomposition provides a systematic way to remove degree two vertices. Betweenness centrality of these degree two nodes computed using information available for adjacent nodes having degree more than two.

Our results indicate that for problems such as betweenness-centrality, using an ear decomposition is effective and practical. We believe that our technique can have independent interest and can be applied to other graph problems.

Related Publications

- [Accepted] Charudatt Pachorkar, Meher Chaitanya, Kishore Kothapalli and Debajyoti Bera, ”Efficient Algorithm for Ear Decomposition and Application to Betweenness Centrality”, to HiPC 2016.

Bibliography

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

[2] A. B. KAHNG, J. LIENIG, I. L. M., AND HU, J. *VLSI Physical Design: From Graph Partitioning to Timing Closure*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[3] 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.

[4] 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.

[5] 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.

[6] BOLLOBAS, B. Random Graphs, Cambridge University Press, 2001.

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

[8] CHAITANYA, M., AND KOTHAPALLI, K. Efficient multicore algorithms for identifying biconnected components. *IJNC* 6:1(2016), pp: 87–106.

[9] 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.

[10] CONG, G., AND BADER, D. A. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps). In *Proc. IEEE IPDPS* (2005).

[11] CORMEN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. Introduction to algorithms, 2001.

[12] BADER, D. A., AND MADDURI, K. GTgraph: A synthetic graph generator suite. (2006).

[13] 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).

[14] 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).

[15] 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.

[16] JAJA, J. An introduction to parallel algorithms. Addison-Wesley, 2004.

[17] 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.

[18] 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.

[19] KANEVSKY, A., AND RAMACHANDRAN, V. Improved algorithms for graph four-connectivity. *Jour. Comput. Syst. Sci.*, V. 42(1991), 288–306.

[20] KARYPIS, G., AND KUMAR, V. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *JPDC*, V. 48(1998), 71–95.

[21] 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).

[22] LÄSZLÖ, L. A note on factor-critical graphs. *Studia Sci. Math. Hung. 7*, pp279-280.

[23] LEDA Graph Generator Tool. http://http://www.algorithmic-solutions.com.

[24] LILJEROS, F., EDLING, C., R., AMARAL, L., A., STANLEY, H., E., AND ABERG, Y. The Web of Human Sexual Contacts. *Nature* (2001), vol. 411, no. 6840, pp907-908.

[25] SHUN, J., AND BLELLOCH, G. E. Ligra: a lightweight graph processing framework for shared memory. *PPOPP* (2013).

[26] MAON, Y., SCHIEBER, B., AND VISHKIN, U. Parallel ear decomposition search (EDS) and st-numbering in graphs. In *Theoretical Comput. Sci.*, 1986, vol. 47, pp. 277–298.

[27] MCLAUGHLIN, A., AND BADER, D. A. Scalable and high performance betweenness centrality on the GPU. In *Proc. ACM SC*, 2014, pp. 572–583.

[28] MCLAUGHLIN, A., AND BADER, D. A. Fast execution of simultaneous breadth-first searches on sparse graphs. In *IEEE ICPADS*, 2015, pp. 9–18.

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

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

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

[32] Porta, S., Latora, V., Wang, F., Strano, E., Cardillo, A., Scellato, S., Iacoviello, V., and Messora, R. Street Centrality and Densities of Retail and Services in Bologna, Italy. In *Environment and Planning B: Planning and design* (2009), vol. 36, no. 3, pp. 450465.

[33] 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.

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

[35] Ramachandran, V., and Reif, J., H. An optimal parallel algorithm for graph planarity. *Proc. 30th Ann. IEEE Symp. on Foundations of Comp. Sci.*, (1989), pp. 282–287.

[36] 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.

[37] 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.

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

[39] Si. Si, D. Shin, I. S. D., and Parlett, B. N. Multi-scale spectral decomposition of massive graphs. In *NIPS* (2014), pp. 2798–2806.

[40] 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.

[41] Soman, J., and Narang, A. Fast Community Detection Algorithm with GPUs and Multicore Architectures. In *Proc. IEEE IPDPS* (2011), pp. 568579.

[42] 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.

[43] 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.

[44] WHITNEY, H. Non-separable and planar graphs. In *Trans. Amer. Math. Soc.* (1932), vol. 34, pp. 339–362.

[45] ZERBINO, D. R., AND BIRNEY, E. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Res.* (2008).