Scalable and Performant Graph Processing on GPU using Approximate Computing

A THESIS

submitted by

SOMESH SINGH

for the award of the degree

of

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2021

THESIS CERTIFICATE

This is to certify that the thesis titled **Scalable and Performant Graph Processing on GPU using Approximate Computing**, submitted by **Somesh Singh**, to the Indian Institute of Technology, Madras, for the award of the degree of **Doctor of Philosophy**, is a bona fide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Rupesh Nasre
Research Guide
Associate Professor
Dept. of CSE
IIT-Madras, 600 036

Place: Chennai

Date: 9th June 2021.

ACKNOWLEDGEMENTS

"Nothing of me is original. I am the combined effort of everybody I've ever known."

– Chuck Palahniuk

My Ph.D. journey at IIT Madras has been – in equal parts – inspiring, challenging and humbling. Several people have been instrumental in enabling me to complete this long journey.

First and foremost, I express my sincere gratitude towards my advisor Dr. Rupesh Nasre. I had joined Ph.D. completely unaware of the rigors of research, and I owe a lot of what I have learned during my Ph.D. to him. He gave me the freedom to explore the ideas of my interest and also allowed me to work on problems that did not pertain to my Ph.D. research. He was always available for both technical and non-technical discussions. He was extremely patient with me and was always very encouraging despite the slow progress and the numerous discouraging results. It was immensely gratifying to work with him. He will forever remain a source of inspiration for me.

I thank my doctoral committee members — Prof. V Krishna Nandivada, Prof. Madhu Mutyam, Prof. M Ramakrishna, and Prof. C Chandra Sekhar, for their valuable suggestions and comments on my research work. I am thankful to Prof. C Siva Ram Murthy, Dr. John Augustine, and Dr. Rajsekar Manokaran for the useful discussions and for sharing their thoughts on my work. Their suggestions were beneficial.

I had the privilege of collaborating with Dr. Felice Pantaleo, Dr. Riccardo De Maria, and Martin Schwinzerl from CERN, during Google Summer of Code 2017 and 2018. I am grateful to them for the inspiring and enriching experience, and the many learnings. The work happening at CERN has always fascinated me, and all credit to them, I could make a minuscule contribution to these efforts. I especially enjoyed working with Martin, one of the most humble, polite, and passionate people I have come across.

I am thankful to MHRD and ACM SIGPLAN PAC for funding my conference travel.

I had the good fortune of being a part of the *vibrant* PACE lab. I thank all my present and past lab mates — Abhilash Bhandari, T V Kalyan, Tripti Warrier, John Jose, Raghavendra K, Sudharsan J, Suyash Gupta, Pritam Majumder, Gnaneswara Rao, Praveen Alapati, Raghesh Aloor, Joe Augustine, Jyothi Krishna V S, Anchu R S, Rahul Shrivastava, Rakesh Patil, Shashidhar G, Dennis Varkey, Indu K, Arun T, Aman Sharma, Aman Nougrahiya, Jyothi Vedurada, Manas Thakur, Saurabh Kalikar, Surya, Sayantan Ray, Puneet Saraf, Anju M A, Shouvick Mondal, Diptanshu Kakwani, Rajendra Dangwal, Sai Deepak, Jash Khatri, Praseetha M and Ramya, for the wonderful moments and memories over the years. I would always fondly remember the lab talks, the several technical and non-technical discussions, and the fun-filled lab outings.

I thank Jyothi, my neighbor in the lab, for all the conversations about research and life in general. She kept me abreast with the research in the areas of programming languages and software engineering. Saurabh, Manas, Arun, and Aman(s) often accompanied me for the "frequent" coffee breaks. Saurabh and Shouvick were my companions during the late hours in the lab. Thanks, guys.

During my initial years at IIT Madras, I spent a lot of my time in the company of my Ph.D. batchmates — Revathy Narayanan, Prasanna Karthik, and Ujjal Kumar Dutta. My Ph.D.'s most cherished memories are from the times we spent together, especially while preparing for the comprehensive exams. Thank you for the wonderful times.

I am thankful to Tejas (Rupesh's M.Tech. student) for his work on approximate betweenness centrality computation. I thank my interns, Ronak Jayesh Shukla (NIT Nagpur) and Milind Srivastava (IIT Madras), for their works on image-segmentation. I enjoyed working with all of them, and it also helped improve my understanding of several aspects of graph processing, parallel computing, and approximate computing.

I thank Saurabh Kalikar, Shreyas Shetty, Prathamesh Deshpande, Amit Rawat, and Arun T, my teammates for course projects or programming contests. I learned a lot from working with all of them.

I am grateful to my Yoga teacher Mrs. Katyayini Reddy, for developing my interest

in Yoga. Yoga has now become a part of my daily routine. I also thank Manas for convincing me to join the Yoga classes despite my initial resistance.

I thank the administrative staff, especially Mr. Mani and Mrs. Sridevi, from the CS office, for ensuring that paperwork was never a hassle. I am grateful to the Deans, and the entire administration of IIT Madras for permitting me to stay on campus during the COVID-19-forced lockdown, for working on my thesis.

I would surely miss the beautiful campus of IIT Madras. It was great to share the campus with the blackbucks, the deers, and the (notorious) monkeys and observe them in their natural habitat from such close quarters.

Last, I am indebted to my parents and brother for their unwavering support. It would not have been possible for me to complete my Ph.D. without their cooperation.

ABSTRACT

KEYWORDS: Graph Analytics; Parallelization; Approximate Computing; Graphics Processing Unit.

A multitude of real-world problems is modeled as graphs where nodes represent entities of interest, and edges capture the relationships among them. In recent years, Graphics Processing Units (GPUs) have gained popularity as accelerators for compute-intensive data-parallel applications due to the massive data-parallelism they offer and their high memory bandwidth. Graph algorithms are inherently irregular, while GPUs are best suited for structured data-parallel computation. Scaling graph algorithms is challenging today due to the rapid growth of unstructured and semi-structured data. The prior art has targeted parallelizing popular graph algorithms on various kinds of architectures such as multi-core CPUs, many-core GPU, and distributed and heterogeneous systems. The primary technical challenge posed by graphs is due to inherent irregularity in the data-access, control-flow, and communication patterns. The recent past has witnessed the emergence of very effective techniques to represent graphs compactly, tame irregular computations, and efficiently map those to the underlying hardware. However, when the graph sizes are huge (e.g., billion-scale networks), or the underlying processing is expensive, it is not practically viable to compute the exact solution in time.

The focus of this thesis is on making graph processing more scalable. We look to achieve this by enhancing the performance of parallel graph algorithms on GPU by trading off computational accuracy, using approximate computing. We propose several i) algorithm- and architecture-independent, ii) algorithm-independent but architecture-specific, and iii) algorithm-specific but architecture-independent techniques for enhancing parallel graph processing efficiency using approximate computing. We present Graprox, a set of four algorithm- and architecture-independent approximate computing techniques that improve the performance of parallel graph analytics by exploiting the general structure of the graph kernel and the flow of information in graph algorithms.

Graph analytics on GPU suffers from low coalesced accesses, high memory latency, and high workload imbalance. To alleviate these issues, we present Graffix, a framework with three novel graph transformation techniques that alter the graph structure to enable faster processing in exchange for small inaccuracies in the final results. Graffix's method for improving memory coalescing creates a graph isomorph that brings relevant nodes nearby in memory and adds a controlled replica of nodes. The second technique reduces memory latency by facilitating the processing of subgraphs inside shared memory by adding edges among specific nodes and processing well-connected subgraphs iteratively inside shared-memory. The third normalizes degrees across nodes assigned to a warp to reduce thread divergence.

Third, we focus on algorithm-specific but architecture-independent approximate computing techniques for improved parallel execution. Specifically, we propose ParTBC, an ensemble of techniques targeted at speeding up the computation of top-k betweenness centrality vertices on GPU. Our proposals restrict the computation of shortest paths from only a fraction of the vertices in parallel betweenness centrality computation, using online stopping criteria to terminate the computation faster.

All our proposals provide *tunable knobs* to change the degree of approximation injected and control the performance-accuracy trade-off in graph applications. Further, our approximate computing techniques complement (and do not compete with) the existing optimization techniques, and could be applied on top of these optimizations to enhance the execution performance further.

We illustrate the efficacy of our proposed techniques on graphs of varying characteristics and sizes and popular graph algorithms through extensive experimental evaluation. Overall, we show that approximate computation of graph algorithms is a robust way of dealing with irregularities. Approximate computing combined with parallelization promises to make heavy-weight graph computation practical, as well as, scalable.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS **i**

ABSTRACT **v**

LIST OF TABLES **xii**

LIST OF FIGURES **1**

1 INTRODUCTION **3**

1.1 Contributions 7

1.1.1 Architecture-agnostic Techniques for Parallel Approximate Graph Processing 7

1.1.2 GPU-specific Optimizations for Graph Processing in the Presence of Approximations 8

1.1.3 Estimation of Top-k Betweenness Centrality Vertices on Heterogeneous Architectures 8

1.2 Organization of the Thesis 9

2 Background **11**

2.1 Basic Definitions 11

2.2 Graph Storage Formats 12

2.3 Classification of Graphs 14

2.3.1 Scale-free Real World graphs 14

2.3.2 Recursive Matrix (R-MAT) graphs 14

2.3.3 Road networks 14

2.3.4 Random graphs 15

2.4 Parallel Graph Algorithms 15

2.4.1 Breadth First Search (BFS) 15

2.4.2 Single Source Shortest Path (SSSP) 16

2.4.3 PageRank 17

2.4.4 Minimum Spanning Tree (MST) 18
2.4.5 Strongly Connected Components 18
2.4.6 Vertex Coloring 19
2.4.7 Betweenness Centrality 21
2.5 Summary 22

3 Related Work 23
3.1 Exact Parallel Graph Processing 23
3.1.1 Parallel graph processing on multicore 23
3.1.2 Parallel graph processing on GPU 25
3.1.3 Parallel graph processing on distributed systems 27
3.2 Approximate Graph Processing 28
3.2.1 Graph sampling based techniques 29
3.2.2 Graph compression based techniques 31

4 Parallel Approximate Graph Processing 33
4.1 Approximation Model 35
4.1.1 Function Application Order 35
4.1.2 Idempotent Approximation 36
4.1.3 Approximation Structure 36
4.2 Approximating Graph Algorithms 36
4.2.1 Graph Algorithms 37
4.2.2 Technique 1: Reduced Execution 37
4.2.3 Technique 2: Partial Graph Processing 38
4.2.4 Technique 3: Approximate Graph Representation 40
4.2.5 Technique 4: Approximate Attribute Values 41
4.3 Benefits to GPU-based Processing 43
4.3.1 Technique 1: Reduced Execution 44
4.3.2 Technique 2: Processing Part of the Graph 44
4.3.3 Technique 3: Approximate Representation 44
4.3.4 Technique 4: Approximate Attributes 45
4.4 Experimental Evaluation 45

4.4.1 Overall Results 47
4.4.2 Effect of Reduced Execution 50
4.4.3 Effect of Partial Graph Processing 53
4.4.4 Approximate Graph Representation 56
4.4.5 Approximate Attribute Values 59
4.4.6 Effect on Graph Type 61
4.4.7 Graprox techniques are platform-independent 61
4.5 Practicality of Graprox techniques 62
4.6 Summary 62

5 GPU-specific Optimizations for Graph Processing in the presence of Approximations 65

5.1 Improving Memory Coalescing 66
5.1.1 Coalescing in Graffix 66
5.1.2 Renumbering Scheme 69
5.1.3 Node Replication 71
5.1.4 Confluence due to Replication 72
5.2 Reducing Memory Latency 73
5.3 Reducing Thread Divergence 75
5.4 Experimental Evaluation 77
5.4.1 Effect of Coalescing 78
5.4.2 Effect of Memory Latency 82
5.4.3 Effect of Thread Divergence 83
5.4.4 Preprocessing Overhead 84
5.4.5 Impact of Approximations on Energy 86
5.5 Summary 87

6 Faster Estimation of Top-k Betweenness Centrality Vertices on Heterogeneous Architectures 89

6.1 Problem Statement and Preliminaries 90
6.2 ParTBC's Approach 91
6.3 Parallelization and Graph Layout 92
6.3.1 Parallelization Strategy 92

6.3.2 Graph Layout 92
6.3.3 Improved Graph Layout 94
6.4 Techniques for Fast BC estimation 95
6.4.1 Random Selection of Source Vertices (Random) 98
6.4.2 Node Selection in Ascending Degree Order (Ascending) . . 99
6.4.3 Node Selection in Descending Degree Order (Descending) . 99
6.4.4 Selecting Low-Degree Neighbors of High Degree Vertices . 100
6.4.5 Dynamic Selection of Source 101
6.5 Experimental Evaluation 103
6.5.1 Overall Results 106
6.5.2 Effects of the fraction of source nodes 110
6.5.3 Controlling the number of outerloop iterations 112
6.5.4 Discussion on quality of the reported top-k 112
6.5.5 ParTBC techniques are platform-independent 114
6.6 Summary 115

7 Conclusion and Future Work 117

7.1 Conclusion 117
7.2 Limitations 118
7.3 Future Work 119

LIST OF TABLES

4.1 Instantiation of the approximation model with various values of $\mathcal{D}$ and $\mathcal{F}$ 34

4.2 Input graphs 46

4.3 Execution time for exact versions of graph algorithms 46

4.4 Overall results 48

4.5 Preprocessing overhead for partial graph processing 53

4.6 Preprocessing overhead for approximate graph representation 57

4.7 Preprocessing overhead for approximate attribute values 59

4.8 Effect of approximating attribute values. 60

4.9 Average execution time of the approximate versions of graph algorithms 63

5.1 Baseline-I: Execution time for the exact versions 78

5.2 Baseline-II: Execution time for Tigr 78

5.3 Baseline-III: Execution time for Gunrock 78

5.4 Effect of memory coalescing 79

5.5 Effect of shared memory 79

5.6 Effect of thread divergence 79

5.7 Effect of memory coalescing 79

5.8 Effect of shared memory 79

5.9 Effect of thread divergence 79

5.10 Effect of memory coalescing 80

5.11 Effect of shared memory 80

5.12 Effect of thread divergence 80

5.13 Preprocessing overhead 85

5.14 Effect of memory coalescing 85

5.15 Effect of shared memory 85

5.16 Effect of thread divergence 85

6.1 Input graphs 104

6.2 Effect of vertex numbering on exact version (NVR: no vertex renumbering, VR: with vertex renumbering) 104

6.3 Effect of vertex renumbering on global memory coalescing 104

6.4 Performance of ParTBC w.r.t. exact parallel Brandes' algorithm and ABRA (VR: vertex renumbering) Error $\sim 6\%$. 105

6.5 Performance of ParTBC w.r.t. exact parallel Brandes' algorithm and ABRA Error $\sim 10\%$, 20%, 50%. 105

6.6 Overall results 108

LIST OF FIGURES

2.1 Graph representations 13

4.1 Algorithm-wise effect of varying the percentage of outer loop iterations 51

4.2 Graph-wise effect of varying the percentage of outer loop iterations 51

4.3 Work done per iteration in SSSP for rmat26, USA-road, LiveJournal, random26 and twitter 52

4.4 Algorithm-wise effect of varying the percentage of graph processed 54

4.5 Graph-wise effect of varying the percentage of graph processed . . . 54

4.6 Work done across iterations in SSSP for rmat26, USA-road, LiveJournal, random26 and twitter due to partial graph processing 55

4.7 Work done across iterations in Color for rmat26, USA-road, LiveJournal, random26 and twitter due to partial graph processing . . 55

4.8 Algorithm-wise effect of varying the Jaccard's coefficient 57

4.9 Graph-wise effect of varying the Jaccard's coefficient 58

4.10 Work done per iteration for LiveJournal in SCC, SSSP, PageRank, BC and MST for varying Jaccard's coefficient 58

4.11 Work done per iteration in MST due to approximating attribute values 60

5.1 Original graph G and its CSR representation 68

5.2 (a) Graph G from Figure 5.1 with renumbered nodes (b) The same graph reoriented for clarity 70

5.3 Holes in nodes after renumbering G 71

5.4 Modified graph G' with its CSR representation. 73

5.5 Reducing memory latency using shared memory 75

5.6 Handling thread divergence by graph transformation. 76

5.7 Effect of varying the threshold for node replication. 81

5.8 Effect of varying the threshold for clustering-coefficient 82

5.9 Effect of varying the threshold for degree normalization. 83

5.10 Effect of coalescing on energy consumption for BC (higher is better). 86

6.1 Original graph G and its CSR representation 93
6.2 Graph G with renumbered vertices 93
6.3 Neighbors of high-degree vertices 100
6.4 Layout for techniques 100
6.5 Graph-wise effect of varying k on the error for $\alpha = 0.5$ 108
6.6 Effect of varying k on the error for DynRR 109
6.7 Graph-wise effect of varying α on the error for k = 500 111
6.8 Effect of varying α on the error for DynRR 111
6.9 Average rank of top-k vertices missed (expressed as a percentage of k) 113
6.10 Average rank of vertices erroneously included in top-k (expressed as k / Avg. rank) 113

CHAPTER 1

INTRODUCTION

Graph is a fundamental data structure for modeling a broad spectrum of real-world problems. Graph analytics pertains to various fields, such as bioinformatics, machine learning, social network analysis, Computer-Aided Design (CAD), and computer security, among others. Graph analytics algorithms extract useful information from graphs by analyzing their structural properties and how information propagates through them, such as the effect of a drug and identifying communities. Scaling graph algorithms is a challenge today due to the rapid growth of unstructured and semi-structured data.

In recent years, there has been widespread adoption of parallel computing for large scale graph processing. Graphics Processing Units (GPUs) have emerged as the platform of choice for data parallel applications due to the massive data-parallelism they offer and their high memory bandwidth. Our focus in this thesis is graph analytics on GPUs. Former research has targeted parallelizing popular graph algorithms on multi-core CPUs (Shun and Blelloch (2013); Harshvardhan *et al.* (2014, 2015); Grossman *et al.* (2018); Balaji and Lucia (2019)), many-core GPUs (Merrill *et al.* (2012); Gharaibeh *et al.* (2012); Zhong and He (2014); Wang *et al.* (2017); Nodehi Sabet *et al.* (2018)), as well as distributed and heterogeneous systems (Malewicz *et al.* (2010); Low *et al.* (2012); Gonzalez *et al.* (2012); Slota *et al.* (2016); Dathathri *et al.* (2018)). According to the TAO model (Pingali *et al.* (2011)), the primary technical challenge posed by graphs is the inherent *irregularity* in the data-access, control-flow, and communication patterns. This forces compilers to make pessimistic assumptions about them as the graphs are available only at runtime, leading to reduced parallelization benefits. The parallelization issues get exacerbated when GPUs are used to execute graph algorithms. The recent past has witnessed emergence of very effective techniques to represent graphs compactly (Shun *et al.* (2015); Sha *et al.* (2019); Chen *et al.* (2019*a*); Besta *et al.* (2019)), tame irregular computations (Zhang *et al.* (2011); Wu *et al.* (2013); Nodehi Sabet *et al.* (2018); Balaji and Lucia (2019)), and map those to the underlying hardware (Wang *et al.* (2017); Nodehi Sabet *et al.* (2018)).

Despite successful parallelization of graph algorithms, when the graph sizes are huge (e.g., billion-scale networks such as Facebook), and/or the underlying processing is expensive, it is not always feasible to compute the exact solution in time. For instance, finding the importance of a node in a network (using betweenness centrality) necessitates an algorithm having time-complexity cubic in terms of the number of nodes. With billions of nodes in a dynamic network, such a computation is prohibitively expensive. The exact vertex betweenness centrality computation on the undirected graph *liveJournal* (having $\sim$4.8M nodes and $\sim$69M edges) using a parallel implementation of Brandes' algorithm on a GPU takes several days to complete.

While an *exact* answer is required in mission-critical applications, there are graph applications which do not *always* demand an accurate answer and are error tolerant. For example, in network visualization tools such as Gephi, when employing a force-directed graph layout algorithm (e.g., ForceAtlas2), there is a trade-off between the quality of the simulation and the time for convergence. Similarly, we may estimate a set of k nodes with the largest betweenness centrality (BC) in a network faster without computing the exact BC values of the nodes. The exact pagerank values are often not required since they evaluate the relative importance of webpages. The problem of genome assembly lends itself well to approximate computing. Generating the genome sequence entails constructing and traversing the *de Bruijn* graph. The quality of the generated genome sequence is contingent on the constructed *de Bruijn* graph and the path traversed on the de Bruijn graph. Other graph problems amenable to approximate computing include clustering and community detection, among others. With the growing importance of edge-computing and low-energy devices, a practical question is: "given an infrastructure, how to make the best use of resources to obtain an acceptable, inexact solution in a reasonable time?"

As a step towards answering this question, in this thesis, we seek to look beyond the inherent limitations of exact parallel graph processing, w.r.t. performance, using approximate computing. We address the scalability issues in graph parallelization by trading off computational accuracy for improved execution performance. While our proposals embody approximations, they are a departure from the traditional approximation algorithms.

As a first contribution, we propose Graprox, a suite of algorithm- and architecture-independent techniques for approximate parallel graph processing. These techniques are generally applicable to graph processing on GPU and shared memory systems.

The basic execution unit on GPUs is a *wavefront* or a *warp*, wherein threads execute in single-instruction-multiple-data (SIMD) fashion. For best performance, a GPU implementation must be tailored for efficient warp execution. It needs to be optimized along three important dimensions: memory coalescing (Zhang *et al.* (2011), memory latency Nasre *et al.* (2013*c*)), and thread divergence (Nodehi Sabet *et al.* (2018)). Graph processing poses challenges for coalesced memory accesses due to arbitrary and unknown connectivity between graph vertices. A common strategy for improving coalescing is reordering of vertices. It is useful in improving the spatial locality of vertices by assigning consecutive id's to those that are likely to be accessed in tandem (Nodehi Sabet *et al.* (2018); Balaji and Lucia (2019)). Thus, the graph vertices could be pre-numbered based on the connectivity, so that neighbors of vertices being processed by warp-threads are nearby in GPU memory (typically, the vertices are numerically indexed). The second crucial dimension for efficient GPU execution is memory latency. Graph algorithms are often memory-bound due to the irregular memory access patterns and the resulting reduced cache benefits, making them more sensitive to memory latency. In the presence of hundreds of thousands of threads running on the GPU, per-thread cache benefits are further diminished. Therefore, literature has proposed various mechanisms such as kernel unrolling and usage of shared memory to reduce memory latency (Nasre *et al.* (2013*c*); Khorasani *et al.* (2014)). Using shared memory requires identifying reusable attribute data (at the vertex or the edge) in the graph algorithm and taking advantage of the temporal locality. The third necessary dimension for efficient GPU execution is thread-divergence. It occurs when warp-threads need to execute different instructions (or no-op) at the same time, resulting in loss of parallelism. Thread-divergence is rampant in graph algorithms due to arbitrary degree-distribution, leading to load-imbalance. For skewed degree distributions prevalent in several real-world graphs, load-imbalance poses a sequentiality bottleneck. Former research has proposed degree-sorting, nested kernels, loop-splitting, and edge-based processing to reduce thread-divergence (Zhang *et al.* (2011); Balaji and Lucia (2019)).

As a second contribution, we propose algorithm-independent but architecture-specific approximate techniques for parallel graph processing. We present Graffix, a system of graph transformation techniques for making the graph *more* amenable to GPU-based processing while adding controlled approximations. We target three important GPU-specific aspects central to performance: memory coalescing, memory latency, and thread-divergence. We improve memory coalescing by making the graph more structured, reduce memory latency by processing the well-connected subgraphs inside shared memory, and alleviate thread divergence by reducing the workload imbalance among warp threads.

As mentioned earlier, graph algorithms that take excessively long to complete and can tolerate some degree of inexactness in the output are well suited for approximate processing. With this motivation, we look at the problem of identifying the most important nodes in a network, based on their betweenness centrality scores.

Betweenness centrality (BC) is a crucial centrality metric in graphs and networks that measures the significance of a vertex. BC(n) is calculated using the number of shortest paths in the graph passing through n. It is used in a variety of applications such as detecting communities in social and biological networks (Girvan and Newman (2002)), targeted advertising (Kim and Park (2012)), analysis of disease spreading (Liljeros *et al.* (2001)), and identifying criminal networks (Coffman *et al.* (2004)), among others. The state-of-the-art Brandes' algorithm (Brandes (2001)) computes the exact BC values for all nodes in a graph $G = (V, E)$ in time $\mathcal{O}(|V||E|)$ for unweighted graphs, and time $\mathcal{O}(|V||E| + |V|^2 \log |V|)$ for graphs having positive weights. As suggested by its complexity, computation of BC is quite time-consuming even on graphs of moderate sizes, having hundreds of thousands of nodes and edges. For example, a single-threaded execution of Brandes' algorithm takes over 13 hours to terminate on an undirected graph *loc-Gowalla* (having $\sim$196,600 vertices and $\sim$950,300 edges).

To make BC computations scalable, Brandes' algorithm has been successfully parallelized on multi-core CPUs, many-core GPUs, and distributed systems (Madduri *et al.* (2009); McLaughlin and Bader (2014); Prountzos and Pingali (2013); Solomonik *et al.* (2017); Hoang *et al.* (2019)). Yet, the cost of BC computation is excessive on modern networks with millions of nodes and tens of millions of edges. Moreover, often

applications are interested in the relative ranking of the vertices according to their BC scores, rather than their actual BC values. In addition, several applications demand identifying nodes with highest BC values. Hence, an estimate of the top-k BC vertices is sufficiently informative in such cases.

Thus, as a third contribution, we explore algorithm-specific but architecture-independent techniques for parallel graph processing. Specifically, we propose ParTBC, a framework for computing the top-k vertices with highest BC in a graph, using approximate computing in conjunction with parallelization. We compute approximate BC values of vertices, such that the relative ordering among the vertices is *nearly* maintained.

1.1 Contributions

The thesis makes the following contributions towards increasing the efficiency of parallel graph algorithms in exchange for small inaccuracies in the final output.

1.1.1 Architecture-agnostic Techniques for Parallel Approximate Graph Processing

We propose Graprox, a set of four architecture-agnostic techniques for parallel graph processing that exploit the general structure of a graph kernel and the flow of information in graph algorithms to arrive at an approximate solution early. Graph algorithms are often iterative. We cut-short the outerloop iterations, based on a stopping criterion, to compute an approximate solution faster, by reducing the overall work done. Further, not all parts of the graph contribute equally to the final fixed-point information. So, we propose to process a subset of the vertices/edges selectively, for each pass through the graph, or for each iteration of the outermost loop. We also explore the effect of storing the graph in an imprecise manner. Instead of working on the exact graph representation, the (exact) algorithm runs on the compressed graph, obtained by merging the nodes with significantly overlapping neighborhoods. We also propose techniques to reduce the computation cost of large graphs by approximating the attribute values of

graph elements. We provide *tunable knobs* to control the accuracy-performance trade-off. We empirically demonstrate the efficacy of our techniques using popular graph algorithms and graphs of different characteristics.

1.1.2 GPU-specific Optimizations for Graph Processing in the Presence of Approximations

We propose Graffix, a framework for approximate computing techniques to improve coalescing, memory latency, and thread divergence of graph processing kernels on GPU. Each of the proposed techniques modifies the graph structure to accomplish the goal. Our techniques offer tunable knobs to control the amount of approximation injected. Our technique for improving memory coalescing creates a graph isomorph that brings relevant nodes nearby in memory and adds controlled replica of nodes to improve coalescing. The second reduces memory latency by facilitating the processing of subgraphs inside shared memory by adding edges among specific nodes and processing well-connected subgraphs iteratively inside shared-memory. The third technique normalizes degrees across nodes assigned to a warp to reduce thread divergence. As we show, our techniques are effective in improving the performance by $1.13\times$ on average with inaccuracy in the ballpark of 10% across various real-world and synthetic graphs for popular graph algorithms.

1.1.3 Estimation of Top-k Betweenness Centrality Vertices on Heterogeneous Architectures

We propose ParTBC, a bouquet of novel techniques for speeding up the estimation of top-k vertices with the highest BC in a graph, using approximate computing on GPU. Our techniques restrict the computation of shortest paths from only a fraction of the vertices in parallel Brandes' algorithm based on online stopping criteria that use tunable knobs. The techniques govern the subset of the vertices that are picked as sources and their order while ensuring that all graph vertices receive a sizeable fraction of their respective BC scores in the early iterations of Brandes' algorithm. BC scores of the nodes

so acquired (in the early iterations) cause the relative BC scores of the graph vertices to be indicative of their *relative* exact BC values, leading to a high accuracy in top-k computation with fewer iterations. On real-world and synthetic graphs, on average, we achieve a speedup of $2.5\times$ compared to the exact parallel Brandes' algorithm on GPU, with an error of less than 6%.

1.2 Organization of the Thesis

The thesis is organized as follows. Chapter 2 introduces the basic terminology related to graphs and describes the popular graph representations. We describe the parallel versions of the popular graph algorithms and the various graph classes used in the thesis.

Chapter 3 describes the prior art in the area of parallel and approximate graph processing, and compares and contrasts those with the ideas presented in this thesis.

Chapter 4 describes in detail our proposed approximate techniques for parallel graph processing. We present a theoretical model of approximation and instantiate it with concrete approximate computing techniques. Our techniques perform reduced execution, process only part of the graph, store graph in an approximate manner, and approximate attribute values to gain in efficiency. We also discuss how approximations can be exploited for an efficient GPU-based parallel processing. All the techniques provide tunable knobs to change the degree of approximation and control the performance-accuracy trade-off. The experimental results validate the efficacy of our proposals.

Chapter 5 details our proposed approximate computing techniques targeting GPU-specific aspects for efficient graph processing. We propose three graph transformations, each targeting one GPU-specific aspect — memory coalescing, memory latency and thread-divergence. The experimental results show that our techniques are indeed effective in improving the performance on real-world graphs and popular graph algorithms. Further, our techniques do not compete with the existing GPU-specific optimizations, but complement those.

Chapter 6 discusses our proposal for speeding up the identifation of the most important nodes in a network based on their betweenness centrality values on GPU. We first

describe a graph reordering mechanism to improve the memory coalescing during BC computation. We next discuss various heuristics for determining the sequence in which to pick nodes as source (in the outermost loop) in Brandes' algorithm, to enable quicker identification of the top-k BC vertices with a very small error. Finally, we present the experimental results which show that our techniques work well in practice.

Chapter 7 concludes the thesis and discusses a few future directions.

CHAPTER 2

Background

In this chapter, we introduce the terminology and notation used throughout the thesis and present an overview of the concepts required for understanding the remainder of the thesis.

2.1 Basic Definitions

Graph. A graph $G(V, E)$ is a set of *vertices*, V, and a collection of *edges*, E, each connecting a pair of vertices (u, v) where u and $v \in V$. If (u, v) is an ordered pair, the graph is *directed*. If (u, v) is an unordered pair, the graph is *undirected*.

Degree of a vertex. The *degree* of a vertex u is the number of edges $(u, v) \in E$.

Tree. A *tree* is an undirected graph in which there is exactly one path between every pair of vertices. A disjoint set of trees is called a *forest*.

Diameter of a graph. The length of the longest shortest path in the graph is its *diameter*.

Clustering coefficient. *Clustering-coefficient* of a graph is a measure of the tendency of vertices to occur in clusters.
Local clustering-coefficient (LCC) of a vertex, v:

$$\text{LCC}(v) = \frac{\text{number of pairs of } v\text{'s neighbors that are neighbors}}{\text{number of pairs of } v\text{'s neighbors}}$$

The definition of local clustering coefficient can be extended to define the clustering coefficient of a subgraph.

2-hop neighbor. The 2-hop neighbors of a vertex u are those vertices that are reachable from u in exactly 2 hops. A vertex w is a 2-hop neighbor of u ($u \neq w$) if $\exists\, v \in V$ s.t. (u, v) and $(v, w) \in E$.

2.2 Graph Storage Formats

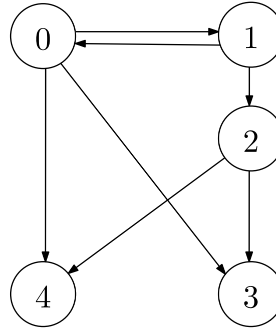
There are two considerations when deciding on the graph representation:

1. space complexity of the representation
2. time-efficiency of accessing the graph elements

We describe two popular graph representations: 1) Adjacency matrix; 2) Compressed Sparse Row (CSR)

Adjacency Matrix. An adjacency matrix is a $|V| \times |V|$ matrix, with the entry in row u and column v set to 1 if there is an edge from vertex u to vertex v, and 0 otherwise. This representation has a space complexity of $|V|^2$ making it prohibitive even for graphs with a few million vertices, which are very common. Figure 2.1a presents an example graph G, whose adjacency list representation is shown in Figure 2.1b. Adjacency matrices for real-world graphs are often sparse (i.e., most of the entries are 0), resulting in suboptimal utilization of the allocated space. Processing a graph involves visiting the neighbors of a vertex. The time complexity of accessing a vertex's neighbors with this representation is $\mathcal{O}(|V|)$.

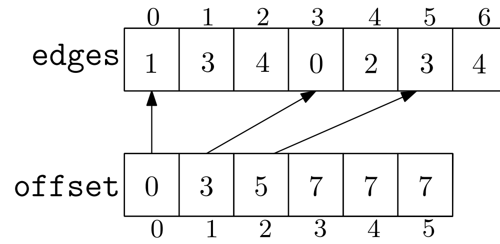
Compressed Sparse Row. The CSR representation is a popular storage format for sparse matrices. In the context of graphs, it stores only the non-zero elements of the (sparse) adjacency matrix, i.e., the edges of the graph. Figure 2.1c shows the CSR representation of the graph G. The CSR format requires two arrays to represent the graph: *offset* array and *edges* array. The offset array is sorted by vertex-id and stores each vertex's starting offset into the edges array. The edges array stores the neighbors of the vertices contiguously, that is, the neighbors of vertex 0, followed by neighbors of vertex 1 and so on. The offset array is of size $|V| + 1$ and the edges array is of size $|E|$. Thus CSR representation has a space complexity of $\mathcal{O}(|V| + |E|)$. Additionally,



(a) An example directed graph G

	0	1	2	3	4
0	0	1	0	1	1
1	1	0	1	0	0
2	0	0	0	1	1
3	0	0	0	0	0
4	0	0	0	0	0

(b) Adjacency Matrix representation of G



(c) CSR representation of G

Figure 2.1: Graph representations

the degree of a vertex, v is given by (offset[$v + 1$] - offset[v]). The time complexity of accessing a vertex's neighbors is $\mathcal{O}$(vertex degree).

Further, in GPU-based graph processing, the graph is read on the host and then transferred to the GPU memory. The CSR representation offers an additional advantage for data transfer to the GPU. The graph can be copied to the GPU using two calls to cudaMemcpy(): one for the edges array and another for the offset array in the CSR representation. Notwithstanding that CPU and GPU have separate address spaces, CSR representation allows us to access the graphs identically on both the host and the device without additional overheads.

This contrasts with using a traditional (space-efficient) adjacency list representation, which uses an array of pointers (one per vertex) with each vertex pointing to a list of its neighbors. Transferring such an array of adjacency lists to the GPU requires moving the neighbor list of each vertex separately. Additionally, we are required to perform serialization–deserialization of the pointers on the GPU.

We use the CSR representation in all our works.

2.3 Classification of Graphs

We describe the nature and characteristics of the class of graphs that we have used in the evaluation of our proposed techniques.

2.3.1 Scale-free Real World graphs

Real-world graphs, such as social networks, are often scale free. A graph is said to be scale free if the fraction of nodes with degree d follows a power law distribution $d^{-\alpha}$, where $\alpha > 1$.

A few properties of the real-world scale free graphs are:

- Very few nodes have large node degrees, while a large fraction of nodes has a very low degree.
- There is a huge difference between the highest and the lowest node degrees.
- The diameter of the graph is small.
- There is a hierarchical community structure.

2.3.2 Recursive Matrix (R-MAT) graphs

R-MAT graphs (Chakrabarti *et al.* (2004)) are synthetic scale free graphs. The R-MAT graph generation model generates graphs by recursively (sub)dividing a matrix of size $|V| \times |V|$ into four equal partitions and distributing the edges among the partitions with unequal probabilities $\alpha, \beta, \gamma, \delta$ such that $\alpha + \beta + \gamma + \delta = 1$.

2.3.3 Road networks

In a road network, a vertex represents an intersection of roads, and the edges represent the roads. The diameter of such graphs is large. Since the number of roads intersecting at a junction is bounded and small, road networks have small node degrees, and the difference between the highest and lowest node degrees is also low. Such networks are nearly planar.

2.3.4 Random graphs

Random graphs are obtained by adding edges at random, starting from a set of disconnected vertices. The Erdős-Rényi model is frequently used for generating random graphs. With this model, an edge is included with a constant probability, independent of other edges. Random graphs have small diameters and the difference between the highest and lowest vertex degrees is also small.

2.4 Parallel Graph Algorithms

We describe the parallel algorithms for the graph problems that we have used for evaluating the proposed approaches in this thesis. These graph algorithms are well-known and are widely used in a myriad of applications.

2.4.1 Breadth First Search (BFS)

Breadth First Search is a graph traversal strategy, wherein starting from a designated source node, the graph vertices are visited level by level. First, all the neighbors of the source vertex are visited, followed by its 2-hop neighbors, and so on. BFS is well-suited for parallelization. Algorithm 1 shows the pseudo-code for a work-efficient BFS traversal. There is a worklist for the nodes to be processed. Initially, the worklist contains only the BFS source (Line 3). The vertices in the worklist are processed in parallel by different threads to update the level information of their neighbors (Line 4). The worklist is updated with the nodes to be processed next. When the nodes at level l are processed, nodes with level $l+1$ are added to the worklist. The algorithm terminates when there are no more nodes left to be processed, i.e., the worklist is empty. It works for a single connected component, and for a forest, it needs to be repeated for a source in each component.

We use BFS traversal extensively in our work.

Algorithm 1 Breadth First Search Traversal on $G(V, E)$

```
1:  v.level = 0    ∀v ∈ V                          ▷ initialization
2:  source.level = 1
3:  Worklist wl = {source}
4:  for all Node n : wl do                          ▷ GPU parallel
5:      for Node v : G.neighbors(n) do
6:          if v.level == 0 then
7:              v.level = n.level + 1
8:              wl.push(v)
9:          end if
10:     end for
11: end for
```

2.4.2 Single Source Shortest Path (SSSP)

Algorithm 2 SSSP Computation over graph $G(V, E)$

```
1:  v.dist = ∞    ∀v ∈ V                           ▷ initialization
2:  source.dist = 0
3:  Worklist wl = {source}
4:  changed = true
5:  while changed do                                ▷ outer loop
6:      changed = false;
7:      for all Node u : wl do                      ▷ GPU parallel
8:          for Node v : G.neighbors(u) do
9:              altdist = u.dist + weight(u → v)
10:             if altdist < v.dist then
11:                 v.dist = altdist                ▷ needs synchronization
12:                 wl.push(v)
13:                 changed = true;
14:             end if
15:         end for
16:     end for
17: end while
```

SSSP computation finds the shortest distance of each vertex from a designated source in a weighted graph. Bellman-Ford algorithm and Dijkstra's algorithm are two of the popular algorithms for SSSP computation. We implement a variation of Bellman-Ford's algorithm which is more amenable to parallelization than the work-efficient Dijkstra's algorithm. Algorithm 2 presents the pseudo-code for SSSP computation that we use in our work. The algorithm iteratively updates the distances of the graph's nodes from the designated source till a fixed point is reached. A worklist is maintained that contains the nodes whose distances have been updated. Initially, the worklist contains only the source node. The distance of the source is initialized to 0 while every other node's distance is initialized to ∞. The nodes in the worklist are processed in parallel

(Line 7) and the distances of their neighbors are updated. Due to multiple threads writing to the same vertex's distance, threads need to synchronize using *atomics* (Line 11).

2.4.3 PageRank

Algorithm 3 PageRank Computation over graph $G(V, E)$

1: PR_old[v] $= \frac{1}{|V|}$ $\quad \forall v \in V$ ▷ initialization
2: PR_new[$|V|$]
3: adjustfactor $= \frac{1-d}{|V|}$ ▷ d is the damping factor
4: **for** $i = 1..M$ **do** ▷ Number of iterations
5: PR_new[v] $= 0$ $\quad \forall v \in V$ ▷ Reset PR_new
6: **for all** Node $u : V$ **do** ▷ GPU parallel
7: **for** Node $v : G$.neighbors(u) **do**
8: PR_new[v] += $\frac{d \times PR_old[u]}{outDegree(u)}$ ▷ needs synchronization
9: **end for**
10: **end for**
11: ***– barrier –***
12: **for all** vertices $v \in V$ **do** ▷ GPU parallel
13: PR_new[v] += adjustfactor
14: **end for**
15: ***– barrier –***
16: $swap$(PR_new, PR_old)
17: **end for**

PageRank is a propagation-based algorithm to compute page rank values (related to importance) of vertices in a web-graph. The algorithm uses a damping factor d, which is usually set to 0.85. Then pagerank of a vertex v in a graph $G(V, E)$ is defined as:

$$PR(v) = \frac{1-d}{|V|} + d \times \sum_{\forall u \rightarrow v} \frac{PR(u)}{outDegree(u)} \tag{2.1}$$

Algorithm 3 presents the pseudo-code for PR that we use in our work. The algorithm is iterative and is generally run for a fixed number of iterations depending on the required precision of the pagerank score. In our experiments, we run the algorithm for 10 iterations, i.e., $M = 10$. In each iteration, the nodes are processed in parallel (Line 6) and the threads operating at a node update its neighbors' pagerank values according to equation 2.1. Due to multiple threads writing to the same vertex's pagerank value, threads need to synchronize using *atomics* (Line 8).

2.4.4 Minimum Spanning Tree (MST)

Algorithm 4 MST Computation over graph $G(V, E)$

```
1:  MSTset = { }                                          ▷ set of edges in MST
2:  union-find uf(V)                                      ▷ union-find data-structure
3:  Workset W = V                                         ▷ set of components' rep. elements
4:  for all Node u : W do
5:      Edge e = getMinWtEdge(u → v) ∀ Node v ∈ G.neighbors(u)
6:      Node x = uf.find(e.src)
7:      Node y = uf.find(e.dst)
8:      if x ≠ y then
9:          MSTset = MSTset ∪ {e}                         ▷ include e in MST
10:         rep = uf.union(x, y)                          ▷ edge contraction
11:         W.insert(rep)                                 ▷ include rep. element in W
12:     end if
13: end for
```

MST computation finds a tree in a given graph having the minimum sum of the tree's edge-weights and which spans all the vertices of a connected graph. Popular graph algorithms for MST computation are Prim's algorithm, Kruskal's algorithm and Boruvka's algorithm. We use Boruvka's algorithm which offers better parallelism over Prim's or Kruskal's algorithms. MST computation necessitates the maintenance of various components across multiple iterations and these components are processed in parallel, using union-find (or its variant), which is a standard data structure maintained to process MST efficiently. Algorithm 4 presents the pseudo-code for MST that we use in our work. In this algorithm, MST is computed using successive *edge-contraction*. The algorithm starts with every node in a separate component (Line 3). The minimum-weight edge from each node to a different component is identified in parallel (Line 5) followed by finding the end-points of the minimum-weight edge between components for every component (Lines 6,7). The end-points so identified are fused into a single component in parallel (Line 10). The minimum-weight edge between components form the MST. The algorithm terminates when only a single component is left.

2.4.5 Strongly Connected Components

A strongly connected component (SCC) of a directed graph $G(V, E)$ is a set of vertices $V' \subseteq V$ such that there is a directed path between every pair of vertices in V'. Finding SCC is a fundamental problem which identifies cycles in a given graph. We use

the Forward-Backward (FB) algorithm for SCC which offers better parallelism over a depth-first search based processing, such as Kosaraju algorithm. Forward-Backward (FB) algorithm performs two traversals on the graph from each node, and forms two sets of vertices $F(v)$ and $B(v)$ for each vertex v corresponding to forward set and backward set respectively. The graph nodes can then be divided into four parts: $V-(F(v) \cup B(v))$, $F(v) \cap B(v)$, $F(v) - (F(v) \cap B(v))$ and $B(v) - (F(v) \cap B(v))$. The intersection $F(v) \cap B(v)$ is an SCC. This process is repeated on the remaining three disjoint sets in parallel. Algorithm 5 presents the pseudo-code for SCC that we use in our work. The pivot node is selected using a heuristic (Line 5). The forward and backward reachability closures of the selected pivot are computed in parallel (Lines 6, 7). Further, each of these is parallelized using a frontier-based BFS traversal. Since each of the residual graphs (Lines 11–13) is disjoint, they are processed in parallel.

Algorithm 5 SCC Computation over graph $G(V, E)$

```
1:  function FW-BW(G, SCC)
2:      if |V| == 0 then
3:          return
4:      end if
5:      u ← pivot(G)                     ▷ select a pivot vertex
    begin in parallel
6:      D ← BFS(G, u)                    ▷ vertices reachable from pivot; GPU parallel
7:      P ← BFS(G^R, u)                  ▷ vertices that can reach pivot; GPU parallel
    end
8:      S ← P ∩ D                        ▷ an SCC
9:      SCC ← SCC ∪S                     ▷ the set of SCCs
10:     R ← V \ (P ∪ D)                  ▷ remaining vertices
    begin in parallel
11:     FW-BW(D \ S, SCC)
12:     FW-BW(P \ S, SCC)
13:     FW-BW(R, SCC)
    end
14: end function
```

2.4.6 Vertex Coloring

Vertex coloring is the problem of assigning colors (or labels) to vertices of a graph such that adjacent vertices are assigned distinct colors, using as few colors as possible. Vertex coloring is NP-hard, but several heuristics-based algorithms exist for vertex coloring that try to reduce the number of colors used. In our work, we use a paral-

lel greedy independent-set based algorithm. Algorithm 6 presents the pseudo-code for vertex coloring that we use in our work. It uses the observation that an independent set of vertices can be assigned the same color. Further, we use the largest-degree-first ordering (LDF) heuristic, i.e., the vertices with larger degrees are colored first. In each iteration, an independent set is constructed in parallel by choosing vertices that have the highest degree locally (Line 10). As a result, the nodes are colored in decreasing degree order. The vertices belonging to an independent set are assigned the same color in parallel (Line 26). In an iteration, the vertices are colored using the smallest available color. The algorithm terminates when all nodes have been colored.

Algorithm 6 Vertex Coloring over graph $G(V, E)$

```
 1: v.color = -1        ∀v ∈ V                              ▷ initialization
 2: v.weight = v.degree        ∀v ∈ V
 3: v.conflictRes = randomInt()        ∀v ∈ V
 4: U = V                                      ▷ U is the set of uncolored nodes
 5: c = 0                                                ▷ Color to be assigned
 6: flag = true
 7: while flag do                               ▷ there are uncolored vertices
 8:     I = { }                                            ▷ Independent Set
 9:     while |U| > 0 do
10:         for all Node v : U do                            ▷ GPU parallel
11:             w_max = 0
12:             for Node w : G.neighbors(v) do
13:                 if w_max < w.weight then
14:                     w_max = w.weight
15:                     x = w
16:                 end if
17:             end for
18:             if v.weight > w_max  then
19:                 I = I ∪{v}
20:             else if v.weight > w_max then
21:                 if v.conflictRes > x.conflictRes then
22:                     I = I ∪{v}
23:                 end if
24:             end if
25:         end for
26:         for all v′ ∈ I do                                ▷ GPU parallel
27:             v′.color = c
28:         end for
29:     end while
30:     U = U \ I
31:     if |U| == 0 then
32:         flag = false
33:     end if
34:     c = c + 1
35: end while
```

2.4.7 Betweenness Centrality

Betweenness Centrality (BC) of a vertex, v, quantifies its importance based on the number of shortest paths in the graph passing through v. The state-of-the-art Brandes' algorithm (Brandes (2001)) computes the exact BC values for all nodes in a graph $G = (V, E)$ in time $\mathcal{O}(|V||E|)$ for unweighted graphs, and time $\mathcal{O}(|V||E| + |V|^2 \log |V|)$ for graphs having positive weights. Algorithm 7 presents the pseudo-code for Brandes' algorithm that we use in our work. In this algorithm, there is an outerloop over the number of nodes (Line 2). Each of the computation steps (Lines 3, 7) is executed in parallel for a single source, and different sources are processed in sequence.

Algorithm 7 Betweenness Centrality computation over graph $G(V, E)$

Require: An undirected, unweighted graph $G(V, E)$
Ensure: Vertex betweenness centrality

```
 1: bc[v] = 0      ∀v ∈ V                              ▷ initialization
 2: for all s ∈ V do
      ▷ Forward Pass: form BFS DAG, D
 3:     for all v : Node ∈ G do                        ▷ GPU parallel
 4:         compute σ_sv
 5:         compute pred(s, v)
 6:     end for
      ▷ Backward Pass: backward traverse DAG, D
 7:     for all v : Node ∈ D do                        ▷ GPU parallel
 8:         compute δ_s(v)
 9:         bc[v] += δ_s(v)
10:     end for
      ▷ Reset graph attributes
11:     for all (u → v) ∈ E do
12:         reset(u → v)
13:     end for
14: end for
```

σ_{sv} (Line 4) is the number of shortest paths from s to v. The *dependency* of a vertex v w.r.t. a given source vertex s is $\delta_s(v)$. It is computed (in Line 8) using the following recurrence:

$$\delta_s(v) = \sum_{w|v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)) \tag{2.2}$$

$pred(s, w)$ is a list of immediate predecessors of w in the shortest paths from s to w (computed using the forward pass). The size of the $pred$ list of a vertex is bounded by its degree. $pred$ lists of all the vertices together induce a directed acyclic graph (DAG) D over the graph G. BC of each vertex is then computed (in Line 9) as a summation

over all the sources (computed using the backward pass):

$$bc(v) = \sum_{s \neq v \in V} \delta_s(v) \tag{2.3}$$

2.5 Summary

In this chapter, we introduced the basic terminology related to graphs in general. We discussed the relevant graph representations and the various types of graphs encountered in practice. Finally, we presented the parallel algorithms for the popular graph problems, which we use to evaluate the techniques proposed in the thesis.

CHAPTER 3

Related Work

This chapter discusses the prior works in the realm of parallel graph processing and approximate computing, and tries to place our contributions towards parallel approximate graph analytics in context. We divide the past works into two categories: 1) Exact parallel graph processing 2) Approximate graph processing.

3.1 Exact Parallel Graph Processing

There has been a vast array of works on parallelization and optimization of graph algorithms on various parallel architectures, including multicore CPU, manycore GPU and distributed systems. We discuss a few prominent ones here.

3.1.1 Parallel graph processing on multicore

Shun and Blelloch (2013) present Ligra which is a framework for parallel graph processing on shared memory systems. Ligra supports vertex based processing using frontiers and edge-map. The active vertices are added to a frontier. The edge-map updates the neighbor information of all the nodes present in the frontier and updates the frontier. It uses coordinated scheduling for synchronization. It supports both push- and pull-based operators and can switch between them based on the number of edges incident on the frontier.

Grossman *et al.* (2018) propose a framework for efficient pull-based processing on shared memory systems. They present a *scheduler-aware* interface for parallel loops which allows programmers to reduce the write-conflicts in scenarios where a thread executes several iterations of the parallel loop. This benefits the pull-based processing. Further, they introduce a new graph representation called Vector-Sparse that facilitates

better vectorization of the loop over the in-neighbors of a vertex in pull-based processing.

The approximate computing techniques presented in the thesis are designed for push-based implementation of graph algorithms following the *vertex-centric* model of parallelization.

Harshvardhan *et al.* (2014) introduce k-level asynchronous (KLA) paradigm for parallel graph processing. KLA tries to bridge the gap between level-synchronous and fully asynchronous processing. Tuning the value of k enables controlling the number of (expensive) global and (cheap) local synchronizations. An appropriate choice of k improves the performance of the traditionally level-synchronous graph primitives by reducing the cost of synchronization.

In our implementation of graph algorithms, we pick either the level-synchronous or fully asynchronous processing.

Madduri *et al.* (2009) propose an efficient parallel implementation for computing vertex betweenness centrality on shared memory multicore architectures. They improve the algorithm to use successors instead of predecessors in the computation of the DAG, which produces a more efficient, locality-friendly algorithm.

With the purpose of alleviating the issues due to irregular accesses in parallel graph processing, various graph reordering schemes have been explored such as LabelPropagation (Boldi *et al.* (2011)), Nested Dissection (Lasalle and Karypis (2013)), SlashBurn (Lim *et al.* (2014)), Gorder (Wei *et al.* (2016)) and ReCALL (Lakhotia *et al.* (2017)). Reverse Cuthill-McKee (RCM) Liu and Sherman (1976) is a commonly used algorithm for reordering sparse matrices and graphs. It uses a modification of BFS. RCM performs level order traversal such that nodes at a level are visited in order of their BFS parent's position in the previous level. If multiple nodes have the same earliest BFS parent, they are picked in descending degree order. Karantasis *et al.* (2014) present a parallel version of RCM to reduce the cost of reordering. Balaji and Lucia (2019) propose RADAR, which combines data duplication and graph reordering to accelerate graph processing on shared memory systems. It uses *degree-sorting* to assign consecutive id's to the highly-connected *hub* vertices. Following the reordering, it cre-

ates per-thread copy for the hub vertices to reduce false sharing and the cost of atomic updates.

We propose a graph reordering technique for better coalesced accesses of vertex-centric graph algorithms on GPUs in Chapter 5, which involves vertex renumbering. Our renumbering scheme is different from the previous ones in that it takes into account the nodes that are likely to be accessed in tandem in GPU-based processing and tries to place them together in memory.

To enable efficient processing of large graphs, out-of-core processing has been explored (Kyrola *et al.* (2012); Roy *et al.* (2013); Maass *et al.* (2017); Dhulipala *et al.* (2018); Jun *et al.* (2017)). Harshvardhan *et al.* (2015) propose a hybrid approach for out-of-core processing wherein a graph is partitioned into multiple subgraphs, each of which can fit into the RAM. The subgraphs are swept in and out of the main memory for processing using a paging-like approach. They cut down on disk I/O by implementing optimizations such as deferred updates, avoiding redundant writes to disk, bringing in multiple subgraphs into the RAM at a time. Their proposed scheme works well on single shared-memory node, as well as, large distributed systems. Gill *et al.* (2020) study graph analytics on large graphs using byte-addressable Intel Optane Persistent Memory Modules.

Our proposals target static graphs that fit completely in memory and are not designed to work for out-of-core processing.

3.1.2 Parallel graph processing on GPU

Parallelization of popular graph algorithms on GPUs has been well-studied and various optimization strategies have been explored (Carrillo *et al.* (2009); Luo *et al.* (2010); Hong *et al.* (2011); Nasre *et al.* (2013*a*,*b*); Khorasani *et al.* (2014); Ashari *et al.* (2014); Sengupta *et al.* (2015); Pai and Pingali (2016); Han *et al.* (2017)).

Merrill *et al.* (2012) propose work-efficient frontier-based graph traversal on GPUs. They use prefix-sum to determine the enqueue offset for each of the threads that enables gathering the nodes in parallel while expanding the global vertex-frontier queue. They further propose exploiting fine-grained and coarse-grained parallelism for reduc-

ing workload imbalance during expansion of the queue and filtering the already visited and duplicate vertices.

Gharaibeh *et al.* (2012) present *Totem*, a framework for processing a graph on heterogeneous systems. *Totem* divides the graph into two partitions. One part is stored in main memory and processed on multicore CPU. The other part is processed on the GPU. The CPU and GPU processing proceeds simultaneously. It is able to process large graphs that do not entirely fit in the GPU device memory. *GStream* due to Seo *et al.* (2015) alleviates the issue of underutilization of GPU in Totem by using the concept of *nested-loop theta-join* operation and utilizing asynchronous GPU streams. GPU kernel is treated as the theta-operator and the attribute vectors and topology data are treated as the outer and the inner join operands respectively.

Zhang *et al.* (2011) present techniques for removal of *dynamic irregularities* in GPU computation, to effect better memory coalescing. They use data reordering and job swapping, and runtime adaptation techniques for effective reduction in dynamic irregularities. Nasre *et al.* (2013*c*) discuss using shared memory for storing part of the worklist, and maintaining local worklist needed for kernel unrolling in the case of data-driven and topology-driven approaches respectively. Zhang *et al.* (2010) study methods for thread divergence removal. They minimize thread divergence through runtime optimizations with the support of a CPU-GPU pipeline scheme.

Wang *et al.* (2017) present Gunrock, a library for parallel graph analytics on GPU. Gunrock operates on frontiers of nodes or edges. A *filtering* operation removes inactive nodes or edges from the frontier followed by application of user-defined functors to the items in the frontier in parallel. It also employs four parallel graph traversal throughput optimization strategies to reduce workload imbalance in graph processing on GPU. We compare our techniques against Gunrock in Chapter 5.

Hong *et al.* (2017) propose different graph representations based on the graph characteristics and traversal patterns to alleviate the issues in GPU-based graph processing. They present MultiGraph which preprocesses the graph to group the vertices into different categories based on their in-degrees and reorders the vertices such that those in the same category are assigned consecutive id's. Further, it chooses between the *heavy-block* and the *sparse block* representation based on the estimate of the active vertices

during the frontier-based iterative processing of the graph.

Nodehi Sabet *et al.* (2018) present Tigr that addresses the graph irregularity issues by transforming the graph to make it more structured. Tigr uses virtual split transformation and memory access optimization, called *edge-array coalescing* to reduce the thread divergence and to improve the data locality in vertex-centric graph processing. Split transformation introduces indirection arrays to maintain (virtual) replicas of nodes having degree greater than the degree-bound and distributing the edges among the replicas, to make node degrees uniform. The edge-array coalescing technique reorders the edges among the replicated nodes during the construction of CSR arrays and the edge-array is accessed in a strided manner to effect improved coalesced accesses of the edge-array after the split transformation. We compare our techniques against Tigr in Chapter 5.

There have been works focusing on parallelizing betweenness centrality computation on GPUs. Prountzos and Pingali (2013) propose a space efficient scalable asynchronous parallel algorithm for BC that is able to extract massive parallelism. They express the BC computation in terms of *operators*, which act on nodes and edges to update their attributes if the specified preconditions are satisfied. The set of operators for the forward and the backward passes are different, since these capture the computation involved in the two phases. They also propose a set of schedules for applying the operators.

McLaughlin and Bader (2014) present an efficient scheme for parallel BC computation on GPUs and heterogeneous architectures. They adopt a hybrid approach wherein they choose one of work-efficient or edge-parallel approaches for each outer-loop iteration of the algorithm based on whether the size of the vertex-frontier is small or large. This reduces the load imbalance among threads while utilizing the available parallelism.

3.1.3 Parallel graph processing on distributed systems

Graph algorithms have been shown to bear enough parallelism in the context of distributed systems and various strategies for efficient parallelization and low communication have been studied (Malewicz *et al.* (2010); Low *et al.* (2012); Gonzalez *et al.* (2012); Hong *et al.* (2015); LeBeane *et al.* (2015); Slota *et al.* (2016); Zhu *et al.* (2016);

Besta *et al.* (2017); Chen *et al.* (2019*b*)).

Dathathri *et al.* (2018) propose various communication optimizations and graph partitioning strategies for distributed memory graph analytics systems. They introduce the *proxy partitioning model* wherein edges are distributed among the host machines. The end points of these distributed edges are locally cached on the host to create proxy vertices. One copy of a vertex in the graph is a master proxy, while others are mirror proxies. During synchronization, the mirror proxies communicate their local values to the master proxy, which will hold the canonical value. In order to reduce the communication overhead, they use *vertex-id memoization* which eliminates the need for sending global vertex-id's across hosts and the translation between global and local vertex-id's.

Parallelization of BC computation on distributed memory systems has been extensively explored. Solomonik *et al.* (2017) propose a parallel algorithm for BC computation using an algebraic formulation. They use the Bellman-Ford algorithm for shortest path computation in order to achieve the maximum-sized vertex frontier at each step, enabling increased parallelism. Further, the frontier relaxations are achieved using sparse matrix-matrix multiplication. The proposed algorithm also reduces the communication by a factor of $\sqrt[3]{p}$ on p processors. Hoang *et al.* (2019) propose a round-efficient distributed BC algorithm in the CONGEST model. Their proposal reduces the number of messages sent across edges. Further, they reduce the number of rounds by a factor of $14\times$ over existing works on 256 hosts.

3.2 Approximate Graph Processing

Approximate graph computation has been well explored from the standpoint of both theory and applications.

Mittal (2016) presents a survey of various approximate computing strategies including *precision scaling*, *loop perforation*, *memoization*, *selective memory accesses*, *data sampling*, *voltage scaling*, *inexact reads/writes*, *lossy compression* and *using universal function approximators* in various domains for improving performance and reducing the energy requirements in exchange for acceptable loss in output quality. Our tech-

niques for parallel approximate graph processing presented in the following chapters may be categorized into loop perforation, selective memory accesses, memoization, data sampling and lossy compression.

Yazdanbakhsh *et al.* (2017) present AxBench, which is a benchmark suite for evaluating various approximate computing techniques using diverse applications running on CPUs and GPUs. Each of the benchmarks contains annotations for the "approximable regions", which are regions of code that consume the most time or energy during execution, or which are error-tolerant. Further, AxBench contains application-specific quality metrics to measure and compare the efficacy of the different approximation techniques.

Shang and Yu (2014) propose a system to auto-synthesize the approximate versions of iterative vertex-centric graph algorithms. They combine multiple approximate computing techniques including task-skipping, sampling, memorization and interpolation for generating the approximate version of the algorithm. In contrast the approximate computing techniques discussed in the thesis provide tunable knobs to the user to control the performance-accuracy tradeoff. Further, each of the techniques has been considered in isolation.

3.2.1 Graph sampling based techniques

Several sampling-based approaches have been proposed for approximate graph computation (Goldberg and Harrelson (2005); Leskovec and Faloutsos (2006); Wang *et al.* (2011); Turk and Turkoglu (2019)). Most of these are sequential.

Benczúr and Karger (1996) propose non-uniform sampling of graph edges to obtain a cut-sparsifier having the same number of vertices as the original graph. The compressed graph is built by including an edge e with probability p_e and assigning it a weight of $\frac{1}{p_e}$ if it is included.

Gubichev *et al.* (2010) present a preprocessing-based technique for finding the approximate single source shortest path from a designated source node for a given graph. The precomputation step involves sampling a set of nodes and computing for every node in the graph, a shortest path to and from few *landmark* nodes in the sampled set of nodes. The set of paths so obtained are stored in external memory. The precomputed

information is used to provide a fast approximation of the node distance at query time. It works by combining the distance of the query nodes, s and d to or from a selected landmark node l into the approximate distance $\tilde{d}(s,d)$. They implement a few other optimizations – Cycle Elimination, Shortcutting, on top of the basic scheme to further reduce the query time.

A subset of the techniques presented in Chapter 4 and the techniques discussed in Chapters 5 involve preprocessing the input graph. However, during preprocessing, we do not precompute information for online processing, but transform the original graph.

Several approximation techniques developed for approximate BC computation have graph-sampling at their core. Bader *et al.* (2007) propose an adaptive sampling based approximation algorithm for accurate estimation of the BC scores of vertices with a high probability. Their approach reduces the number of single-source shortest path computations for vertices with high BC. Haghir Chehreghani (2013) proposes a generic randomized framework that relies on sampling source vertices for unbiased approximation of vertex BC to achieve high efficiency and accuracy. The focus of this work is to approximate betweenness centrality of a subset of vertices faster than computing betweenness centrality of all vertices. Mumtaz and Wang (2017) propose an approximate algorithm for BC maximization problem. They devise an estimation technique based on progressive sampling with early stopping conditions to get better accuracy with smaller sample sizes.

Riondato and Upfal (2018) propose progressive sampling schemes based on Rademacher averages to compute high-quality approximation of the vertex BC values for static and dynamic graphs. They extend their arguments to compute an approximation of top-k vertices with nice probabilistic guarantees. We compare our techniques for estimation of top-k BC vertices against their work in Chapter 6. Borassi and Natale (2019) present KADABRA, which is an improvement over the existing sampling based schemes for computing approximate vertex BC scores and top-k vertices. They devise an adaptive sampling scheme that takes into account the stochastic dependence between the runtime of the algorithm and the approximation quality. Their scheme results in reduced sample sizes and also reaches the stopping condition early. Further, they use bidirectional BFS to sample the pairwise shortest paths between vertices. van der Grin-

ten and Meyerhenke (2019) propose a parallel version of the KADABRA algorithm in a distributed setup. Specifically, they parallelize the adaptive sampling phase.

The shemes for estimating the top-k BC vertices presented in Chapter 6 determine the order in which to select the source vertices in Brandes' algorithm to enable quicker identification of top-k nodes. Further, our work applies these approximate techniques in the context of GPUs.

3.2.2 Graph compression based techniques

Various graph compression techniques including spanners, sparsifiers, graph sketching and graph summarization have been extensively studied for approximate graph processing and storage (Spielman and Teng (2011); Ahn *et al.* (2012); Guha and McGregor (2012); Koutis and Xu (2016); Shin *et al.* (2019)).

Bandyopadhyay *et al.* (2016) propose a graph sketching technique using minwise hashing to sample neighborhood edges. They devise a mechanism to construct the sketch incrementally. Their scheme preserves important topological properties of the graph, such as clustering coefficient and pagerank.

Spielman and Teng (2004) construct spanners by performing a natural random rounding of the graph to achieve a good approximation of the original graph.

Besta *et al.* (2019) present *SlimGraph*, a framework for lossy graph compression. The framework enables users to define simple *compression kernels* based on the local view of the graph. SlimGraph implements a large set of popular compression schemes by application of these kernels. The compression proceeds by removing vertices, edges or triangles from the graph until the compressed graph having a desired quality is obtained. They propose several metrics to evaluate the quality of compression and the impact of compression on the algorithm's outcome.

In contrast, our technique for lossy graph compression in Chapter 4 compresses the graph by successively merging similar nodes. We use Jaccard's coefficient to determine the nodes to be merged.

CHAPTER 4

Parallel Approximate Graph Processing

Graph analytics algorithms have been shown to bear sufficient parallelism. Parallelizing graph algorithms is challenging due to the irregular memory accesses involved in graph traversals. The parallelization issues get exacerbated on GPUs.

We attempt to address the scalability issues in parallel graph algorithms — improving the execution performance of exact parallel graph processing, w.r.t. performance, by trading off computational accuracy, using approximate computing. Our focus is *propagation-based* algorithms (which is a large subset), following the *vertex-centric* model of parallelization, wherein the threads operating on the assigned set of nodes propagate information to the nodes' neighbors along their incident edges. A typical implementation of a graph algorithm on GPU has the following structure:

```
Graph G(V,E) = read_input(); // CPU
transfer_input(); // CPU → GPU
do { // outermost loop
  changed = false;
  for all Node u:V do {
    for Node v:G.neighbors(u) do {
      if(condition) {
        // Update attributes of v
        changed = true;
      }
    }
  }
} while(changed);
transfer_output(); // GPU → CPU
```

Listing 4.1: General Structure of a Graph Kernel

We target adding controlled approximations to graph algorithms to improve their efficiency. Thus, instead of computing the exact answer, algorithms perform less work to calculate only an approximate solution. Towards this goal, we propose four

Domain $\mathcal{D}$	Mapping function $\mathcal{F}$	Examples
Iterations	iteration $> K \implies void$ truncate iterations based on an error threshold ...	reduced execution (Section 4.2.2) more approximate computation ...
Graph processing	process partial graph process top K graph elements conditional processing ...	truncate computation to K vertices or edges (Section 4.2.3) sort based on a criterion such as degree and select top K vertices process only those graph elements that satisfy a condition ...
Graph representation	lossy graph compression maximum degree K clustering reduced storage ...	merge vertices based on neighborhood similarity (Section 4.2.4) truncate graph beyond degree K per vertex merge nearby vertices store graph in a smaller adjacency matrix / lists ...
Attribute values (both vertex and edge)	integer division by K round-off to the nearest power of 2 modulus by K hashing ...	consecutive K entries similar mapped to powers of 2 (Section 4.2.5) round-robin mapping from 0 to K-1 similarity based on the hash function ...
...	...	...

Table 4.1: Instantiation of the approximation model with various values of $\mathcal{D}$ and $\mathcal{F}$

architecture- and algorithm-agnostic techniques for parallel graph processing that exploit the general structure of a graph kernel (Listing 4.1) and the flow of information in graph algorithms to arrive at an approximate solution early. These techniques target various parts of computation and data. While many such heuristics-based techniques are feasible (Table 4.1), we study two techniques directed towards computation and two techniques directed towards data. Our investigation reveals that each of these techniques is quite beneficial in improving the execution time, at the cost of accuracy. Interestingly, a user can control the exhibited inaccuracy by controlling the injected approximation.

This chapter is organized as follows. In Section 4.1, we devise a theoretical model of approximation and illustrate its generality by instantiating it with different kinds of approximations. In Section 4.2, we propose Graprox, a host of techniques for executing graph algorithms on GPUs in an approximate manner. In particular, our techniques perform reduced execution, process only part of the graph, store graph in an approximate manner, and approximate attribute values to gain in efficiency. We next discuss how approximations can be exploited for an efficient GPU-based parallel processing in Section 4.3. In Section 4.4, we evaluate our proposed techniques and show that they work quite well in practice compared to the exact versions. Parameterized solutions provide tunable knobs to change the degree of approximation. Using five large graphs and six graph algorithms, we illustrate that approximate versions offer considerable performance benefits with a small accuracy-loss.

4.1 Approximation Model

We define a theoretical model to characterize approximations. By instantiating this model with various parameters, we obtain different approximation techniques.

An approximation $\mathcal{A}$ ($\mathcal{D}$, $\mathcal{F}$) is defined over domain $\mathcal{D}$ of entities, where function $\mathcal{F}$: entity $\rightarrow$ entity is used to approximate the entities. For instance, $\mathcal{A}$ (AttrVal, Div1024) where Div1024(attrval) :- (attrval / 1024) approximates attribute values by making consecutive 1024 values non-distinguishable. In other words, $\mathcal{F}$ *maps* entities in $\mathcal{D}$ to a subset of entities in $\mathcal{D}$. Thus, $\mathcal{F}$, in general, is a many-to-one function, which provides the necessary approximation.

The mapping function $\mathcal{F}$ can be arbitrary, ranging from identity function (indicating no approximation) to constant function (indicating maximum approximation). A judicious selection of $\mathcal{F}$ can help improve algorithmic performance (both time and space) without losing much precision.

4.1.1 Function Application Order

The mapping function $\mathcal{F}$ is type-preserving, that is, it maps an entity in the domain to another entity in the same domain. Therefore, $\mathcal{F}$ application can be cascaded: $\mathcal{F}.\mathcal{F}.\mathcal{F}.x$, leading to a chain of approximations. However, as long as $\mathcal{F}$ is deterministic, the ordering in which various function applications are performed does not affect the outcome. For instance, computing minimum among a set of values does not depend upon the order. Such a property is crucial in a parallel setting where various thread orderings lead to the same approximate entity for the original domain entity. On the other hand, a non-deterministic $\mathcal{F}$ may affect the outcome for a different thread-schedule; e.g., merging nodes based on their neighborhood similarity. We experiment with both kinds of approximation functions in this study.

4.1.2 Idempotent Approximation

Non-determinism in thread-scheduling may result in a different number of approximate function applications. This non-determinism may, in general, lead to different amounts of approximations added to the processing across different executions of the same program. Such behavior may be unacceptable as it means different outputs across different runs of the same program over the same input. Need for such a determinism necessitates $\mathcal{F}$ to be an idempotent function. Thus, multiple applications of $\mathcal{F}$ should be equivalent to a single application. Although expecting the mapping function to be idempotent may sound restrictive, in practice, most of the standard approximation techniques are indeed idempotent. For instance, mapping an edge weight (say 23) to the nearest power of two (32) is an idempotent approximation, as a remapping (on 32) would maintain the value (as 32). All the approximation techniques we propose are idempotent. This allows us to faithfully assess the effect of approximations compared to the exact processing.

4.1.3 Approximation Structure

We define a relation $\mathcal{R}$ between a pair of entities induced by the approximation function $\mathcal{F}$. Thus, $x_1 \mathcal{R} x_2$ iff $\mathcal{F}(x_1) = \mathcal{F}(x_2)$. $\mathcal{R}$ is a reflexive ($x \mathcal{R} x$), symmetric ($x \mathcal{R} y \implies y \mathcal{R} x$) and transitive ($x \mathcal{R} y$ and $y \mathcal{R} z \implies x \mathcal{R} z$), forming an equivalence relation. Thus, $\mathcal{R}$ partitions the domain $\mathcal{D}$.

At one extreme, when the approximation function is an identity function, each element in the domain is in a separate partition, say with cardinality N. At the other extreme, when the approximation function is a constant function, all the elements are in the same partition with cardinality 1. In general, various approximation functions form K partitions with $1 \leq K \leq N$, leading to different precision values.

4.2 Approximating Graph Algorithms

We instantiate the approximation model with various values of $\mathcal{D}$, and accordingly, multiple values of $\mathcal{F}$. Figure 4.1 presents such approximations. The number of instan-

tiations can be huge; we pick one interesting approximation technique for four domains and explore it in depth.

4.2.1 Graph Algorithms

We work with a variety of algorithms: Single Source Shortest Paths (SSSP), Minimum Spanning Tree (MST), Betweenness Centrality (BC) PageRank (PR), Strongly Connected Components (SCC), and Vertex Coloring (Color). The details of these algorithms are described in Section 2.4. The input to each graph algorithm is a directed graph. The graph-edges have weights in case of SSSP, MST.

4.2.2 Technique 1: Reduced Execution

In the reduced execution technique, we cut-short the execution to compute an approximate solution. Graph algorithms are often iterative. We exploit this fact to add approximation to the total amount of work done in terms of the number of iterations. That is, we execute the main processing loop (outermost if there are nested loops) for fewer iterations (compared to the corresponding exact version) with the hope of improving performance. Less overall work forbids the algorithm from reaching the fixed-point or the correct solution. For instance, consider single-source shortest paths (SSSP) computation shown in Algorithm 2. The outer `while` loop at Line 5 is cut-short. Reduced execution approximation is useful for algorithms where a large amount of work gets done in the initial iterations. One way to implement this approximation is by configuring a percentage threshold on the number of loop iterations. This is feasible as long as the loop executes a fixed number of iterations (such as Prim's minimum spanning tree algorithm, Brandes' betweenness centrality computation, and Bellman-Ford shortest paths processing). In general, a more effective way is to provide an inaccuracy-tolerance, and the implementation chooses the maximum possible number of iterations respecting the inaccuracy limit. The inaccuracy-tolerance refers to the amount of inaccuracy permitted by an application, and varies with application. However, we also note that there may exist computations wherein inaccuracy cannot be calculated without computing the exact solution. Reduced execution can be applied in such scenarios too, but without any

guarantees on the approximation.

Our experimental evaluation shows that a small decrease in the outer loop iterations achieves good benefits in execution time at the cost of small loss in accuracy. For example, we find that for SSSP, reducing the outer loop iterations to 90% achieves an average speedup of 1.6×, with an inaccuracy of up to 7%. However, the inaccuracy increases rapidly as we further reduce the number of iterations. For instance, reducing the loop iterations to 65% of the exact answer results in a performance gain of around 1.7× at the cost of 21% accuracy loss. We also found that SSSP is a good candidate for reduced execution as most of the distances get settled within about 50% of the iterations (Figure 4.3). In contrast, Color exhibits a much higher inaccuracy (29.18%) for a modest (45%) performance improvement.

4.2.3 Technique 2: Partial Graph Processing

Our next proposal is to process only part of the graph, to improve execution time. Not all parts of the graph contribute equally to the final fixed-point information. Ideally, we would like to process only the highest-contributing parts – to reduce execution time to the minimal, incurring minimal inaccuracy. However, such information is often not efficiently computable, and, in fact, changes across iterations. Therefore, we would need to depend upon heuristics to choose the part of the graph to be processed. Thus, based on criteria, for each pass through the graph, or for each iteration of the outermost loop, we choose to selectively process only a subset of the vertices/edges.

Partial graph processing resembles performing a random walk on the graph and is performed as below. For one pass through the graph, for each node we choose to process, we assign special values to its outgoing edges. The values are drawn uniformly at random from the set of non-negative integers $\in [0, m)$, where m is the number of edges. In other words, we generate one of the permutations of the edge identifiers. From among these values, we traverse only the highest few (say top 50% or 60%). The other edges are omitted, and the nodes on which the omitted edges are incident may not be processed in that iteration. An iteration is said to be complete when all the threads have completed their share of work. The work of each thread is to process

the (selected) edges for the nodes assigned to it, once. We stop when the change in a parameter (dependent on the algorithm) across two successive iterations is *small*.

In a nutshell, we select a subset of edges to process, at every node. The number of edges selected uniformly at random at a node is a function of the node degree. So, the technique takes into consideration the topology of the graph. The technique could be refined and improved by considering other local and global characteristics of the graph.

For instance, in SSSP as shown in Algorithm 2, partial graph processing would change the `for` loop at Line 7 to go over a subset of vertices. We may fix the same edge-permutation for every iteration. But it leads to high deviation from the exact output. Therefore, we propose generating a new permutation of edge identifiers in every iteration. We also experimented with selectively skipping the same outgoing edges for the vertices we process. Such a scheme considerably reduces precision. Note that our method does not even traverse the edges not selected. In some graph algorithms (such as finding the vertex with the maximum-degree) where the processing loop enumerates through vertices or edges, the two approximations, namely, reduced execution and partial graph processing may overlap.

Partial processing allows us to reduce the total number of graph operations compared to the exact version. This also reduces the amount of synchronization required in processing the graph. For instance, in SSSP computation, the number of atomics reduces due to fewer vertices being processed. Since atomics on GPUs are costlier than regular reads and writes, this leads to better execution time.

Our experiments show that if we process only a fraction of the graph without modifying the edge/vertex attributes, the inaccuracy grows fairly quickly as we reduce the fraction of the graph processed. For instance, in SSSP, if we process 75% of the edges, it achieves around 1.2$\times$ speedup, with 21% inaccuracy. However, when we process only 25% of the edges, it achieves a speedup of around 2.8$\times$ but the inaccuracy shoots-up to 63%. On the other hand, if we assign values edge/vertex attributes carefully, then the error grows gradually even with the processing of a small fraction of the original graph. In case of SSSP, when we assign edge weights after preprocessing the graph before applying this approximation, we observe that processing 75% of the graph causes the answer to deviate from the exact value by 18% on an average. Also, with 25% of

the graph processed, the inaccuracy is close to 24% and does not increase drastically.

4.2.4 Technique 3: Approximate Graph Representation

Reduced iteration and partial graph processing discussed in the last subsections work with the original (exact) graph. In the approximate graph representation technique, the graph itself is stored in an imprecise manner. Thus, instead of working on the exact graph representation, the (exact) algorithm runs on graph's approximation. There are multiple ways to implement this. One way is to assign the same vertex-id to multiple vertices. Alternatively, we can store the graph in a probabilistic data structure (such as bloom filters). In this work, we explore vertex-merging, which involves logically merging the adjacency lists (both incoming and outgoing) of the vertices being merged. Merging leads to a smaller graph containing fewer vertices (and edges), which reduces the execution time. If there is a triangle a-b-c and a-b get merged, then we assign the weight to the edge ab–c as the mean of the weights of the edges a-c and b-c.

The merging can, in general, be performed on an arbitrary pair of vertices. But it reduces inaccuracy if performed carefully. We enable vertex-merging for a pair of vertices if their neighborhoods are *similar*. Two vertices have similar neighborhoods if their Jaccard's coefficient is above a threshold. Jaccard's coefficient J_{ij}, for vertices v_i and v_j with sets of neighbors $N(v_i)$ and $N(v_j)$ respectively, is:

$$J_{ij} = \frac{|N(v_i) \cap N(v_j)|}{|N(v_i) \cup N(v_j)|} \tag{4.1}$$

As vertices get merged, they form a meta-vertex, which, in turn, may get merged with another vertex or meta-vertex, and so on. v_i or v_j in Equation 4.1 may represent an original vertex or a meta-vertex. The merging order is important to the quality of the approximate representation (see Section 4.1.1). We merge the vertices using a greedy heuristic, prioritizing merging vertices with higher degrees. We observed that the nodes with high degrees tend to have a larger overlap of neighbors. As a result, merging such nodes helps in reducing the size of the graph (in terms of number of edges) faster since we ensure that compressed graph does not have parallel edges throughout the

compression process. Further, we could merge such nodes when the threshold is set to a high value (say 0.8) which leads to smaller inaccuracies.

Node-merging necessitates logical merging of the incoming and the outgoing edges of the vertices being merged. That is, neighbors of the vertices become neighbors of the merged vertex (removing self-loops if there was an edge between the vertices being merged).

The minimized graph thus obtained is fed as an input to the exact version of the algorithm. In our experiments, we find that decreasing the merging threshold of the Jaccard coefficient increases the inaccuracy. Decreasing the threshold also increases the speedups we obtain, in most cases. This is because decreasing the threshold decreases the number of vertices in the minimized graph, though the number of outgoing edges for a vertex may increase. The choice of Jaccard's threshold is governed by the desired compression factor, which in turn has a bearing on the inaccuracy injected. We need to choose the Jaccard's threshold judiciously so as to achieve good performance benefits while keeping within the acceptable limits of inaccuracy compared to the exact version. The appropriate threshold needs to be determined empirically.

4.2.5 Technique 4: Approximate Attribute Values

Our fourth proposal is to reduce the computation cost of large graphs by approximating attribute values of graph elements. Numeric attribute values (such as vertex distances or edge weights) can be rounded-off to discrete values, say powers of 2. This rounding-off of the attribute values enables reaching the termination criteria faster, in fewer rounds. Non-numeric values can be changed to be chosen from a smaller domain (e.g., vertex colors). We discuss below applying such a discretization for SSSP.

In SSSP, discretization is a two-step process. In the first step, we perform a traversal through the edges to find the maximum and the minimum weight edges. Let the maximum and the minimum edge-weights be w_{max} and w_{min} respectively. In the second step, we perform another traversal on the edges to modify their edge-weights. w_{min} is rounded-up to its nearest power of 2 and w_{max} is rounded-down to the nearest power of 2. We call these new limits as w'_{min} and w'_{max} respectively. All the edge weights

are rounded to their nearest power of 2 in the range $[w'_{min}, w'_{max}]$. With the above modification, we are guaranteed that any edge in the graph can take only one of the $k \triangleq \{log_2(w'_{max}) - log_2(w'_{min})\}$ values. Assuming each edge takes any of these values with equal likelihood, an edge has an expected weight calculated as below.

Let X be a random variable which is defined as the weight assigned to an edge $e \in E$. X can take values from the set $S = \{w'_{min}, \ldots, w'_{max}\}$. An edge can be assigned any of these values with probability $\frac{1}{k}$. So,

$$\begin{aligned} E[X] &= \sum_{x \in S} x.Pr(X = x) \\ &= \sum_{x \in S} x.\frac{1}{k} \\ &= \frac{2 \times w_{max} - w_{min}}{k} \end{aligned}$$

So in expectation, the maximum distance between any two vertices can be $(graph\ diameter) \times \frac{2 \times w_{max} - w_{min}}{k}$.

We can make an informed choice about initial attribute values which would aid in reducing the portion of the graph we process. For instance, in case of SSSP computation, we can initialize the distance from the source to every vertex to some value other than the customary ∞. Such a value is computed as a preprocessing step as follows. We run a single pass of the Breadth-First-Search (BFS) on the graph starting at the source vertex s. This gives us the hop distance of every vertex from the source vertex. During the traversal, we also find the largest edge-weight value. Now, we set the initial distance of every vertex as: $v \in V$, $dist(s, v) = (\#\ of\ hops\ from\ s\ to\ v) \times (max\ edge\ weight)$.

Approximation of edge-weights, with a careful extra preprocessing, can enable us to transform the SSSP computation into an easily parallelizable BFS. As a preprocessing step, we run BFS on the given graph, from the source vertex s to get the level information of all the vertices with respect to s, in the form of the BFS tree rooted at s. In this BFS tree, we compute a weighted mean of the weights of the edges from level i to $i + 1$, where $i \in \{0, 1, 2, \ldots\}$. The weights are drawn from a uniform distribution with values in the range (0, 1). We assign this weighted mean to all the edges from level i to $i + 1$ and do the same for all the levels. After assigning the new edge-weights to

all the edges in the BFS tree, a traversal of the tree gives the approximate shortest path distance of every vertex from the source node. For computing the weighted mean, the weight assigned to an edge-value is inversely proportional to it, i.e., higher edge-value is multiplied by a lower weight. This is done so that the weighted mean is not skewed towards the higher edge-values and is only slightly away from the exact shortest path length. This approximate technique makes it feasible to achieve good speedups of the iterative SSSP computation with plausible error bounds.

Similarly, for the PageRank algorithm, we initialize the pageranks of all the vertices to $\frac{1}{n}$ and not to some arbitrary discrete value. This serves two purposes. First, $\frac{1}{n}$ implies that the surfer lands on each page with equal probability. Secondly, since the PageRank computation essentially gives an estimate of the likelihood that the surfer lands on each page, the value does not vary drastically from the initial value. Hence, even here, we can afford to process the graph only partially and still have a reasonable solution. It has the effect that in large graphs every unprocessed vertex can be reached with a fairly low probability.

In case of MST, we round-up or -down the edge-weight to the closest power of 2. We stop when the weight of the MST overshoots a threshold set to $(n-1) \times (mean\ weight)$ rounded to the nearest power of 2, where n is the number of nodes in the graph. We find that this scheme leads to better execution time compared to the exact version.

In BC, the betweenness centrality value of each vertex is in the range [0..1]. We subdivide this range into 10 equal-sized buckets, and the centrality value of each vertex is rounded to the nearest tenth.

4.3 Benefits to GPU-based Processing

Though the proposed approximation techniques can be applied independent of the architecture, employing them on a parallel GPU code is particularly useful since they address important performance bottlenecks. Such bottlenecks are artifacts of issues such as synchronization, workload-imbalance, CPU-GPU data transfer, etc. We discuss how our approximation techniques help in diminishing their effect.

4.3.1 Technique 1: Reduced Execution

Typically, a graph algorithm gets modeled in a bulk-synchronous fashion where the host code repeatedly calls the processing kernels. Such processing involves an implicit barrier at the end of each kernel invocation (e.g., Line 16 in Algorithm 2). The approximation technique of reducing the number of iterations of the algorithm reduces the number of barriers invoked. For algorithms that require a large number of iterations, the cumulative effect of reducing the number of barriers helps improve performance.

We also observe that in typical algorithms, the amount of work done in later iterations is relatively much lesser (see, for example, Figures 4.3, 4.6). Especially in the context of massive-multithreading, such a behavior reduces the parallel work-efficiency. Reduced execution mitigates such an effect, and improves average work efficiency.

4.3.2 Technique 2: Processing Part of the Graph

In case of partial processing, the algorithm operates on only a subset of the graph. Processing fewer edges translates to lesser synchronization in each iteration. For instance, in SSSP computation of Algorithm 2, which uses an atomic operation at Line 11, reducing the number of processed edges implies that the number of incoming edges to a node reduces, thereby reducing the number of atomic operations.

Partial processing also helps partially address the problem of unbalanced work distribution among threads in a vertex-centric GPU implementation. Load imbalance happens due to a few vertices having large outdegrees (as in social networks). Due to the approximation of partial processing, however, the number of edges processed by each thread is lesser, mitigating the effect of load imbalance.

4.3.3 Technique 3: Approximate Representation

One of the primary bottlenecks in CPU–GPU systems is the inter-device data transfer over a relatively slow PCIe interconnect, especially for large graphs. Thus, it is desirable that the number of CPU–GPU transfers be reduced and the amount of data being sent be

also small. With approximate graph representation, the device-to-device data transfer reduces, leading to performance benefits.

4.3.4 Technique 4: Approximate Attributes

Approximating attribute values helps in achieving the fixed-point in fewer iterations, leading to reduced synchronization in terms of implicit barriers. In case of algorithms such as SSSP, we approximate the vertex attributes to get rid of the atomic operations. We initialize vertex distances using a BFS-based approximation, which enables us to transform the SSSP computation to an easily parallelizable level-by-level BFS processing. Since level-synchronous BFS can be implemented without explicit atomic instructions (Nasre *et al.* (2013*b*)), avoiding the synchronization improves performance.

In case of MST computation, which often requires several iterations to converge, we devise a policy for it to converge faster. MST's parallelism profile suggests that it has a good amount of parallelism initially, which reduces as the algorithm progresses. Online approximation of attributes, across iterations, helps us terminate the algorithm early. By rounding the edge attributes to powers of two and setting a suitable threshold (Section 4.2.5), the algorithm makes rapid strides and converges faster.

4.4 Experimental Evaluation

In this section we evaluate the performance of the various approximation techniques, and compare it with the exact versions of the respective algorithms.

Machine Configuration. All our experiments are carried out on an Intel Xeon 32-core E5-2650 v2 CPU @ 2.6GHz having 96 GB RAM running CentOS 6.5. We use Nvidia Kepler (Tesla K40C) GPU having 2880 cores spread across 15 SMXs with 12 GB memory. We use CUDA 8.0 to compile and execute our methods on the GPU.

Input Graphs. We select graphs with varying characteristics to test the efficacy of our proposed techniques. The graphs include R-MAT graphs, Erdős-Rényi graphs, generated by GTgraph (Madduri and Bader (2006)); small-diameter social networks; and

Graph	$\|V\|$ $\times 10^6$	$\|E\|$ $\times 10^6$	Graph type
rmat26	67.1	1073.7	R-MAT graph using GTgraph (Madduri and Bader (2006))
random26	67.1	1073.7	Random graph using GTgraph
LiveJournal	4.8	68.9	Social network, small diameter
USA-road	23.9	57.7	Road network, large diameter
twitter	41.6	1468.3	Twitter graph 2010 snapshot

Table 4.2: Input graphs

Graph	**Exact Time (sec)**					
	SSSP	MST	SCC	Color	PR	BC
rmat26	37	8996	21	14	12	15223
random26	29	10087	23	18	16	13127
LiveJournal	2	3424	7	5	1	1711
USA-road	152	82	12	10	1	2043
twitter	231	10943	37	29	18	21462

Table 4.3: Execution time for exact versions of graph algorithms

large-diameter real-world road networks, from SNAP (Leskovec and Sosič (2014)) and KONECT (Kunegis (2017)). Table 4.2 lists the input graphs for Graprox.

Graph Algorithms. We study six graph algorithms: single-source shortest paths computation (SSSP), minimum spanning tree computation (MST), finding strongly connected components (SCC), vertex coloring (Color), page rank (PR) and node betweenness centrality computation (BC). We compare our approximate SSSP and approximate MST with the respective exact versions from LonestarGPU (Burtscher *et al.* (2012)), approximate SCC with the exact SCC by Devshatwar *et al.* (Devshatwar *et al.* (2016)), Color with our parallel implementation of exact largest-degree-first (LDF) coloring algorithm, and approximate PR with our parallel implementation of PR, and BC with our parallel implementation of Brandes' algorithm. We run the PR algorithm for 10 outerloop iterations, while for BC we run the algorithm for 10000 outerloop iterations (i.e., from 10000 sources). The chosen baseline implementations follow a vertex centric model of parallelization and use the compressed sparse row representation (CSR) of the graph in memory, which are commonly encountered.

Table 4.3 reports the execution times of the exact parallel versions of these graph algorithms. While other techniques do not require any preprocessing, approximate graph representation needs to compute neighborhood similarities to calculate Jaccard's coefficient. Since this is a one-time cost, we do not account for this preprocessing in the execution time.

An important aspect of measuring the effectiveness of approximations is to compare

the accuracy of the computed values. This can be achieved by computing an absolute difference between the attribute values of the vertices for the exact and the approximate versions, and taking an average across vertices for a run. For multiple runs (say, across graphs), we compute the geomean difference over the averages for each run. For SSSP, the attribute is the distance value; for PR, it is the page rank value; and for BC, it is the betweenness centrality value. For SCC, we calculate the difference in the number of SCCs computed by the exact and the approximate methods. For MST, we calculate the difference in the weight of the minimum spanning tree computed by the two methods. Such a mechanism does not work for discrete values such as colors in vertex coloring. A straightforward solution is to use number of colors as a measure to compare. However, due to the non-determinism in thread-scheduling, multiple runs of our coloring algorithm may result in different colorings; leading to a difference in the number of colors used for the same graph. In our implementation of LDF algorithm, this happens when one or more neighbors of a vertex have the same degree. Therefore, the baseline accuracy of such an exact version cannot be faithfully captured in the parallel setting. To address this, we measure the accuracy as the percentage of *pairs* of adjacent nodes having different colors. Such a quantity indicates the degree of closeness with the exact coloring (which would have this value as 100%), and importantly, it will be independent of the thread-scheduling.

4.4.1 Overall Results

Table 4.4 summarizes the effects of the four approximation techniques for the six graph algorithms. We list the geomean speedup and the inaccuracy values across all the graphs in our setup. The approximation of the attribute values is inapplicable for PR, SCC and Color; hence the entries are marked as –. Overall, we observe that the approximations have the capability to achieve high speedups by trading off more and more accuracy. However, some algorithms seem to be more amenable to effective approximation than others. For instance, approximate SSSP and PR achieve relatively higher speedups for lower inaccuracy compared to MST, SCC, Color and BC. This indicates that continuous-value-based algorithms (such as SSSP and PR) provide better approximation opportunities than discrete-value-based (such as Color). Second, algorithms

Algo.	Technique	Mean Speedup	Mean Inaccuracy
SSSP	Outer-loop iterations	1.49	6.34%
	Partial processing of graph	1.47	17.82%
	Approx. graph representation	1.27	14.37%
	Approx. attributes	1.92	17.64%
MST	Outer-loop iterations	1.22	14.08%
	Partial processing of graph	1.74	17.23%
	Approx. graph representation	1.56	15.5%
	Approx. attributes	1.48	19.07%
SCC	Outer-loop iterations	1.26	16.48%
	Partial processing of graph	1.32	19.50%
	Approx. graph representation	1.45	21.5%
	Approx. attributes	–	–
Color	Outer-loop iterations	1.45	**29.18%**
	Partial processing of graph	1.28	16.43%
	Approx. graph representation	1.36	18.39%
	Approx. attributes	–	–
PR	Outer-loop iterations	2.03	**2.45%**
	Partial processing of graph	1.82	12.76%
	Approx. graph representation	1.54	13.63%
	Approx. attributes	–	–
BC	Outer-loop iterations	1.74	18.07%
	Partial processing of graph	1.42	16.73%
	Approx. graph representation	1.33	14.35%
	Approx. attributes	1.41	23.16%

Table 4.4: Overall results

that depend heavily on the graph structure (such as SCC and Color) have a relatively higher inaccuracy. This is an artifact of values getting refined in each iteration, but structures are often binary (either an SCC or not, either a neighbor or not), which affects approximation opportunities. High inaccuracy for outerloop iterations in Color is because reduced execution for Color translates to using fewer colors.

We now look at the overall effect of individual approximations; detailed discussion follows in the subsequent sections. First, the effect of reduced execution follows value-based approximations – more effective for SSSP and PR (with lower inaccuracy and higher speedup) and less so for MST, SCC and Color. BC gets benefitted in execution time (1.74×), but at the cost of high inaccuracy (18.07%), due to BFS from fewer source vertices. However, PR appears to be exceptionally benefitted by this approximation, as the page rank values converge rapidly to their final values in a few iterations. Thus, for algorithms where fixed-points are approached quickly and then refined slowly, reducing execution turns out to be very useful. Therefore, it is a useful approximation for gradient-descent kind of algorithms.

Second, partial graph processing is uniformly useful across algorithms, with high speedups, but the usefulness is offset by a higher inaccuracy. This is an indication of algorithms working on graph properties that are *global*, and get affected by most of the

graph elements. For instance, removing some edges of the graph would affect shortest path or page rank value propagations. Partial graph processing is more beneficial for algorithms that compute *local* properties, such as computing the maximum clique or minimum spanning tree. In such problems, removal of a few edges or nodes would have a reduced probability of affecting the max-clique or MST, leading to the approximation being more effective. We observe such behavior for MST wherein the performance of the approximate version with partial graph processing is particularly higher (speedup of $1.74\times$) compared to other techniques (speedup $\leq 1.5\times$). For BC, partial processing achieves moderate benefits.

Third, approximate graph representation (using Jaccard's similarity) is consistently beneficial for performance, with relatively better accuracy (compared to other techniques). This is understandable for structure-based algorithms such as SCC. Even for Color, since two vertices with almost common neighborhood can be given the same color, merging them is likely to maintain accuracy. A similar propagation effect happens in case of PR – common neighbors propagate common values across vertices – hence merging the nodes does not adversely affect accuracy. However, such a merging approximation is unlikely to be useful for SSSP where edge-weights play a major role despite the common neighborhood. Therefore, we observe reduced benefits due to this approximation for SSSP (speedup of $1.27\times$), compared to other techniques (speedup $>$ $1.3\times$). For BC, setting the Jaccard's similarity threshold for merger moderately affects the speedup. A lower threshold aggressively merges the vertices but, in turn, increases the number of neighbors of the meta-vertex. On the other hand, setting a high Jaccard's similarity threshold merges fewer vertices.

Finally, approximating values is applicable for weighted algorithms such as SSSP and MST. We have also applied it to unweighted BC, approximating the vertex attribute, that is, the BC value. While this approximation achieves good performance for both the weighted algorithms, the inaccuracy is higher for MST (19.07%) compared to SSSP (17.64%). This happens because of the algorithm's behavior – MST is implemented using Boruvka's algorithm which merges components based on the lightest inter-component edge. Thus, the number of choices for inter-component edge increases after the power-of-2 approximation – which can lead to a different edge getting selected.

An accumulation of errors across multiple iterations leads to increased inaccuracy for MST. A similar effect happens in SSSP too, but since the effect is restricted to choosing the minimum distance across neighbors (rather than across a collection of vertices), the effect is small, leading to better accuracy. For BC, we observe that the decent speedup of 1.41× is accompanied by a high inaccuracy of 23.16%. This is primarily due to the per-iteration approximation of BC values. To be specific, we discretize the BC values after each iteration to the nearest tenth. The algorithm halts when the difference in the BC values across iterations is less than a threshold.

We discuss the effect of each technique in more detail.

4.4.2 Effect of Reduced Execution

Figure 4.1 presents the effect of the reduced execution technique (Section 4.2.2) on the six algorithms. To avoid clutter, we show results for three largest graphs. We observe that by trading off some accuracy, one may enjoy considerable performance benefits. From the shape of the plots, we see that algorithms which compute global properties such as MST and SCC, the inaccuracy almost linearly follows the added approximation. However, SSSP and BC depend upon a source vertex and perform the computation based on it. Such algorithms are more sensitive to reduced execution – they can achieve high speed-up with high inaccuracy. USA-road is a notable exception – the performance benefits due to reduced execution approximation is relatively low. This is an artifact of structural properties of the road networks. In particular, road networks have large diameters and uniform degree distribution. This is in contrast to other networks that follow *small-world* property and have power-law degree distribution. PR benefits substantially by the approximation on outer loop iterations with little drop in accuracy. This happens due to fast convergence of PR. On the other hand, Color has the highest overall inaccuracy with low-performance improvement. This happens in power-law graphs as there is a long tail of small degree nodes, only some of which get processed. Note that other algorithms do not process these scale-free graphs in degree order.

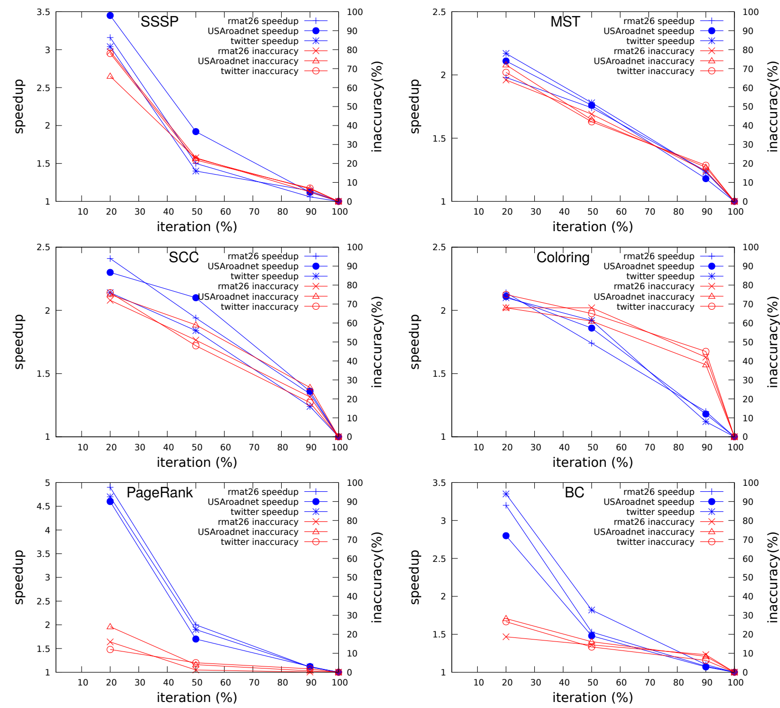


Figure 4.1: Algorithm-wise effect of varying the percentage of outer loop iterations

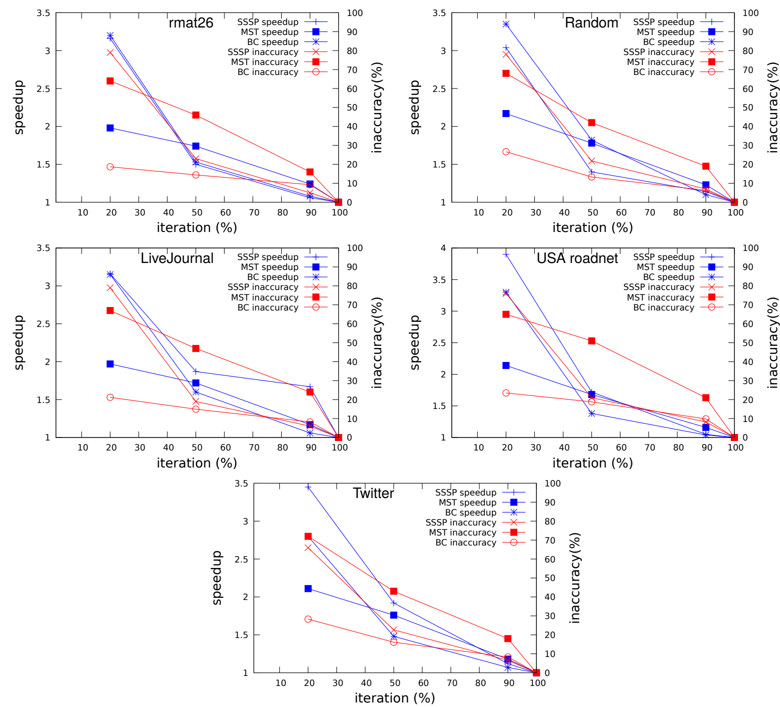
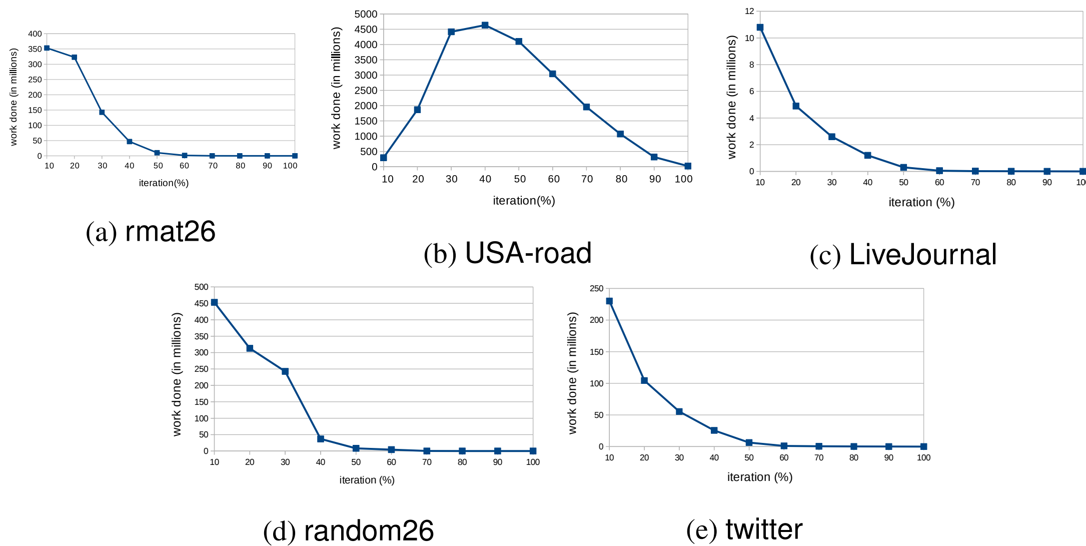


Figure 4.2: Graph-wise effect of varying the percentage of outer loop iterations



(a) rmat26 (b) USA-road (c) LiveJournal

(d) random26 (e) twitter

Figure 4.3: Work done per iteration in SSSP for rmat26, USA-road, LiveJournal, random26 and twitter

Takeaway 1 *Reduced execution approximation is beneficial for algorithms that converge quickly to the final solution.*

Takeaway 2 *Reduced execution approximation provides reduced benefits for algorithms whose precision gets affected by the long tail of vertices processed in scale-free graphs.*

Figure 4.2 shows the effect of varying the percentage of outer loop iterations for various graphs (20%, 50% and 90% iterations). We observe a good similarity in the plots across the graphs: not only the trend, but also the values are similar – which hints at the robustness of this technique (as we will see, not all techniques exhibit this robustness).

Figure 4.3 shows the work done (number of vertex distances settled) per iteration in exact SSSP for various graphs. We can choose to reduce the execution based on how much work is sufficient for the algorithm. The amount of work done is directly proportional to the accuracy and inversely proportional to its execution time.

Graph	Preprocessing Time (sec)
rmat26	26
random26	18
LiveJournal	1
USA-road	43
twitter	87

Table 4.5: Preprocessing overhead for partial graph processing

4.4.3 Effect of Partial Graph Processing

Figure 4.4 presents the effect of processing part of the graph (Section 4.2.3). A striking difference with the reduced execution (Figure 4.1) is that the accuracy of the results is largely uniform across graphs as well as across algorithms. It is interesting to observe that the speedup effect differs considerably across algorithms (MST being more amenable to this approximation over SSSP), but the inaccuracy values do not. For BC, we observe that the effect of partial graph processing on the scale-free graphs is relatively small in terms of the impact on inaccuracy than that on the road network. This is due to the difference in diameters. For low-diameter graphs, vertices can be reached from one another by traversing only a few edges. Therefore, processing only a fraction of the edges still has a high probability of traversing from one vertex to another, reducing the overall BC error. This is an indication that the inaccuracy of partial graph processing depends primarily on the amount of graph processed (more detailed results follow). This is expected, but not always true with other approximations.

Figure 4.5 shows the effect of varying the percentage of the graph processed for each graph in our testbed. We observe a considerable similarity in the shapes of the plots indicating a near-uniform effect of this approximation across graphs. There is some variation in the behavior for the road network USA-road in MST computation, but otherwise, the speedups and the inaccuracy values follow the trend.

The preprocessing overhead for partial graph processing is presented in Table 4.5. This includes the time for selecting the edges that we process in every iteration of the graph algorithm.

Takeaway 3 *Partial graph processing affects computation in a uniform manner across various graphs in our testbed.*

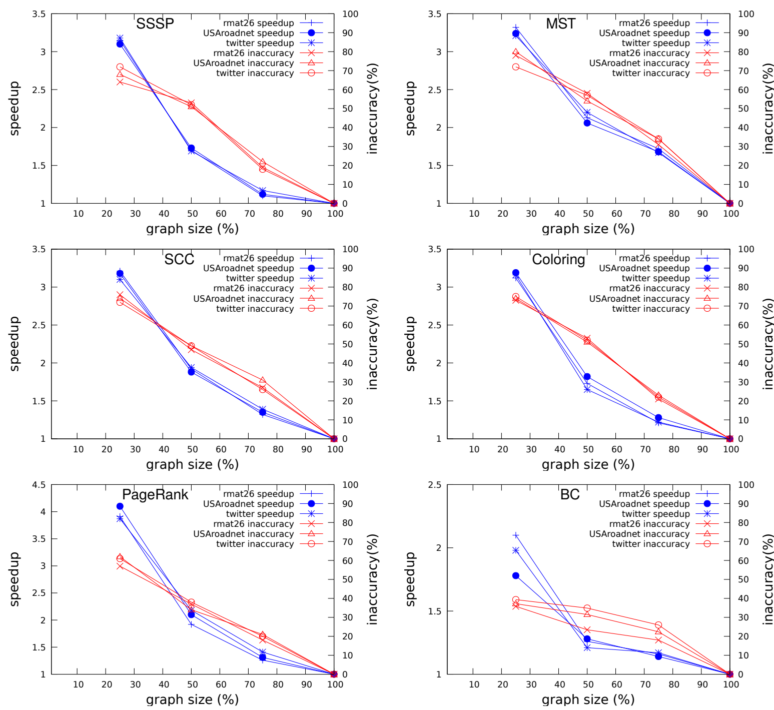


Figure 4.4: Algorithm-wise effect of varying the percentage of graph processed

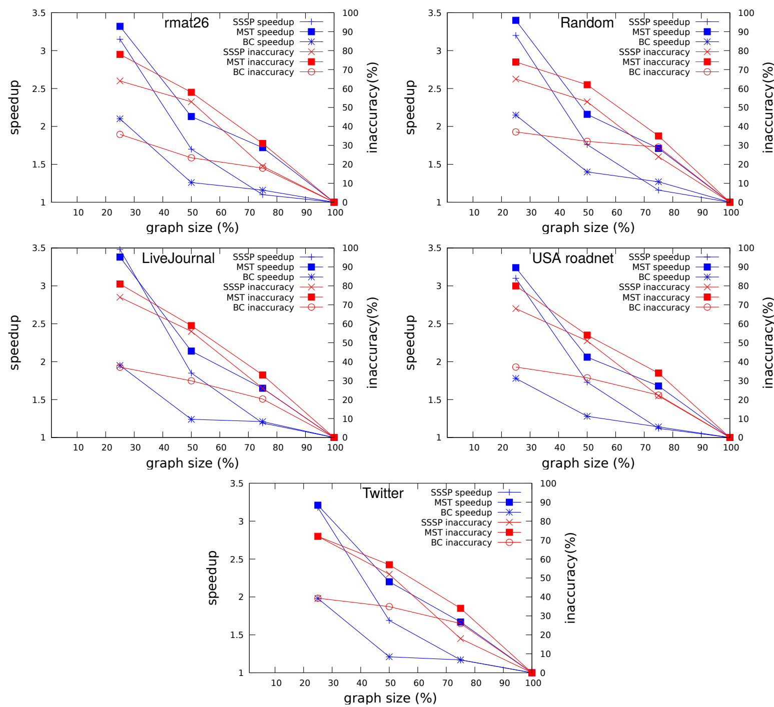
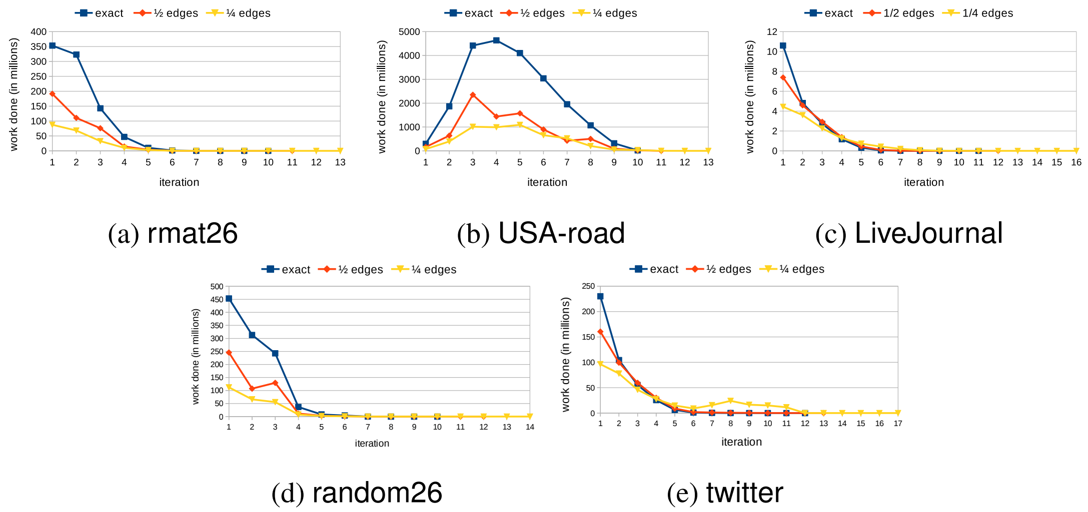


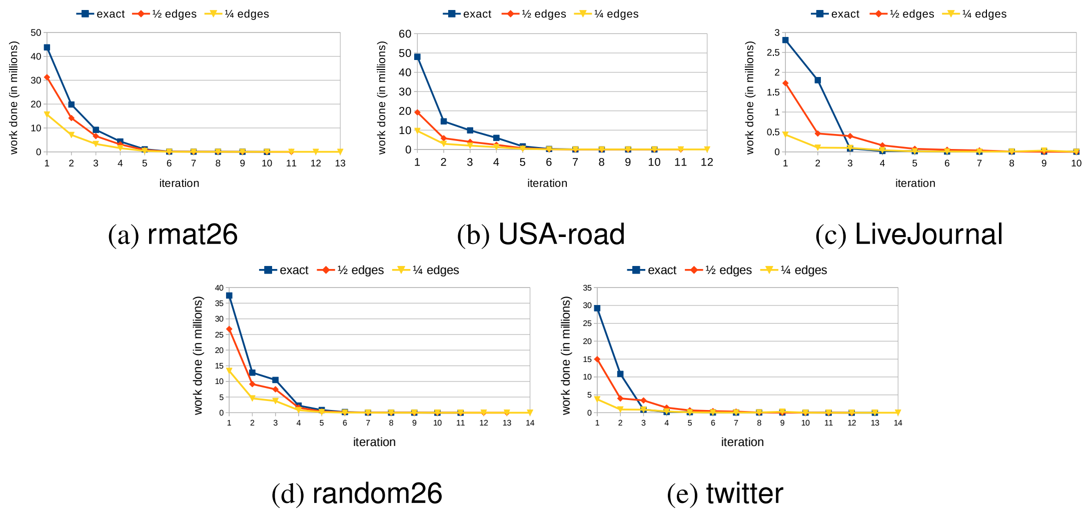
Figure 4.5: Graph-wise effect of varying the percentage of graph processed



(a) rmat26 (b) USA-road (c) LiveJournal

(d) random26 (e) twitter

Figure 4.6: Work done across iterations in SSSP for rmat26, USA-road, LiveJournal, random26 and twitter due to partial graph processing



(a) rmat26 (b) USA-road (c) LiveJournal

(d) random26 (e) twitter

Figure 4.7: Work done across iterations in Color for rmat26, USA-road, LiveJournal, random26 and twitter due to partial graph processing

Figure 4.6 shows the effect of processing part of the graph in SSSP for rmat26, USA-road and LiveJournal. It plots the amount of work done in each iteration for the number of edges processed as 100% (exact), 50% and 25%. The three plots show different shapes of these curves: for low-diameter graphs such as rmat26 and LiveJournal, the amount of work done is initially high and reduces gradually; whereas for road networks, the work done is high in the middle (due to uniform degree distribution). In all the cases, we observe that the approximate versions clearly perform much lesser work, leading to better performance.

Figure 4.7 shows the variation in work done per iteration for partial graph processing in case of Color. In this algorithm, we consider work done to be the number of nodes colored. An interesting observation is that, unlike in SSSP, the shapes of the plots remain the same and are not guided by the diameter. This occurs because coloring follows the largest-degree-first processing, and thus performs more work initially. Based on this observation, one may wish to reduce the fraction of the edges processed with increasing number of iterations.

4.4.4 Approximate Graph Representation

Approximate graph representation offers relatively higher benefits compared to the other approximation techniques.

Figure 4.8 shows the algorithm-wise effect of approximate graph representation using Jaccard's coefficient. We observe a uniformity in behavior both in terms of trend and magnitude. An interesting aspect is that the speedups and the inaccuracy values get clustered for this approximation for each graph. This occurs because similarity measure makes sure that the nodes being merged are indeed similar.

Figure 4.9 shows the graph-wise effect of the approximation. We observe a similar trend across various algorithms, and there is also a high uniformity in the magnitudes. We observe that the speedup for MST is consistently higher than SSSP and BC. This is because in parallel Boruvka's algorithm for MST, initially, there is a lot of parallelism as several nodes can perform independent edge contractions. However, after each edge contraction, the graph becomes denser with fewer nodes, lowering the available paral-

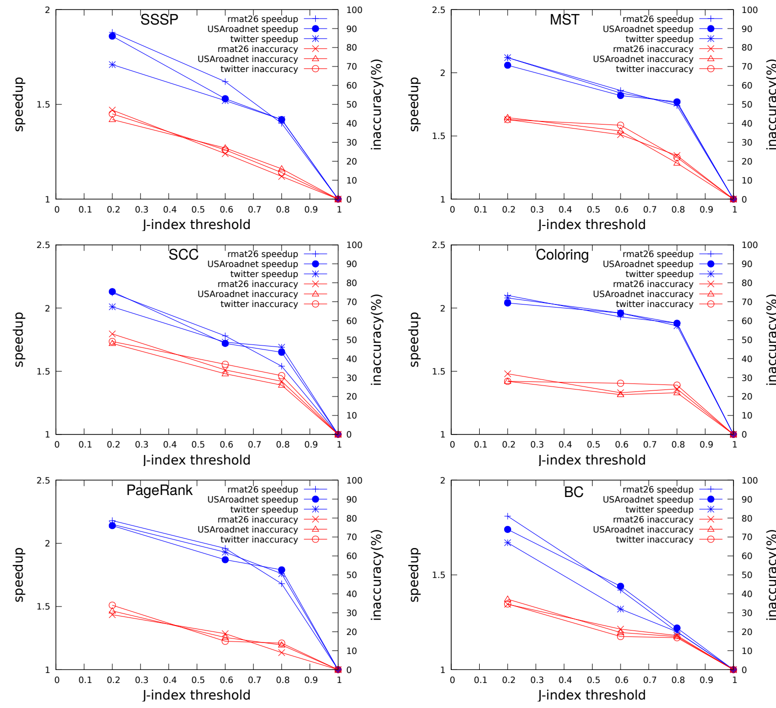


Figure 4.8: Algorithm-wise effect of varying the Jaccard's coefficient

Graph	Preprocessing Time (sec)	
	J-coeff = 0.6	J-coeff = 0.8
rmat26	81	77
random26	103	94
LiveJournal	27	18
USA-road	273	203
twitter	340	321

Table 4.6: Preprocessing overhead for approximate graph representation

lelism. In contrast, in SSSP and BC the available parallelism does not change across iterations. With a compressed graph, having fewer edges and nodes, the effect of reduced parallelism is less pronounced. Hence, the speedup for MST is higher compared that of SSSP and BC.

The preprocessing overhead for the lossy compression of the graph is presented in Table 4.6 for different thresholds for Jaccard's coefficient.

Takeaway 4 *Partial graph processing with Jaccard's similarity affect speedup and inaccuracy uniformly in our testbed.*

Figure 4.10 plots the work done per iteration for different thresholds of Jaccard's index for SCC, SSSP, PageRank, BC and MST computations. For all the algorithms, the overall work done per iteration reduces as we lower the Jaccard's index threshold

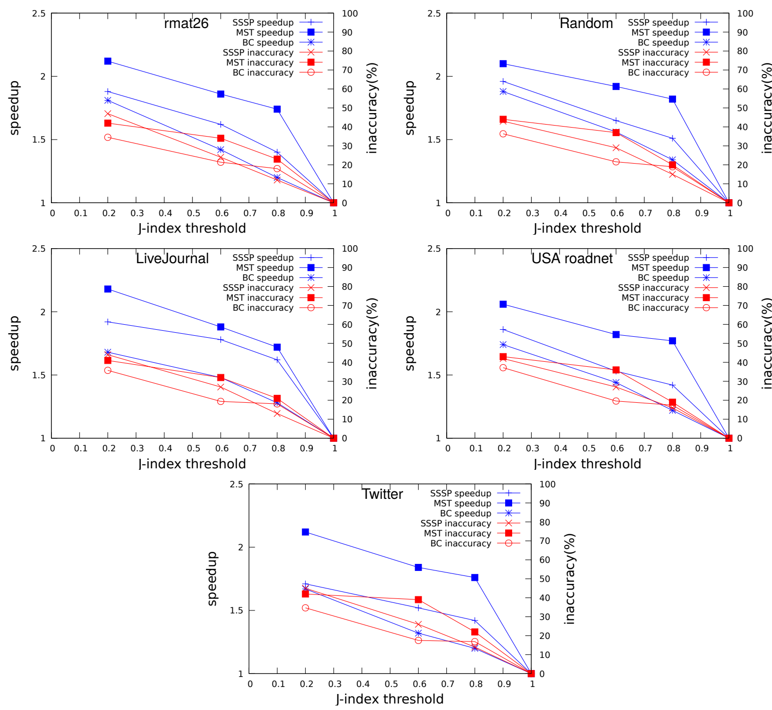
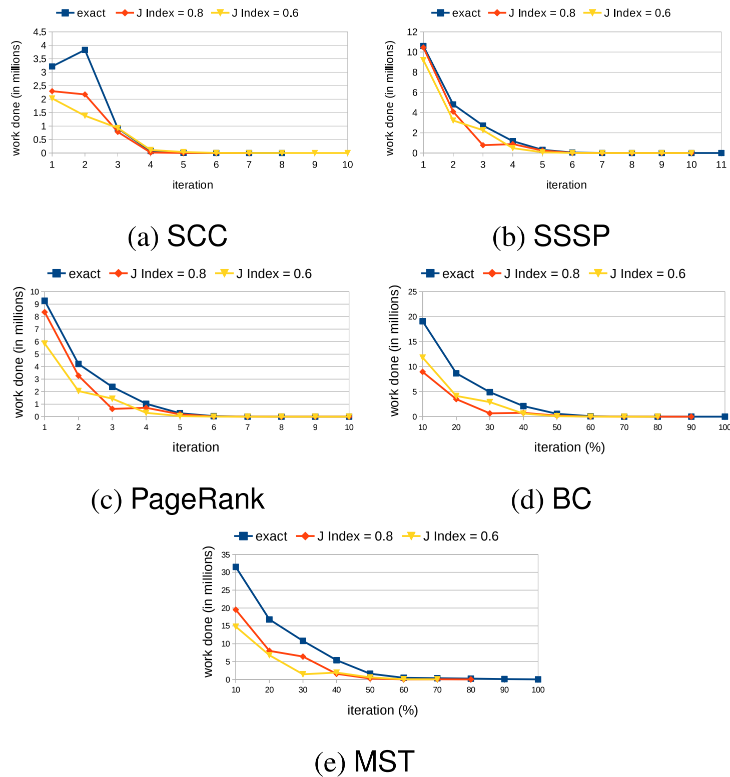


Figure 4.9: Graph-wise effect of varying the Jaccard's coefficient



(a) SCC

(b) SSSP

(c) PageRank

(d) BC

(e) MST

Figure 4.10: Work done per iteration for LiveJournal in SCC, SSSP, PageRank, BC and MST for varying Jaccard's coefficient

Graph	Preprocessing Time (sec)
rmat26	28
random26	19
LiveJournal	2
USA-road	43
twitter	89

Table 4.7: Preprocessing overhead for approximate attribute values

for merging of vertices. This is due to power-law degree distribution for LiveJournal. When the threshold for merging is high (J-index = 0.8), the merger causes the higher degree vertices to merge while the smaller degree nodes are largely left unmerged. So the number of nodes reduces but the degree of the merged nodes increases. As we reduce the threshold for merging (J-Index = 0.6), the merger causes even the smaller degree nodes to merge.

4.4.5 Approximate Attribute Values

Table 4.8 presents the effect of approximating the attribute values in SSSP, MST and BC (see Section 4.2.5). We observe relatively higher benefits with moderate inaccuracies for SSSP and MST, but consistently high inaccuracies for BC. On an average, we observe a 17% inaccuracy with a harmonic mean speedup of 1.9× in SSSP. For MST, an average inaccuracy of 19% fetched a speedup of around 1.4×. For BC, we observe a speedup of 1.4× and a high inaccuracy (∼23%). While the speedup is encouraging, the high inaccuracy is due to discretization of the BC values obtained after *every* iteration.

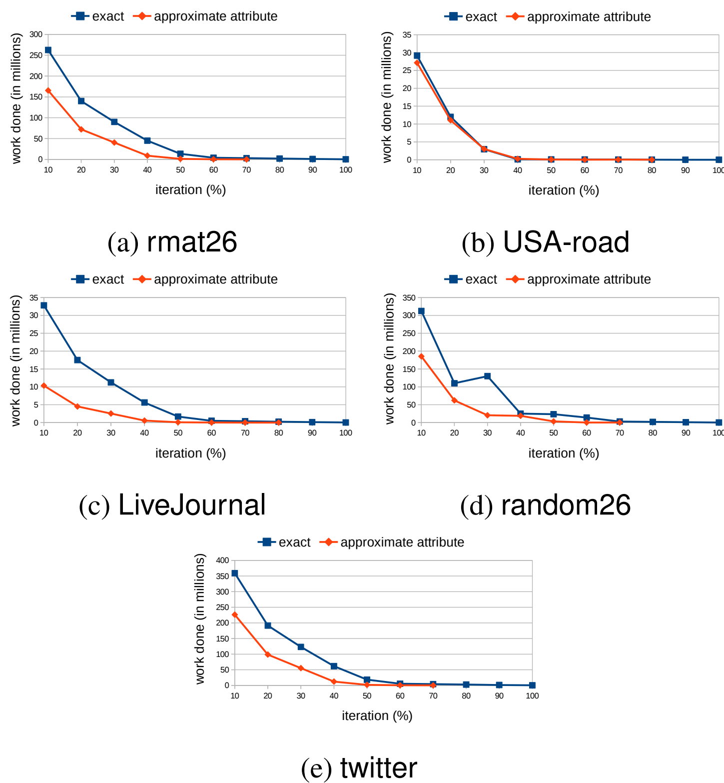
The preprocessing overheads for the approximate attribute value technique is presented in Table 4.7. This includes the time for traversing the graph and modifying the edge attributes.

Takeaway 5 *Approximating attribute values achieves better speedup at the cost of accuracy in our testbed.*

Figure 4.11 plots the work done per iteration in MST for rmat26, USA-road, LiveJournal, random26 and twitter. Work done per iteration is measured as the number of edges contracted. We observe that the work done per iteration with approximate attribute values closely follows the exact version for USA-road. Note that it does not

	Graphs	Speedup	Inaccuracy
SSSP	rmat26	2.65	14%
	random	1.42	19%
	LiveJournal	2.18	21%
	USA-road	2.06	17%
	twitter	1.58	18%
MST	rmat26	1.58	16%
	random	1.43	22%
	LiveJournal	1.64	19%
	USA-road	1.29	21%
	twitter	1.53	18%
BC	rmat26	1.40	21%
	random	1.53	23%
	LiveJournal	1.34	22%
	USA-road	1.41	27%
	twitter	1.42	25%

Table 4.8: Effect of approximating attribute values.



(a) rmat26 (b) USA-road

(c) LiveJournal (d) random26

(e) twitter

Figure 4.11: Work done per iteration in MST due to approximating attribute values

mean that the two versions contract the *same edges*; it simply indicates that they contract almost the *same number of edges*, but the approximate version converges faster. This reduces the overall work done and, correspondingly, the execution time. In case of rmat26, rmat26, LiveJournal, random26 and twitter, the work done per iteration reduces considerably due to approximating attribute values.

4.4.6 Effect on Graph Type

Our testbed consists of scale-free low-diameter graphs (rmat26, LiveJournal and Twitter), Erdös-Renyí style random graph (random), and large diameter road network (USA-road). Work-done per iteration for scale-free graphs is initially high and quickly reduces, and remains low for several iterations (long tail). In contrast, for large diameter graphs, it increases in the initial iterations, remains high for some iterations, and then gradually reduces. For random graphs, it remains almost uniform throughout. This behavior dictates how an approximation affects processing on these graphs. Reduced execution is more useful for large diameter graphs. Partial graph processing changes only the magnitude of work done per iteration. Hence, it affects graphs uniformly. A similar behavior is observed with approximate graph representation. However, we see that the performance and the inaccuracy values are more sensitive to partial graph processing compared to the approximate graph representation. Finally, the effect of approximating attribute values is not conclusively dependent upon the graph type, as graph type is a structural property whereas attribute value is a numeric property of the graph elements.

4.4.7 Graprox techniques are platform-independent

We believe that the Graprox techniques are platform-independent. The techniques focus on the the flow of information in iterative parallel graph algorithms and target various parts of computation and the input graph to arrive at an approximate solution early. These do not depend on or exploit any GPU-specific details such the GPU architecture. Hence, while we have evaluated the techniques only on GPU, we believe that Graprox would work on multicore CPUs too.

4.5 Practicality of Graprox techniques

The Graprox techniques show that applying approximations is consistently helpful in achieving better performance in graph processing. However, the proposed techniques are difficult to employ in practice in their current form. We discuss a few issues in using the techniques in practice and ways to make them more practical.

While each of the techniques provides tunable knobs to control the performance-accuracy tradeoff, arriving at the *right* values of these tunable parameters for the desired accuracy or speedup, for a new algorithm-graph pair, requires extensive experimentation. This is because the performance and accuracy of approximate techniques are contingent on the nature of the graph algorithm and the input graph. Currently, there are no mechanisms in place to estimate the performance-accuracy tradeoff during the online processing in order to aid in stopping when the specified accuracy threshold is reached. Augmenting the techniques with such a model would aid in using the techniques in scenarios where an approximation budget is specified. The model should take into consideration the flow of information in the graph algorithm, the characteristics of the input graph and the termination condition for the algorithm based on the specified accuracy metric.

Further, none of the techniques provides a worst-case theoretical bound on the amount of inaccuracy injected. As a result, in principle, for a new algorithm-graph pair, it is difficult to estimate the inaccuracy before hand. For making the Graprox techniques more practical, proving theoretical bounds on the inaccuracy would be helpful.

4.6 Summary

We studied the effect of various approximate computing techniques on parallel graph algorithms. It is believed that for irregular computations such as graph algorithms, the effectiveness of a technique depends primarily upon the input. For instance, there exist algorithms that target specially power-law graphs and which do not work well with large diameter graphs. On the contrary, our study reveals that while, in general, al-

Algo.	Technique	Mean Time (sec)
SSSP	Outer-loop iterations	60.54
	Partial processing of graph	61.36
	Approx. graph representation	71.02
	Approx. attributes	46.98
MST	Outer-loop iterations	5497.05
	Partial processing of graph	3854.25
	Approx. graph representation	4298.97
	Approx. attributes	4531.35
SCC	Outer-loop iterations	15.87
	Partial processing of graph	15.15
	Approx. graph representation	13.79
	Approx. attributes	–
Color	Outer-loop iterations	10.48
	Partial processing of graph	11.87
	Approx. graph representation	11.18
	Approx. attributes	–
PR	Outer-loop iterations	4.73
	Partial processing of graph	5.27
	Approx. graph representation	6.23
	Approx. attributes	–
BC	Outer-loop iterations	6157.01
	Partial processing of graph	7544.51
	Approx. graph representation	8055.04
	Approx. attributes	7598.01

Table 4.9: Average execution time of the approximate versions of graph algorithms

gorithmic processing and input graphs affect the magnitude of benefit, approximations consistently offer considerable improvement. In other words, approximate computation of graph algorithms is a robust way of dealing with irregularities. Table 4.9 lists the mean execution times for the different approximate techniques, across graphs, for each of the graph algorithms. Our techniques are general and applicable to other graph algorithms as well.

CHAPTER 5

GPU-specific Optimizations for Graph Processing in the presence of Approximations

According to the TAO model (Pingali *et al.* (2011)), the primary technical challenges in parallel graph processing arise due to inherent *irregularity* in the data-access, control-flow, and communication patterns. This forces compilers to make pessimistic assumptions about them as the graphs are available only at runtime, leading to reduced parallelization benefits. GPU is designed to work well on structured data. Thus, the issues with parallel graph processing get exacerbated on GPU. There are broadly three fundamental aspects that affect performance on GPU, namely, memory coalescing, memory latency, and thread-divergence. Parallel graph processing poses challenges for coalesced memory accesses due to unpredictable connectivity between graph vertices. Further, graph algorithms, are often memory-bound, and thus more sensitive to memory latency. Thread-divergence is also rampant in graph algorithms due to the load-imbalance among warp threads resulting from arbitrary degree-distribution in the graph. Hence, parallel graph processing imposes heavy performance penalties on GPU.

The prior works have successfully parallelized several popular graph algorithms on GPU. In order to make the graph processing "more" amenable to GPU, we propose Graffix, a framework of three novel graph transformation techniques, each targeting one GPU-specific aspect. The proposed graph transformations inject controlled approximations by altering the graph structure to enable faster processing in exchange for small inaccuracies in the final results. We devise a new graph reordering strategy to enable coalesced accesses, a new method exploiting clustering coefficient to improve usage of shared memory, and an edge-insertion based method to reduce thread-divergence while improving convergence. Our techniques offer *knobs* that can be *tuned* to control the amount of approximation injected. Further, our techniques do not compete with the existing GPU-specific optimizations, but complement those. They can be combined for improved benefits.

This chapter is organized as follows. Sections 5.1.1,5.2 and 5.3 describe in detail our proposed techniques for improving memory coalescing, reducing memory latency and reducing thread divergence, respectively. Section 5.4 quantitatively evaluates the performance of Graffix techniques and compares those with the state-of-the-art exact versions of the respective algorithms.

5.1 Improving Memory Coalescing

A GPU-parallel algorithm exemplar: *betweenness centrality computation*. Consider the parallel Brandes' algorithm (Prountzos and Pingali (2013)) for computing the vertex betweenness centrality in an unweighted graph, as shown in Algorithm 7. It as an exemplar of a general class of parallel algorithms on GPU.

We pursue the inner parallel strategy of parallelizing Brandes' algorithm, i.e., each of the computation steps (lines 3, 7 in Algorithm 7) is executed in parallel for a single source, and different sources are processed in sequence. In the forward pass, each thread enumerates a vertex's neighbors and updates σ_{sv}. On GPU, due to multiple threads writing to the same vertex's σ, threads need to synchronize using an atomic instruction (such as `atomicAdd` from CUDA). On the memory access front, the memory access for σ in Algorithm 7 is generally uncoalesced due to unpredictable connectivity of nodes. Reading (and writing) a node's neighbors' σ also suffers from low locality which causes significant memory latency and limits overall performance. In addition, since warp-threads assigned to different vertices may process different numbers of neighbors, the forward pass incurs thread divergence. Similarly, in the backward pass, processing δ attribute of a node's predecessors leads to reduced coalescing, low locality, and high thread-divergence.

5.1.1 Coalescing in Graffix

Graffix makes the graph more *structured* to improve coalescing. To this end, we devise a modified graph layout by rearranging graph nodes, edges, and their associated information to make warp-threads access nearby memory locations with higher probability.

Algorithm 8 Graffix technique for improving memory coalescing

```
Input: Graph G(V, E)
Output: Graph G'(V', E')
 1: function TRANSFORMGRAPH(G)
    ▷ Step 1: Vertex renumbering
 2:     v.level = ∞        ∀v ∈ G.V
 3:     for Node s : G.V orderedby (decreasing node degree) do
 4:         if s.level == ∞ then
 5:             s.level = 0
 6:             BFS(G, s)                                    ▷ Assigns levels to nodes
 7:     RENUMBERVERTEX(G, k)                                 ▷ k is the chunk size
    ▷ Step 2: Node replication
 8:     REPLICATEVERTEX(G, k)
    ▷ The transformed graph is G'(V',E')
 9: end function

10: function RENUMBERVERTEX(G, k)
11:     gId = 0;
12:     for Node n : L_0 do          ▷ L_0 is list of nodes at the 0^th level in G's BFS forest
13:         n.id = gId++;
14:     for i = 0 .. numLevels-2 do                  ▷ numLevels is number of BFS levels
15:         gId = ⌈gId/k⌉ × k
16:         for j = 0 .. (max node degree in L_i) do     ▷ L_i is the list of nodes at level i
17:             for Node n : L_i do
18:                 if (n.degree > j) && (n.neighbors[j] ∈ L_{i+1}) then
19:                     n.neighbors[j].id = gId++
20: end function

21: function REPLICATEVERTEX(G, k)
    ▷ Nodes array divided into chunks of size k, such that, chunkId[u] = u/k
22:     for Node n : G.V do                                  ▷ n is a non-hole node
23:         count_n = [ ]        ▷ hash table to count the number of edges from n to a chunk
24:         for Node v : n.neighbors do
25:             if v ∈ L_i && ∃u ∈ L_{i-1}, such that, u is a hole then
26:                 count_n[v.chunkId]++
27:         for curChkId : count_n.ChunkIds do          ▷ ids of chunks having edges from n
28:             connectedness^n_curChkId = count_n[curChkId] / # non-hole nodes with curChkId
29:             if connectedness^n_curChkId ≥ threshold then
30:                 Duplicate n to get n'
31:                 n'.id = u.id, such that, curChkId in L_i && u ∈ L_{i-1}        ▷ Fill holes
32:                 for Node p : n.neighbors, such that, p.chunkId == curChkId do
33:                     Remove edge n → p
34:                     Add edge n' → p
35:                 Add edges n' → q    ∀q, such that, q.chunkId == curChkId;
                                        q is a 2-hop neighbor of n
36: end function
```

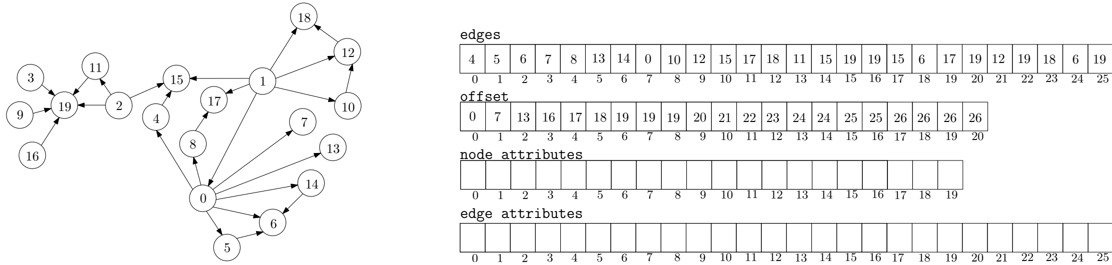


Figure 5.1: Original graph G and its CSR representation

Note that it is not sufficient to rely on hardware prefetchers or software prefetch instructions to get the data on demand. Hardware prefetches are not very effective when the accesses are irregular. Further, software prefetch instructions need to be placed meticulously in the code to ensure that the data brought into the cache is used before being evicted, which is again challenging due to irregular accesses. Graph restructuring helps make the graph more structured and also aids the prefetcher.

We use the Compressed Sparse Row (CSR) format to represent the graph, having an *offset* array, an *edges* array, and auxiliary arrays to store *edge attributes* and *node attributes*. Figure 5.1 shows an example directed graph and its CSR representation. In a vertex-based processing, a thread is assigned to a vertex . Hence, accesses to the offset array and the source vertex attribute array are coalesced. However, due to the *irregular* memory access pattern of the *node attributes* array resulting from the neighbor traversal of the nodes, the accesses to *node attributes* array are largely uncoalesced.

A key primitive in graph operations is *neighbor enumeration* wherein a warp, assigned to a set of vertices, iterates through their neighbors to propagate information. Such a neighbor enumeration is done in all the algorithms in our experimental setup. Graffix improves coalescing for this primitive. At a high level, our technique uses a careful combination of renumbering and replication to bring together in memory the data of those nodes that are likely to be accessed in tandem. Vertex re-numbering helps bring connected nodes and their data together. However, it has a limitation that a node occurs exactly once, and therefore it cannot be nearby all its neighbors (as their node ids could be far apart). This limitation is overcome with replicating the node, and thereby, allowing such a node to be nearby its neighbors. Graffix creates copies of a node, subject to a certain condition, and inserts the copies of these nodes, along with their edge-lists, in the vicinity of their neighbors in the CSR representation. Algorithm 8

presents the pseudocode of our technique. TRANSFORMGRAPH() is the driver routine. We explain the scheme in detail below.

5.1.2 Renumbering Scheme

Graffix renumbers vertices such that warp-threads are assigned nearby id's. While node re-numbering is well-explored in literature to improve thread-divergence and locality (Hong *et al.* (2011); Balaji and Lucia (2019)), it is ineffective when applied directly to improve coalescing. This is because the numbering assigns consecutive id's to a node's neighbors. This improves locality in serial processing, as all the neighbors would be processed by the same thread. However, due to this, threads belonging to the same warp end-up processing vertices numbered far apart – reducing coalescing. For instance, in Figure 5.1, assume the warp-size to be 4. The nodes 0–3 are assigned to threads having the same id as the node. With vertex centric processing, the warp-threads would access the attributes of the first neighbor of the respective nodes concurrently, and so on. The first neighbors are indicated by the `offset` array: 0, 7, 13, 16, to be indexed into the `edges` array. Hence, the warp threads would access the locations 4, 0, 11, and 19 in the node attributes array together. Further, assuming that the accesses to a chunk of 4 words can be coalesced, clearly, the accesses to the destination nodes' (4, 0, 11, 19) data in the *node attributes* array are not coalesced since each of these lies in a separate 4-word chunk. Therefore, we propose a new numbering scheme for improving coalescing.

The numbering starts with a vertex having the highest outdegree and performs breadth-first traversal (BFS) on the graph, till all the nodes in the graph are visited, to obtain a BFS tree or a BFS forest if the graph is disconnected. In the graph is disconnected, the source nodes for the subsequent BFS traversals are picked in the decreasing order of outdegree among the unvisited nodes. The levels of the visited nodes are updated to a lower value, if possible, in the case of multiple BFS traversals. This is accomplished by the loop at line 3 in Algorithm 8. For example, in the graph G from Figure 5.1, vertex 0 has the highest outdegree. Performing BFS from vertex 0 on G visits vertices {0, 4, 5, 6, 7, 8, 13, 14, 15, 17}. The source for the next BFS is vertex 1 since it has the highest outdegree among the unvisited nodes. BFS from 1 covers

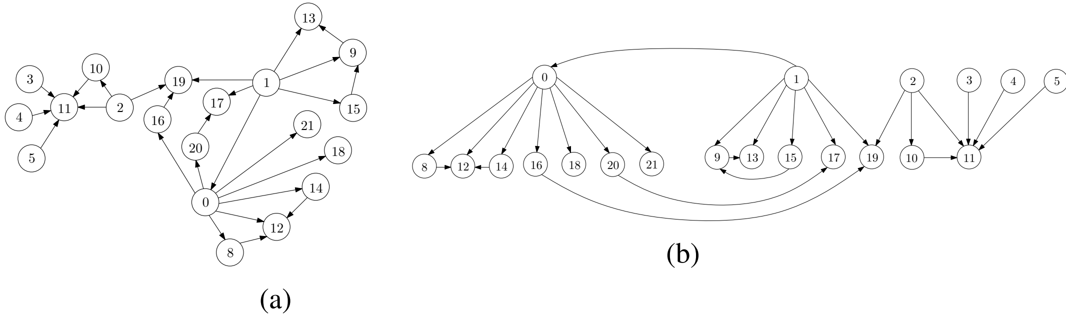


Figure 5.2: (a) Graph G from Figure 5.1 with renumbered nodes (b) The same graph reoriented for clarity

vertices {1, 10, 12, 18} among the unvisited nodes. Further, among the already visited nodes, the levels of nodes 15 and 17 are reduced to 1. Next, applying BFS from node 2 covers vertices {2, 11, 19}, while BFS from 3, 9 and 16 cover vertices 3, 9 and 16, respectively. Thus, vertices 0, 1, 2, 3, 9 and 16 are at level zero, while all others are at level one.

An important observation is that the nodes at the same level in the BFS forest are going to be accessed by consecutive threads. Hence, those are assigned id's incrementally in a round-robin fashion. Thus, the first neighbor of each of the parents from the previous level is assigned a new id followed by the renumbering of all the second-neighbors, and so on. For instance, in Figure 5.2b, which shows the renumbered graph, node 8 is the first unnumbered neighbor of node 0, while node 9 is the first unnumbered neighbor of node 1. A crucial aspect of Graffix's numbering scheme is that the new node id's at each level of the BFS forest start from a multiple of k ($1 \leq k \leq$ warp-size) as shown in RENUMBERVERTEX() routine at line 10 of Algorithm 8. This is different from the prior numbering schemes, and provides an opportunity for accesses to be coalesced *at every level*. For instance, Figure 5.2a is the renumbered graph with $k = 8$, and Figure 5.2b is its isomorph. With the new renumbering, vertices 0 through 5 are at BFS level zero. The next level starts with a multiple of k $(= 8)$ greater than the last vertex id 5 (that is, there are no vertices with id's 6 and 7). Hence, the next level is occupied by vertices 8 through 21.

Creation of Holes. An important aftereffect of Graffix numbering is that since not all levels have the number of nodes in multiples of k, the renumbering scheme may create *holes* in the CSR representation arrays. For instance, the renumbering gives rise

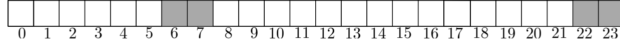


Figure 5.3: Holes in nodes after renumbering G

to *holes* in the nodes array at locations 6, 7, 22 and 23 as shown in Figure 5.3. The choice of k controls the number of *holes* at each level of the BFS forest. The number of holes at a level is $\leq (k-1)$. Graffix exploits these *holes* to enhance the degree of coalescing. It uses replication to copy specific nodes to these holes. The controllable node replication modifies the underlying graph and can introduce some approximation. Carefully identifying the nodes to replicate aid the underlying graph computation to reach its fixed-point faster.

5.1.3 Node Replication

The node replication to fill the holes needs to ensure that (i) it improves coalescing, leading to improved execution time, and (ii) the error is small. This is done as follows: Following the renumbering, the *nodes* array (which now also includes holes) is divided into chunks of size k, the same as that used for vertex renumbering. The nodes of a chunk are processed by a warp. If a node is *well-connected* to a chunk, our goal is to replicate the node in the chunk containing the parents of the chunk's nodes (as obtained from the BFS forest). This makes sure that we take advantage of the *renumbering* after replication of the nodes. Graffix achieves it as follows. From each of the non-hole nodes, we maintain a count of the outgoing edges to the chunks whose parent chunks have *holes*. Further, we define $\text{connectedness}_{\text{chunk}}^{\text{node}} \triangleq \left(\frac{\text{\# edges to chunk from a node}}{\text{\# non-hole nodes in chunk}}\right)$ for each such node–chunk pair. If the *connectedness* of a node to a chunk is higher than a threshold, the node is deemed to be well-connected to the chunk and thus we replicate the node. The *threshold* is a tunable parameter and controls the amount of inaccuracy. When there are more candidate nodes eligible for replication to a chunk, than holes in that chunk, the nodes with higher *edge-count* are given priority.

Since *holes* in the CSR representation arrays do not contribute to any 'useful' work done, it is instructive to reduce the unoccupied holes in the modified graph. A judicious choice of k and *threshold* is instrumental in increasing the occupancy of the holes. In our experiments, we use $k = 16$ and set the *threshold* to 0.6 and 0.4 for the scale-free

graphs and the road networks respectively. We determined these values of the threshold empirically. These were found to produce the best results. From a node replica, we introduce new edges to the non-hole nodes of a chunk. If the node being replicated has an edge to a node in the chunk, we add edges from the replica to its 2-hop neighbors inside the chunk to which there is not already an edge.

Controlling the approximation. Adding edges in this manner expedites the propagation of information among nodes, while ensuring that the node attributes read or written to in a coalesced fashion also contribute to some meaningful computation. The amount of inexactness is proportional to the number of new edges added in the graph. Thus, by controlling the number of newly added edges, Graffix can keep the inaccuracy in check. The addition of edges as above results in only a few additional edges per replica since we restrict the view to a contiguous chunk of size k in the nodes array, at a time, while looking for the 2-hop neighbors of the node being replicated. Graffix ensures that the node to be replicated has a high degree. So, adding few extra edges adds only small inaccuracy.

For our example, we divide the *nodes* array (Figure 5.3) in the renumbered graph into chunks of size k $(= 8)$. Assume that the threshold on *connectedness* for replication of a node is set to 0.6. In the renumbered graph in Figure 5.2, node 0 has 4 edges to the chunk 16..23 and the chunk has two holes. Hence, the $connectedness^{0}_{16..23} = \frac{4}{6} = 0.667$. Since the *connectedness* of 0 to the chunk is greater than the *threshold*, we replicate 0 in chunk 0..7. We assign the id 6 to the replica of 0 and distribute the existing edges of 0 between 0 and 6. We also add new edges from 6 to nodes 17 and 19, as these are the 2-hop neighbors of 0 in the chunk 16..23. This leads to the modified graph G' shown in Figure 5.4.

5.1.4 Confluence due to Replication

Due to controlled node replication, the underlying graph structure undergoes some changes. As an aftermath, different node-copies in the modified graph may exhibit different attribute values at the end of a GPU kernel iteration. Since logically these copies represent the same node, these attribute values need to be merged. The merging

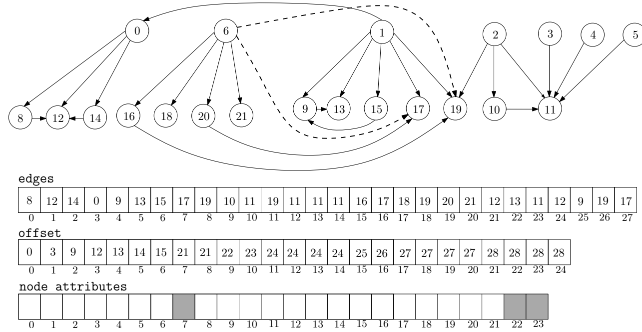


Figure 5.4: Modified graph G' with its CSR representation.

or the confluence may be done after certain number of iterations or at the end of the overall computation. To reduce inaccuracies, Graffix merges attribute values from the copies of the same node after every iteration. The merge operator itself could be defined in two-ways: (i) algorithm-aware, and (ii) algorithm-agnostic. The former is likely to result in better accuracy, but needs additional knowledge. Graffix uses the latter approach and applies a generic confluence operator which computes arithmetic mean of different values. One can easily redefine the merging. For instance, in Figure 5.4, at the beginning of the algorithm, the node attributes of nodes 0 and 6 will be the same. After each iteration, we merge the attribute values of nodes 0 and 6 using arithmetic mean to ensure that both are the same at the beginning of each iteration. At the end of the algorithm, all the node copies will have identical attribute values.

5.2 Reducing Memory Latency

Since graph algorithms are memory bound, we seek to exploit the GPU memory hierarchy to reduce the time spent in fetching/updating data from/to global memory to curtail the execution time. Shared memory available per thread-block has been exploited in various ways in literature, and demands reuse of data items. For instance, in an unrolled kernel, the updated attribute values can be kept temporarily in shared memory. Alternatively, when a connected subgraph is processed by a thread, the stack or the queue can be stored in shared memory depending upon whether the subgraph traversal is depth-first or breadth-first (Nasre *et al.* (2013*b*)).

Graffix proposes a new way of exploiting shared memory to process more-

frequently-accessed nodes. Identifying such nodes at runtime adds inefficiency. On the other hand, identifying such nodes based on crude approximations such as degree is not very fruitful. Graffix exploits the graph property of *clustering coefficient* (CC) to identify such nodes. For the purpose of computing CC, we consider the graph to be undirected. The nodes having CC higher than a *threshold* are moved to shared memory, along with their neighbors. For instance, in Figure 5.5a, node N_1 has a high CC, so it can be moved to shared memory along with its neighbors. As nodes with a high CC are part of well-connected clusters, such clusters will be accessed frequently in iterative processing of the graph. Such high-CC nodes can be moved to shared memory. Due to the power-law distribution, very few nodes have very high CC leading to underutilization of shared memory. Adding approximation improves applicability of the technique. Graffix selectively adds edges between nodes to effect the following:

(1) Increase the CC of the nodes having CC lower than, but close to, the threshold. This allows moving such nodes, along with the neighbors, from global memory to shared memory.

(2) Further boosting the CC of the nodes whose CC are already higher than the threshold.

In the first case, we add new edges preferentially between those neighbors of a high-CC node that have common neighbors. The purpose is to increase the CC of the node, and its neighbors, to make them candidates for being processed inside shared memory. In the second scenario, we add edges between those neighbors of a high-CC node that have the fewest edges with the other neighbors of that high-CC node. The rationale is to increase the connectivity among the neighbors of the high-CC node. Graffix looks at the connectivity only among the siblings of the high-CC nodes since these nodes will be in shared memory. We move the high-CC nodes to shared memory, along with their immediate neighbors alone. For instance, in Figure 5.5a, we add edges between the neighbors of N_1 having the fewest edges incident on them, that is, nodes `A, B, C, D`. For faster convergence, in both the scenarios above, the edges are added between the 2-hop neighbors. Only a few edges are added in this manner. Additionally, we maintain a global limit for the number of edges added to the graph to contain the approximation. To enable reuse, the sub-graphs in shared memory are processed for a few iterations (say, t). We found that setting $t \sim (2 \times \text{diameter of the subgraph})$ gave

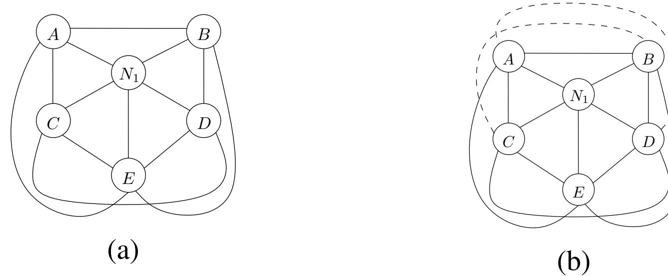


Figure 5.5: Reducing memory latency using shared memory

good performance benefits because of sufficient reuse. Thereafter, the attribute values of the nodes are pushed back to global memory.

Discussion. An alternative scheme for increasing the number and the size of the subgraph processed inside shared memory is to set a lower threshold on the clustering coefficient. However, this results in diminished benefits due to low reuse and impaired accuracy. Therefore, it is recommended to keep the CC cut-off relatively high.

5.3 Reducing Thread Divergence

While degree-sorting (Balaji and Lucia (2019)) is an effective way to address thread-divergence, it is often an overkill, since having *nearly-uniform* degrees *only within each warp* often suffices. Graffix combines bucket-sort and approximate computing to reduce thread-divergence, as we explain below. As a preprocessing step, Graffix performs bucket sort on the `nodes array` using the node-degree as the key. This groups the nodes having *similar* degrees together. In each bucket we assign nodes to warps in the order of their bucket positions. When node degrees are different, we carefully add a few edges to reduce thread-divergence. Judicious addition of edges reduces the effect of approximations.

Adding edges. Additional edges are the source of approximation. Hence, among the warp-nodes, we add extra edges to only those nodes that are deficient in their connectivity. If the difference of a node's degree to the warp's max-degree is lower than a *threshold*, we add edges to it to get its degree *close* to the warp nodes' max-degree. This causes the warp node degrees to become more uniform. The threshold dictates the

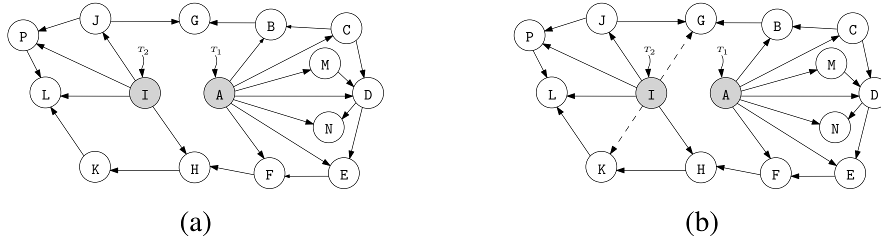


Figure 5.6: Handling thread divergence by graph transformation.

number of edges added. As an extreme, it is possible to remove thread-divergence fully with this technique.

By noting that most graph algorithms are propagation-based, we choose the destination nodes to be the 2-hop neighbors, leading to faster convergence. While the structural changes take care of node degrees, the choice of the edge-weights for the new edges (for weighted algorithms such as SSSP) is often fuzzy. In the case of weighted graphs, we set the weight of a new edge as the sum of the weights of the edge between the node and the 1-hop neighbor, and the edge between the 1-hop and the 2-hop neighbors. One can choose an alternative method to setup the edge-weights.

By adding edges in this manner, the warp threads which would otherwise be waiting for the longest running warp thread to complete are also able to perform some useful work in the meantime. The information propagated to their 2-hop neighbors is useful for the next iterations of the algorithm. Thus, the extra work done by the few warp threads per iteration contributes to the overall improvement in performance.

Example. Consider the graph in Figure 5.6a. Suppose threads T_1 and T_2 belong to the same warp and are operating on nodes `A` and `I` respectively. Since the outdegree of node `A` (7) is more than that of `I` (4), T_1 has to process more edges than T_2. Assume that the threshold on the difference in node degrees for the purpose of adding edges is $\frac{max-degree}{2}$. Also, assume that vertex A is the max-degree node in the warp. As the difference in the degrees of I and A is 3, which is less than $\frac{7}{2}$ (= 3.5), our method adds new edges `IG` and `IK` to make the outdegree of node `I` close to the max-degree. The new outdegree of `I` is 6 ($\sim$85% of max-degree). Nodes `G` and `K` are 2-hop neighbors of `I`. Figure 5.6b shows the modified graph.

Effect of Graffix techniques on parallel BC. In Algorithm 7, Graffix's technique for

memory coalescing brings closer nodes that are accessed in tandem by the warp-threads during the graph traversal on lines 3 and 7. The technique for memory latency ensures that the *well-connected* subgraphs are processed iteratively in shared memory. Further, the degrees of the warp-nodes during the traversal are normalized to curtail workload-imbalance.

5.4 Experimental Evaluation

We evaluate the performance of our approximate techniques and compare it with the exact versions of the respective algorithms.

Machine Configuration. We use the same machine as described in Section 4.4.

Input Graphs. We select graphs with varying characteristics to demonstrate the robustness of our approach. The graphs we use for evaluation are listed in Table 4.2. These graphs exhibit different behaviors for different techniques.

Graph Algorithms. We study five graph problems: single-source shortest paths computation (SSSP), minimum spanning tree computation (MST), finding strongly connected components (SCC), page rank (PR), and node betweenness centrality computation (BC). Parallel algorithms for these problems have been described in detail in Section 2.4. All these problems are popular in the community and, along with various graphs, their parallel algorithms stress-test our techniques.

Baselines. We use three baselines to evaluate our techniques. First, we compare our approximate techniques with the exact implementation of SSSP, PR and BC available in Gunrock (Wang *et al.* (2017)). Second, we compare our approximate techniques with the exact implementation of SSSP, PR and BC available in Tigr (Nodehi Sabet *et al.* (2018)). Third, we compare our approximate SSSP and approximate MST with the respective exact versions from LonestarGPU (Burtscher *et al.* (2012)), approximate SCC with the exact SCC by Devshatwar *et al.* (Devshatwar *et al.* (2016)), and approximate PR and BC with our own parallel implementations of PR computation (10 iterations) and Brandes' algorithm (10000 iterations) respectively.

Graph	Exact Time (sec)				
	SSSP	MST	SCC	PR	BC
rmat26	37	8996	21	12	15223
random26	29	10087	23	16	13127
LiveJournal	2	3424	7	1	1711
USA-road	152	82	12	1	2043
twitter	231	10943	37	18	21462

Table 5.1: Baseline-I: Execution time for the exact versions

Graph	Exact Time (sec)		
	SSSP	PR	BC
rmat26	6	0.914	587
random26	4	1.180	498
LiveJournal	0.046	0.452	66
USA-road	12	0.130	38
twitter	17	3.000	827

Table 5.2: Baseline-II: Execution time for Tigr

The execution times of the exact methods on the five graphs for the algorithms from the three baseline implementations are presented in Tables 5.1, 5.2 and 5.3. We report on the effect of approximations on the actual execution times of the algorithm implementation, which preclude file I/O and preprocessing steps, but include graph attribute initialization (such as vertex distances), initial CPU-GPU data transfer, and the main fixed-point loop repeatedly calling the primary kernel. We measure the inaccuracy incurred for each of the techniques by averaging the absolute difference between the attribute values of the vertices for the exact and the approximate versions. For SSSP, the attribute is the distance; for PR, it is the page rank; and for BC, it is the betweenness centrality. For SCC, we calculate the difference in the number of connected components, while for MST, we calculate the difference in the minimum spanning tree weights computed by exact and approximate methods.

5.4.1 Effect of Coalescing

Table 5.4 shows the effect of Graffix's technique for coalescing for five graphs on the five algorithms from Baseline-I. We report the results with threshold on *connectedness*

Graph	Exact Time (sec)		
	SSSP	PR	BC
rmat26	19	1.070	872
random26	8	1.500	740
LiveJournal	0.142	0.530	98
USA-road	25.139	0.181	56
twitter	53	4.000	1227

Table 5.3: Baseline-III: Execution time for Gunrock

	Graphs	Speedup	Inaccuracy
SSSP	rmat26	1.22 ×	12%
	random26	1.13 ×	10%
	LiveJournal	1.18 ×	11%
	USA-road	1.15 ×	9%
	twitter	1.17 ×	12%
MST	rmat26	1.18 ×	13%
	random26	1.13 ×	15%
	LiveJournal	1.14 ×	12%
	USA-road	1.23 ×	11%
	twitter	1.17 ×	13%
SCC	rmat26	1.14 ×	8%
	random26	1.08 ×	14%
	LiveJournal	1.13 ×	7%
	USA-road	1.16 ×	11%
	twitter	1.15 ×	12%
PR	rmat26	1.20 ×	5%
	random26	1.15 ×	7%
	LiveJournal	1.21 ×	7%
	USA-road	1.19 ×	6%
	twitter	1.22 ×	7%
BC	rmat26	1.17 ×	9%
	random26	1.12 ×	13%
	livejournal	1.15 ×	10%
	USA-road	1.19 ×	12%
	twitter	1.14 ×	11%
	Geomean	**1.16** ×	**10%**

Table 5.4: Effect of memory coalescing

Graphs	Speedup	Inaccuracy
rmat26	1.26 ×	12%
random26	1.08 ×	17%
LiveJournal	1.22 ×	13%
USA-road	1.30 ×	13%
twitter	1.18 ×	12%
rmat26	1.22 ×	16%
random26	1.10 ×	18%
LiveJournal	1.18 ×	16%
USA-road	1.20 ×	19%
twitter	1.16 ×	15%
rmat26	1.20 ×	12%
random26	1.10 ×	16%
LiveJournal	1.22 ×	13%
USA-road	1.20 ×	12%
twitter	1.18 ×	13%
rmat26	1.32 ×	7%
random26	1.16 ×	11%
LiveJournal	1.26 ×	7%
USA-road	1.30 ×	5%
twitter	1.22 ×	9%
rmat26	1.24 ×	14%
random26	1.13 ×	18%
LiveJournal	1.21 ×	16%
USA-road	1.26 ×	15%
twitter	1.17 ×	13%
Geomean	**1.20** ×	**13%**

Table 5.5: Effect of shared memory

Graphs	Speedup	Inaccuracy
rmat26	1.06 ×	8%
random26	1.03 ×	9%
LiveJournal	1.07 ×	8%
USA-road	1.12 ×	7%
twitter	1.09 ×	6%
rmat26	1.05 ×	10%
random26	1.02 ×	11%
LiveJournal	1.07 ×	8%
USA-road	1.09 ×	10%
twitter	1.05 ×	9%
rmat26	1.04 ×	9%
random26	1.00 ×	7%
LiveJournal	1.04 ×	6%
USA-road	1.05 ×	9%
twitter	1.06 ×	8%
rmat26	1.10 ×	4%
random26	1.04 ×	9%
LiveJournal	1.08 ×	5%
USA-road	1.06 ×	8%
twitter	1.09 ×	8%
rmat26	1.11 ×	11%
random26	1.05 ×	14%
livejournal	1.09 ×	9%
USA-road	1.12 ×	7%
twitter	1.06 ×	12%
Geomean	**1.07** ×	**8%**

Table 5.6: Effect of thread divergence

Approximate Graffix versus exact Baseline-I

	Graphs	Speedup	Inaccuracy
SSSP	rmat26	1.16 ×	12%
	random26	1.06 ×	10%
	LiveJournal	1.13 ×	11%
	USA-road	1.08 ×	9%
	twitter	1.12 ×	12%
PR	rmat26	1.14 ×	5%
	random26	1.08 ×	7%
	LiveJournal	1.15 ×	7%
	USA-road	1.12 ×	6%
	twitter	1.15 ×	7%
BC	rmat26	1.09 ×	9%
	random26	1.05 ×	13%
	livejournal	1.07 ×	10%
	USA-road	1.11 ×	12%
	twitter	1.06 ×	11%
	Geomean	**1.10** ×	**9%**

Table 5.7: Effect of memory coalescing

Graphs	Speedup	Inaccuracy
rmat26	1.24 ×	12%
random26	1.07 ×	17%
LiveJournal	1.20 ×	13%
USA-road	1.26 ×	13%
twitter	1.15 ×	12%
rmat26	1.30 ×	7%
random26	1.14 ×	11%
LiveJournal	1.26 ×	7%
USA-road	1.28 ×	5%
twitter	1.22 ×	9%
rmat26	1.19 ×	14%
random26	1.11 ×	18%
LiveJournal	1.17 ×	16%
USA-road	1.23 ×	15%
twitter	1.16 ×	13%
Geomean	**1.19** ×	**12%**

Table 5.8: Effect of shared memory

Graphs	Speedup	Inaccuracy
rmat26	1.02 ×	8%
random26	1.01 ×	9%
LiveJournal	1.02 ×	8%
USA-road	1.04 ×	7%
twitter	1.03 ×	6%
rmat26	1.06 ×	4%
random26	1.02 ×	9%
LiveJournal	1.04 ×	5%
USA-road	1.03 ×	8%
twitter	1.05 ×	8%
rmat26	1.04 ×	11%
random26	1.01 ×	14%
livejournal	1.02 ×	9%
USA-road	1.05 ×	7%
twitter	1.03 ×	12%
Geomean	**1.03** ×	**8%**

Table 5.9: Effect of thread divergence

Approximate Graffix versus exact Baseline-II

	Graphs	Speedup	Inaccuracy
SSSP	rmat26	1.20 ×	12%
	random26	1.1 ×	10%
	LiveJournal	1.17 ×	11%
	USA-road	1.12 ×	9%
	twitter	1.16 ×	12%
PR	rmat26	1.17 ×	5%
	random26	1.13 ×	7%
	LiveJournal	1.19 ×	7%
	USA-road	1.18 ×	6%
	twitter	1.20 ×	7%
BC	rmat26	1.11 ×	9%
	random26	1.07 ×	13%
	livejournal	1.09 ×	10%
	USA-road	1.16 ×	12%
	twitter	1.09 ×	11%
	Geomean	**1.14** ×	**9%**

Table 5.10: Effect of memory coalescing

Graphs	Speedup	Inaccuracy
rmat26	1.22 ×	12%
random26	1.06 ×	17%
LiveJournal	1.23 ×	13%
USA-road	1.28 ×	13%
twitter	1.16 ×	12%
rmat26	1.27 ×	7%
random26	1.12 ×	11%
LiveJournal	1.19 ×	7%
USA-road	1.25 ×	5%
twitter	1.17 ×	9%
rmat26	1.21 ×	14%
random26	1.13 ×	18%
LiveJournal	1.19 ×	16%
USA-road	1.24 ×	15%
twitter	1.14 ×	13%
Geomean	**1.19** ×	**12%**

Table 5.11: Effect of shared memory

Graphs	Speedup	Inaccuracy
rmat26	1.07 ×	7%
random26	1.03 ×	8%
LiveJournal	1.06 ×	7%
USA-road	1.08 ×	7%
twitter	1.05 ×	6%
rmat26	1.09 ×	5%
random26	1.03 ×	6%
LiveJournal	1.10 ×	5%
USA-road	1.07 ×	8%
twitter	1.08 ×	8%
rmat26	1.06 ×	11%
random26	1.04 ×	13%
livejournal	1.08 ×	10%
USA-road	1.10 ×	6%
twitter	1.07 ×	12%
Geomean	**1.07** ×	**8%**

Table 5.12: Effect of thread divergence

Approximate Graffix versus exact Baseline-III

set to the value which provides best results (which is different for different graphs). In particular, threshold of 0.6 performs well for power-law graphs and of 0.4 for the road-network. We observe significant performance gains (mean $1.16\times$) for several algorithm-technique pairs, with some accuracy loss (mean 10%). For SSSP, SCC and PR, the reduction in execution time (in absolute terms) is small in exchange for 10% inaccuracy. Approximations are, in general, more useful for higher-complexity algorithms, such as betweenness centrality. However, approximate computations in algorithms with lower computational complexity are useful when used repeatedly in applications that are error tolerant. For instance, in machine learning, while training a model, the distance measure used could be approximate shortest distance and it may not matter much. Now across multiple iterations during the training process, we would observe a significant reduction in total training time. So, the approximate results for SSSP, SCC, PR could be useful as well. Tables 5.7 and 5.10 show the effect of our approximate techniques for coalescing for five graphs on the algorithms in Tigr and Gunrock, respectively. We observe that the speedups achieved over Gunrock are similar to Baseline-I. The speedups achieved over Tigr are lower since Tigr implements a memory access optimization, *edge-array coalescing*, to alleviate the irregularity in memory accesses. The inaccuracies for graph–algorithm pairs are similar across all baselines, because inaccuracy is tied to the modifications in the graph's structure.

Effect of Connectedness. *Connectedness* forms the tunable knob between speedup

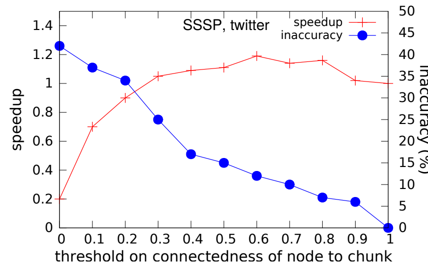


Figure 5.7: Effect of varying the threshold for node replication.

and inaccuracy. Figure 5.7 compares against Baseline-I for a fixed chunk-size of 16. For a small threshold, the speedup is low and the inaccuracy is high due to more replications.

We observe a steady increase in the speedup with increase in the threshold up to a point (0.6 in the plot), followed by a gradual decline in the performance gains. This is because the number of nodes getting replicated is enough for the combined benefits of coalesced accesses to show effect. Also, the occupancy of the *holes* is high. However, upon further increasing the threshold, only a few nodes get replicated and the number of unoccupied *holes* is large. Thus the reduced performance benefits for larger thresholds. The inaccuracy, on the other hand, gets benefited by increasing the threshold. This is due to fewer edges getting added.

Guidelines for the Threshold. The threshold on *connectedness*, for a fixed chunk size, is based on the degree distribution. The power-law graphs have some high degree nodes. Majority of such nodes may be replicated if the threshold is low. To ensure that only the nodes with a high *connectedness* are replicated, in the interest of accuracy and the graph size, the threshold is set to a fairly large value. Setting a high threshold (above 0.6) would prohibit enough replication which would hurt performance. In contrast, the node degrees in a road-network are small and largely uniform. For good hole occupancy, the threshold is chosen to be small (below 0.5).

Comparison with Existing Graph Reordering Techniques

We do not compare our graph reordering technique with the the other existing techniques. While several graph reordering techniques have been proposed in the literature, their source codes are not readily available. Thus, comparing our graph reordering

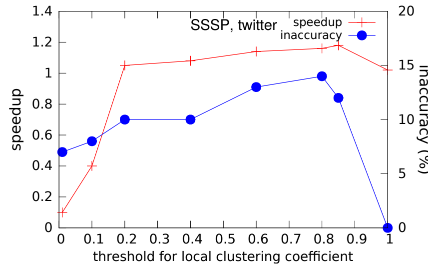


Figure 5.8: Effect of varying the threshold for clustering-coefficient

technique with the previously proposed ones would require implementing those from scratch. Further, we are not aware of any works that compare and contrast the different graph reordering techniques. So, for a fair comparison, we would need to compare our technique with all the existing graph reordering techniques. Thus, such a comparison is substantial work and a project in itself.

5.4.2 Effect of Memory Latency

Table 5.5 shows the effect of using shared memory on the five algorithms for five graphs from Baseline-I. We observe performance gains for various algorithm-technique pairs. The threshold for clustering coefficient (CC) is set to a different value for each of the graphs for obtaining decent accuracy and speedup. Tables 5.8 and 5.11 show the effect of our approximate techniques for reducing memory latency for five graphs on the algorithms in Tigr and Gunrock, respectively. The speedups achieved over Gunrock and Tigr are similar (1.19×) to those achieved over Baseline-I. The inaccuracies for graph–algorithm pairs are similar (11%) across all baselines.

Effect of CC Threshold. Figure 5.8 plots the speedup and inaccuracy, w.r.t. Baseline-I, with varying thresholds for clustering coefficient. There is a consistent increase in speedup with increase in the threshold, since an increased threshold implies well-connected subgraphs occupying the shared memory, thereby benefiting from its low memory latency. However, for threshold ~ 1, fewer nodes are moved to shared memory, resulting in diminished gains.

As the threshold is increased, the inaccuracy first rises and later reduces. The rise in inaccuracy is because it exposes more nodes whose CC can be increased by addition

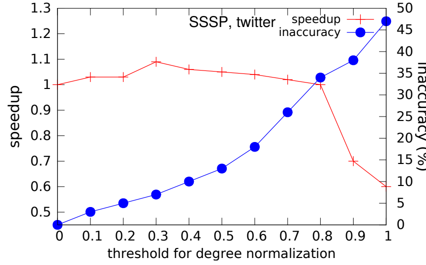


Figure 5.9: Effect of varying the threshold for degree normalization.

of edges using the scheme presented in Section 5.2. However, after a point (threshold = 0.8), the inaccuracy reduces as the candidate nodes for processing inside shared memory have better connectivity; so we add fewer edges.

Guidelines for the Threshold. The choice of the threshold for CC is based on the graph's average CC and degree distribution. Since the focus is on finding nodes that are part of a well-connected cluster, the threshold must be set to a high value for all graphs.

5.4.3 Effect of Thread Divergence

Table 5.6 shows the effect of reducing thread divergence for five graphs on the five algorithms from Baseline-I. We obtain minor performance improvements for various algorithm-technique pairs, with reasonably high accuracy in most cases. Tables 5.9 and 5.12 show the effect of our approximate techniques for reducing thread divergence for five graphs on the algorithms in Tigr and Gunrock, respectively. We observe that the speedups achieved over Gunrock are similar to Baseline-I. Tigr already implements node splitting transformations for reducing thread-divergence. Therefore, speedups achieved over Tigr are lower. The inaccuracies for graph–algorithm pairs are similar across all baselines.

Effect of Degree Similarity. To measure the variation in node degrees, we define:

$$\text{degreeSim}_{\text{node}} \triangleq \left(1 - \frac{\text{node degree}}{\text{maximum degree of warp nodes}}\right)$$

degreeSim identifies the deficit in the degree of a node compared to other nodes assigned to the same warp. Figure 5.9 plots the speedup and inaccuracy, w.r.t. Baseline-I,

with varying thresholds for $\text{degreeSim}_{\text{node}}$. The node degree is made 85% of the warp's max-degree. As we increase the threshold, we allow more edges. We observe that the speedup increases with increase in threshold up to a point (0.3 in Figure 5.9) after which it begins to drop. This is because when limited edges are added, the performance improves due to the combined effect of reduced thread divergence and faster propagation. Performance gains drop with further increase in threshold. This is because the size of the graph increases due to addition of considerable number of edges, which begins to dominate. Inaccuracy increases monotonically with increase in threshold since a higher threshold allows for addition of more edges.

Guidelines for the Threshold. For obtaining reasonable accuracy and speedup, the threshold on *degreeSim* is set based on the degree distribution. If on an average, the mean node degree in a bucket is quite low, or if it is closer to the maximum node degree than to the minimum node degree in the bucket, then the threshold should be set to a low value (below 0.4). Picking the threshold this way ensures addition of limited extra edges as we normalize the degree of only relatively-large-degree nodes in a warp.

5.4.4 Preprocessing Overhead

The preprocessing overheads for the approximate techniques targeting memory coalescing, memory latency, and thread divergence are presented in Table 5.13. We observe that the mean times for transforming the graphs in our test-suite for improving coalescing, reducing memory latency and reducing thread divergence are 182s, 233s, and 58s respectively. This is a one-time offline cost. The execution of complex algorithms such as those for BC and MST consume more time than preprocessing. For the simpler algorithms such as those for SSSP, SCC, and PR, the preprocessing time is significantly higher. This extra preprocessing cost may be amortized over several runs of multiple algorithms. The corresponding mean extra space consumed by the transformed graphs (w.r.t. the original graph) is 8%, 5.6%, and 2.3% respectively for the three techniques, which is practically not high.

Tables 5.14 – 5.16 present the total reduction in the end-to-end execution time for MST and BC using Graffix techniques w.r.t. Baseline-I. We observe that for MST and

Technique	Graph	Preprocessing overhead	
		Time (sec)	Additional space
improving coalescing	rmat26	18	9%
	random26	21	11%
	LiveJournal	8	6%
	USA-road	11	8%
	twitter	30	6%
reducing latency	rmat26	23	5%
	random26	27	8%
	LiveJournal	9	5%
	USA-road	13	4%
	twitter	46	7%
reducing thread-divergence	rmat26	6	2%
	random26	7	3%
	LiveJournal	4	2%
	USA-road	2	1.5%
	twitter	9	4%

Table 5.13: Preprocessing overhead

	Graphs	Reduction in end-to-end time
MST	rmat26	15.05%
	random26	11.29%
	LiveJournal	12.04%
	USA-road	5.28%
	twitter	14.25%
BC	rmat26	14.41%
	random26	10.55%
	LiveJournal	12.57%
	USA-road	15.42%
	twitter	12.14%

Table 5.14: Effect of memory coalescing

Graphs	Reduction in end-to-end time
rmat26	17.77%
random26	8.82%
LiveJournal	14.99%
USA-road	0.81%
twitter	13.37%
rmat26	19.20%
random26	11.30%
LiveJournal	16.83%
USA-road	19.99%
twitter	14.31%

Table 5.15: Effect of shared memory

Graphs	Reduction in end-to-end time
rmat26	0.046%
random26	0.019%
LiveJournal	0.064%
USA-road	0.058%
twitter	0.047%
rmat26	0.098%
random26	0.047%
LiveJournal	0.080%
USA-road	0.106%
twitter	0.056%

Table 5.16: Effect of thread divergence

Reduction in total end-to-end time of Graffix w.r.t. exact Baseline-I

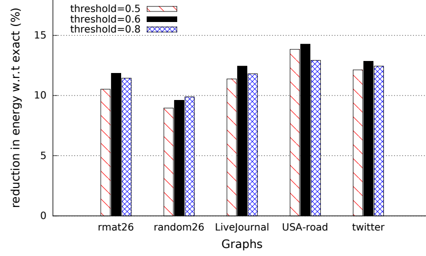


Figure 5.10: Effect of coalescing on energy consumption for BC (higher is better).

BC, even if we perform the graph transformations once for every input graph, there is a reduction in the total time w.r.t. the exact version.

5.4.5 Impact of Approximations on Energy

For measuring the energy consumption accurately, the algorithm needs to run for long. Thus, we choose to study the effect of energy consumption on BC. Figure 5.10 shows the reduction in the energy drawn during the execution of BC using Graffix's technique for improving memory coalescing. We show the decrease in energy for different thresholds of *connectedness* for *node-replication*. We observe that the amount of energy drawn during execution with approximations is lower than that for the exact version. The reduction in energy with approximations can be attributed to (i) reduction in power (ii) reduction in total execution time. Reduction in power primarily results from the coalescing of multiple accesses into fewer memory transactions. We measure the power consumption of the GPU and its associated circuitry (e.g. memory), in milliwatts, while the GPU kernel is running. We measure the power using the nvmlDeviceGetPowerUsage() API provided by the NVML library. Reduction in total execution time results from improved coalescing and the approximation technique discussed in Section 5.1.1. Hence, the mean energy consumption during execution, computed as $\left(\text{mean power} \times \text{mean execution time}\right)$ is lower for the approximate version.

5.5 Summary

We proposed graph transformation techniques for efficient graph processing on GPUs using approximate computing. Our techniques improve memory coalescing, memory latency, and thread-divergence by graph reordering and graph transformation. Using a suite of five popular graph algorithms and five large graphs, we illustrated that our proposed techniques reduce execution times of parallel implementations of graph algorithms appreciably by incurring a small loss in the quality of the solution.

CHAPTER 6

Faster Estimation of Top-k Betweenness Centrality Vertices on Heterogeneous Architectures

Betweenness centrality (BC) is a crucial centrality metric in graphs and networks that measures the significance of a vertex. BC(v) is calculated using the number of shortest paths in the graph passing through vertex v in the graph. It is used in a multitude of applications such as detecting communities in social and biological networks (Girvan and Newman (2002)), targeted advertising (Kim *et al.* (2012)), analysis of disease spreading (Liljeros *et al.* (2001)), and identifying criminal networks (Coffman *et al.* (2004)), among others. The state-of-the-art Brandes' algorithm (Brandes (2001)) computes the exact BC values for all nodes in a graph $G = (V, E)$ in time $\mathcal{O}(|V||E|)$ for unweighted graphs, and time $\mathcal{O}(|V||E| + |V|^2 \log |V|)$ for graphs having positive weights. As suggested by its complexity, computation of BC is quite time-consuming even on graphs of moderate sizes, having hundreds of thousands of nodes and edges.

To make BC computations scalable, Brandes' algorithm has been successfully parallelized on multi-core CPUs, many-core GPUs, and distributed systems (Madduri *et al.* (2009); McLaughlin and Bader (2014); Prountzos and Pingali (2013); Solomonik *et al.* (2017); Hoang *et al.* (2019)). Yet, the cost of BC computation is excessive on modern real-world graphs with millions of nodes and tens of millions of edges. For example, the exact vertex-BC computation on the undirected graph *liveJournal* (having ∼4.8M nodes and ∼69M edges) using a parallel implementation of Brandes' algorithm on a GPU takes several days to complete. Moreover, often applications are interested in the relative ranking of the vertices according to their BC scores, rather than their actual BC values. In addition, several applications demand identifying nodes with the highest BC values. Hence, an estimate of the top-k BC vertices is sufficiently informative in such cases.

We present ParTBC, a bouquet of novel techniques for speeding up the estimation of top-k vertices with highest BC in a graph, using approximate computing in conjunc-

tion with parallelization. We propose to compute approximate BC values of vertices, such that the relative ordering of the vertices is maintained. Further, we present a novel graph reordering scheme to make the graph layout more *regular* to enable efficient coalesced access of data in parallel Brandes' algorithm on GPU, improving performance. The graph-layout is also beneficial to other vertex-centric parallel graph algorithms. ParTBC is the first system that combines parallelization and approximate computing to estimate the top-k BC vertices in a graph.

The chapter is organized as follows. Section 6.1 presents a formal description of the problem of top-k BC-vertex computation, and also discusses Brandes' algorithm, which forms the basis for the parallelization and the approximate techniques. Section 6.2 presents our proposed scheme at a high-level. Section 6.3 presents the parallelization strategy and our proposal for a modified graph layout. Section 6.4 presents our techniques for estimating top-k BC vertices. Section 6.5 quantitatively evaluates the proposals and analyzes results.

6.1 Problem Statement and Preliminaries

Problem Statement. Given an undirected, unweighted graph $G(V, E)$ and a positive integer $k \leq |V|$, find a set of k vertices, S_k, where $S_k \subseteq V$ and S_k contains vertices having the highest BC values in G. In this work, we determine the set S_k faster with a small error in set membership.

Betweenness Centrality. The state-of-the-art Brandes' algorithm for computing vertex betweeness centrality is described in detail in Section 2.4.

Property 1: In Brandes' algorithm, BC of a vertex does not change in the iteration in which it is the source.

Proof: For each source, s, we compute the BFS DAG rooted at s. Now, s will always lie at one end of the shortest paths to all other nodes, from s. So, s cannot lie on the shortest path between any two other nodes, as all the edges have unit weight. Thus, in the iteration in which the node is a source, its BC does not change. The same is captured

in equation 2.3. □

Observation-1: Our experiments show that high BC nodes are usually either (i) the high degree nodes, or (ii) those low degree nodes that lie on the paths connecting two or more large well-connected clusters.

Justification: High degree nodes are connected to a large number of nodes and thus lie on a large number of point-to-point paths. Consequently, these lie on a large number of point-to-point shortest paths. Further, those low degree nodes that connect large clusters in a graph, lie on the shortest paths between the nodes lying in separate clusters. Thus such nodes too have high BC.

6.2 ParTBC's Approach

Algorithm 9 Approximate top-k computation

Input: An undirected, unweighted graph $G(V, E)$
Input: k
Input: desired accuracy ($\leq 100\%$)
Output: top-k betweenness centrality vertices

```
1: bc[v] = 0      ∀v ∈ V                                  ▷ initialization
   ▷ Phase-I
2: G′(V,E) = graphReordering(G)          ▷ vertex renumbering;   G′ ≅ G
   ▷ Phase-II
3: nextIter = true
4: while nextIter do
5:     nextIter = false
6:     s = getSource()                                    ▷ pick source vertex
   ▷ Forward Pass: form BFS DAG D
7:     for all v : Node ∈ G′ do
8:         compute σ_sv
9:         compute pred(s, v)
   Let D be the DAG formed by the forward pass
   ▷ Backward Pass: backward traverse DAG D
10:    for all v : Node ∈ D do
11:        compute δ_s(v)
12:        bc(v) += δ_s(v)
13:    if stopping criteria not met then
14:        nextIter = true;
   ▷ Reset graph attributes
15:    for all (u → v) ∈ E do
16:        reset(u → v)
```

Algorithm 9 outlines the approach adopted in ParTBC. The computation proceeds in two phases. Phase-I performs graph reordering by renumbering the vertices of the graph to bring together in memory the data of those nodes that are likely to be accessed

in tandem in parallel Brandes' algorithm on GPU (Section 6.3). The reordered graph is the input to Phase-II. In Phase-II, BC computation happens in parallel and the source vertices are picked (Line 6) using the techniques described in Section 6.4. The algorithm terminates when the stopping condition is satisfied, which is calculated online based on the desired accuracy in the set of top-k vertices.

6.3 Parallelization and Graph Layout

6.3.1 Parallelization Strategy

Brandes' algorithm has been shown to be parallelized mainly in two ways: outer parallel and inner parallel (Prountzos and Pingali (2013); Jin *et al.* (2010); Bader and Madduri (2006)) In outer parallel, multiple source vertices are processed in parallel (line 2 of Algorithm 7), but the forward and the backward passes are executed sequentially by each thread. Thus, the contribution of each source to BC values of other vertices can be computed by the thread assigned to that source. The final computation of $bc(v)$ involves a *reduction* of the contribution of each of the sources. In this scheme, every outer loop iteration requires its own storage, leading to a substantial space overhead, of the order of $O(n^2)$ (Prountzos and Pingali (2013)).

In inner parallel scheme, on the other hand, each source is processed sequentially, but each of the computation steps (lines 3, 7 in Algorithm 7) for a single source are executed in parallel. A crucial advantage of this approach is its space-efficiency, as DAG (and other transient data) corresponding to only one source need to be maintained at a time. Therefore, such an approach can be used for large graphs (Prountzos and Pingali (2013)).

6.3.2 Graph Layout

We use the popular Compressed Sparse Row (CSR) storage format to represent the graph (Figure 6.1). Further, we use the vertex-centric model of parallelization, wherein threads are assigned to vertices. Our parallel implementation of BC uses three ker-

nels – one for the forward pass and two as part the backward pass (one to compute δ values and another to accumulate BC values). We employ both *topology-driven* and *data-driven* implementations of the forward pass kernel (Pingali *et al.* (2011)). Our implementation automatically chooses one of the two implementations based on the *skewness* (third-central moment) of the vertex degree distribution. For graphs with skewed degree distribution ($|$coeff. of skewness$| > 0.05$), such as social networks, topology-driven implementation is chosen, while data-driven implementation is used for graphs with uniform degree ($|$coeff. of skewness$| \leq 0.02$), such as road networks.

edges

3	4	13	11	14	6	0	10	12	14	0	11	8	11	14	2	9	12	13	10	12	5	12	6	3	7	1	4	5	3	6	7	8	14	0	6	1	3	5	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

offset

0	3	5	6	10	12	15	19	21	23	24	26	29	34	36	40
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

node attributes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 6.1: Original graph G and its CSR representation

edges

1	2	0	3	5	0	4	5	6	7	1	8	11	2	5	9	12	1	2	4	11	2	12	2	10	13	14	3	9	4	8	10	7	9	3	5	4	6	7	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

offset

0	2	5	10	13	17	21	23	27	29	32	34	36	38	39	40
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

node attributes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 6.2: Graph G with renumbered vertices

6.3.3 Improved Graph Layout

A natural way to compute top-k BC vertices faster is to improve the performance of the exact parallel implementation of Brandes' algorithm. With this motivation, we propose a scheme to modify the graph layout to make it more *structured* to make it amenable to GPU-based processing. We intend to improve the memory coalescing and better utilize GPU's high memory bandwidth.

The forward-pass of Brandes' algorithm involves breadth-first-search (BFS) traversal on the graph from a designated source node in every iteration. In our parallelization strategy of the forward-pass, we process the vertices in a level-synchronous fashion, and the thread assigned to a node updates the attributes of its neighbors. Reordering of vertices is shown to be effective in improving the spatial locality of vertices by assigning consecutive id's to those that are likely to be accessed in tandem (Nodehi Sabet *et al.* (2018); Balaji and Lucia (2019); Liu and Sherman (1976)). To improve vertex-centric processing, ParTBC proposes a novel vertex-renumbering scheme to modify the graph layout such that the connected nodes and their data are together for GPU-based processing. For instance, in Figure 6.1, assume the warp-size to be 4. The vertices 4 – 7 are assigned to threads having the same id as the node. With vertex-centric processing, the warp-threads will access the attributes of the first neighbor of the respective vertices concurrently, and so on. Hence, the warp threads will access the locations 0, 8, 2 and 10 in the node attributes array together. Further assume that the accesses to a chunk of 4 words can be coalesced. Clearly, the accesses to the destination vertices' {0, 8, 2, 10} data in the *node attributes* array are not coalesced since these lie in three separate 4-word chunk. We renumber the vertices such that the vertices to be accessed by the warp-threads are assigned nearby id's; this results in improved coalescing. The vertex-renumbering is performed once at the time of loading the graph. The following is the renumbering scheme (for Line 2 in Algorithm 9):

We pick a lowest degree neighbor of a vertex having the highest degree and perform a BFS traversal on the graph, to obtain a BFS tree. The vertices at the same level in the BFS tree are assigned id's in a round-robin fashion: the first neighbor of each of the parents from the previous level is assigned a new id followed by the renumbering of all the second-neighbors, and so on. The foregoing renumbering scheme ensures

that the threads of a warp access nearby locations while accessing the attributes of the destination vertices in the node attribute array. Since the graph is undirected, the renumbering helps improve the coalescing in every outer iteration of Brandes' algorithm. The choice of the source of this BFS traversal helps on two accounts: (i) as we will see, we are likely to pick a node with a low degree as a source vertex in Brandes' algorithm (Section 6.4). So, picking a low-degree neighbor of the high-degree node as a source ensures near-perfect coalesced accesses in an iteration of the Brandes' algorithm. (ii) a high-degree node is likely to be visited a higher number of times overall iterations in BC computation (Sections 6.4). So picking a neighbor of a high degree node is a better choice than starting at an arbitrary node.

For example, in the graph G from Figure 6.1, vertex 12 has the highest degree. We perform BFS from vertex 8, which is a lowest degree neighbor of 12. Vertex 8 is at level zero, vertices 5 and 12 are at level 1, vertices 3, 6, 7, 11, 14 are at level 2, while other vertices are at level 3. Figure 6.2 shows the graph with vertices renumbered, along with its memory layout. The attributes array for the destination vertices has more coalesced accesses in the renumbered graph.

The renumbering-scheme is applicable, in general, to all graph algorithms that are implemented using the vertex-centric approach and node values are propagated by updating neighbors' values through outgoing edges.

6.4 Techniques for Fast BC estimation

We present a systematic study of the use of approximate computing in the computation of top-k betweenness centrality vertices. Our key observation is that not all sources in Brandes' algorithm (line 2, Algorithm 7) contribute equally to the BC values. We present a host of techniques to identify vertices to be picked as sources that enable *enough* contribution to the BC of the vertices early to facilitate quicker identification of top-k BC vertices. Our strategy is to identify those vertices which would *eventually* have high BC values, in the early iterations of the Brandes' algorithm.

Based on *Property-1* and *Observation-1* from Section 6.1 we hypothesize that in

order to impart a high share of their BC values early to the eventual high BC nodes, we should preferentially pick the low and moderate degree nodes as sources. The intuition is that such a choice of sources would increase the BC score of a high degree node and not of the low and the moderate degree neighbors of it. This would widen the gap between the eventual low and high BC vertices, thus allowing us to decide the top-k nodes in the early iterations. For our purpose, we categorize the nodes into low-order, moderate-order and high-order vertices. In the list of vertices sorted in ascending order by node degree, the first 25% are the low-order vertices, the next 50% are moderate-order vertices and the remaining 25% are the high-order vertices. Our experiments and analysis revealed that the number of iterations required for termination are indeed fewer when preferentially picking low- and moderate-order neighbors of high order nodes as sources. So, the high-level idea is to devise schemes that enable us to pick such nodes as sources in the early iterations. By filtering out the source vertices carefully, performance gains are substantial due to a sizeable reduction in the total work done.

Termination of execution. For each of the techniques, there are two ways to specify the termination of execution. One, we may specify the number of iterations as a percentage ($\alpha\%$) of the total number of iterations. Two, we may specify an online stopping criterion to get the desired accuracy. The second approach is preferable since it allows us to control the performance–accuracy tradeoff.

For the latter approach, we need to define a metric that captures the quality of the top-k nodes reported. A desired metric for computing the output quality of top-k betweenness centrality nodes in a graph is the set difference between the set of exact top-k nodes and those computed using an approximate technique. This quality metric requires knowing the ground-truth (i.e., the exact top-k nodes), which would be available only upon running Brandes' algorithm to completion. Hence, we require a metric that can be computed at each iteration of Brandes' algorithm rather than at the end of the computation, to help us decide if we have reached the desired accuracy and thus terminate the execution.

A plausible proxy for the above metric is: tracking the node having the k-th highest BC value in every iteration and checking if the node having the k-th highest BC value has stabilized (i.e., it is unchanged for all remaining iterations).

Lemma 1: If the k-th highest BC node does not change across iterations, then the set of top-k BC nodes also remains unchanged.

Proof: Consider two consecutive iterations of the outermost loop in Brandes' algorithm: i and $i+1$, such that $i < i+1 \leq |V|$. Note that BC of a vertex monotonically increases in every iteration. Let, the set of vertices, V be partitioned into two sets, S and S'. S contains the top-k ($k \leq |V|$) BC nodes and $S' = V \setminus S$. At each iteration, we maintain the invariant that the cardinality of S is k and that it holds the top-k BC nodes. Further, let's define a sequence on the elements of set S: $\pi_s = (v_1, v_2, v_3, \ldots, v_k)$ such that BC(v_1) $\geq$ BC(v_2) $\geq$ BC(v_3) $\ldots \geq$ BC(v_k). Let, $v_k \in S$ be the last element in the sequence π_s, at the end of iteration i. Now, suppose at the end of iteration $i+1$, a node from S' moves to S (due to increase in its BC value), then a node from S must move to S' to maintain the invariant. Further, the node that moves from S' to S must have BC greater than or equal to BC(v_k). So, the element to be displaced from S must have BC equal to BC(v_k). Let us assume the last element in sequence π_s, i.e. v_k, will be displaced in the event of multiple nodes having the same BC value as v_k. Thus, if the element v_k is the same after iterations i and $i+1$, it implies that the rest of the elements in the set S are also unaltered. The result holds for any two iterations i and j s.t. $i < j \leq |V|$. $\square$

There are two issues with using the aforementioned proxy metric: (1) it requires determining the k-th highest BC node after every iteration (which has a time complexity of $\mathcal{O}(|V|)$), thus introducing significant time overhead. (2) the node with the k-th highest BC needs to be tracked for all iterations in order to establish that the position of k-th highest BC node is unchanged — this makes this metric unsuitable for online error estimation.

To design a pragmatic scheme for accurate estimation of online error, we draw on the observation that by the iteration when the ranks of highest *few* BC nodes are settled, the other high BC nodes also get a sufficiently large share of their respective BC values. Thus tracking only a fixed number of highest BC nodes may suffice.

Therefore, in each iteration of Brandes' algorithm, we maintain the set, S_t, of top-t ($t \leq k$) vertices. We track the number of successive iterations for which the set S_t remains unchanged. We terminate the execution when this count reaches C_t. The

choices of t and C_t depend on the techniques used for picking the source nodes (Sections 6.4.1 through 6.4.5) and the type of the input graph. This scheme enables us to accurately estimate on-the-fly when the error in the top-k becomes substantially small.

Our experiments showed that for the techniques in Sections 6.4.1–6.4.3, $40 \leq t \leq 50$ and $5 \leq C_t \leq 10$ results in desirable speedups and accuracy. On the other hand, for the techniques presented in Sections 6.4.4–6.4.5, $5 \leq t \leq 10$ and $3 \leq C_t \leq 5$ are sufficient to achieve similar accuracy as with the previous techniques. This can be attributed to the fact that the choice of sources using the latter techniques imparts a larger share of the respective BC values to the vertices quicker. Hence stabilization of the ranks of a few high BC nodes for a few iterations indicates that the ranks of the other nodes are also stabilized, with a good chance.

We discuss the techniques proposed by ParTBC below.

6.4.1 Random Selection of Source Vertices (Random)

From a uniformly random permutation of the vertices, we select a subset, guided by the stopping criterion. With this technique, we pick source vertices with varied connectivity and characteristics. In real-world scale-free graphs (that have many low degree nodes and a few high degree nodes), the probability of picking the high degree nodes as source early is less due to their number. Hence, random selection of nodes naturally leads to the selection of low and moderate degree sources. The nodes so picked include a fair number of non-high degree neighbors of high degree nodes. Hence, the BC scores of the nodes acquired in the early iterations cause the relative BC scores of the graph vertices to be indicative of their relative exact BC values, leading to high accuracy in top-k computation with a small number of iterations. However, we need a large value of t for getting high accuracy in fewer outerloop iterations. High value of t requires the computation of the tth-largest BC node (*Lemma 1*) in each iteration, adding up to high overhead.

6.4.2 Node Selection in Ascending Degree Order (Ascending)

A natural order is based on vertex degree: ascending and descending. We pick the vertices as sources in Brandes' algorithm in that order. The overhead of sorting of vertices ($\mathcal{O}(|V| \log |V|)$), which is a one-time operation, is a tiny fraction of the overall computation time of exact BC scores. We observe that in several real-world graphs, the low degree vertices are connected to other low degree vertices. So, the DAG formed by selecting a low degree vertex as the source has more levels as compared to the one resulting from picking a high degree vertex as the source; the DAG formed also has, on average, a small number of nodes at each level. However, when the sources are selected in ascending order of degrees, the vertices with high BC may not get enough contribution in the early iterations. This happens because the high degree vertices appear towards the bottom of the DAG, and thus, not many vertices are reachable through them; this results in reducing the dependency contribution. Further, in real-world graphs, low degree nodes are connected to other low degree nodes, so few low-order nodes are neighbors of high-order nodes. Hence, a larger number of iterations are required to determine the top-k accurately. Similar to the *Random* technique, we need a large value of t, adding a significant execution overhead.

6.4.3 Node Selection in Descending Degree Order (Descending)

In this technique, we arrange the source vertices in decreasing order by degree. Generally, the high degree vertices are found to be connected to other high degree vertices. Hence, the DAG formed from such a vertex will have fewer levels and a high average number of nodes at each level. Since the sources selected are in descending order, all the vertices with high BC receive large BC contributions in the early iterations because the high degree vertices tend to appear at the top of the DAG which results in more number of vertices being reachable via them. The moderate- and low-order neighbors of high-order nodes are picked sooner than when following ascending order for selecting sources. So, fewer iterations are required to estimate the top-k with high accuracy. However, similar to *Ascending*, computing the stopping criteria has a high overhead.

6.4.4 Selecting Low-Degree Neighbors of High Degree Vertices

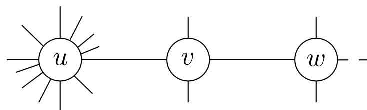


Figure 6.3: Neighbors of high-degree vertices

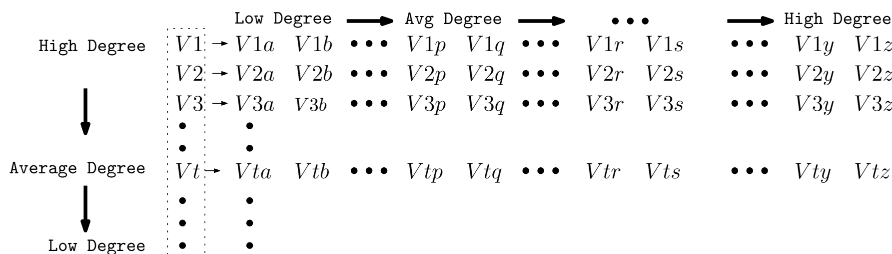


Figure 6.4: Layout for techniques

From *Observation-1*, high degree vertices and the cut-vertices connecting large clusters are more likely to have large BC values. We also note that in Brandes' algorithm, when we pick a node as a source, the BC values increase more for those vertices that are at the initial levels of the DAG and have more vertices reachable from them. Thus, based on *Property-1*, to reduce the *polluting* of the vertices (increase in BC value of an unimportant node by as much as an important node, in an iteration), it is beneficial to pick the immediate neighbors of high degree vertices as sources (Figure 6.3). Further, among the immediate neighbors, we pick the low degree neighbors as sources first, with the assumption that the low degree vertices are likely to have lower BC than the high degree vertices, in general. Consider the scenario in Figure 6.3. Suppose v and w are low degree 1-hop and 2-hop neighbors, respectively, of a high degree node, u. In this case, we prefer to pick the node v over w as the source. Picking v (1-hop neighbor of u) as the source would increase the BC value of u more than that of w in that iteration and widen the gap in their BC values, enabling computation of top-k in fewer iterations. With this technique, the vertices connecting large clusters also get a high BC value, as desired. When a source is selected, the DAG formed has such connecting vertices as the dominator of many other vertices, and thus, they receive enough BC contribution.

The technique can be combined with *Ascending* and *Descending*. We consider only the high-order vertices and sort them in descending order by degree. Further, the neighbors of each of these are sorted in ascending order by degree. Additionally, nodes

having equal degrees are arranged in ascending order by node id. We then select the neighbors in a round-robin fashion. Round-robin means that we select the unpicked lowest-degree neighbor of the highest degree vertex, followed by the lowest-degree neighbor of the second-highest vertex, that is not already selected, and so on. For example, in Figure 6.4, the vertices selected as sources are $v1a$, $v2a$, $\ldots$, $v3a$, $v1b$, $v2b$, and so on, in that sequence.

A caveat in the round-robin selection of source vertices is that in real-world graphs, high-degree vertices are often neighbors of other high-degree vertices, and this scheme may end up selecting the high-degree neighbors (instead of the low-degree neighbors) of high-degree vertices. To address this issue, we select only the low-order neighbors.

Interestingly, setting the threshold on the neighbors' degrees prohibits selecting those high-order vertices as sources that are the neighbors of other high-order vertices. Since a node is chosen as source only once, this prevents the selection of a vertex with high degree over a vertex with low degree.

The threshold for high-order, moderate-order and low-order vertices can be tuned to control the accuracy in the top-k vertices, and the resulting speedup. We call this technique Restricted-Round-Robin (RRR). Additionally, the number of low- and moderate-order neighbors of high-order nodes picked as source using this technique are substantially high by design. So, this method provides improved accuracy compared to earlier techniques, but *Random*, for similar speedups. Further, unlike in the prior techniques, the stopping condition can be faithfully computed by tracking only upto 5 highest BC nodes across iterations, which is computable in $\mathcal{O}(1)$.

6.4.5 Dynamic Selection of Source

In this technique, we also take into account the BC scores of the vertices up to that iteration to determine the source vertex for the next iteration. We observe that once the vertices get a *healthy* share of their respective final BC scores, the vertices with low BC are likely to continue having a low relative BC score in the subsequent iterations. We exploit this observation to select that so far unpicked vertex as the source for the next iteration, which has the least BC value up to that iteration. Again, based on *Property-1*,

we further try to minimize the BC value of that particular vertex by selecting it as a source. By minimizing the BC value of the lowest BC node, we increase all the other remaining vertices' BC values and not just of a select few. So by selecting different sources dynamically, all the vertices having moderately high or high BC values get contributions relative to the vertices' eventual BC values by selecting the low BC vertices as sources, and this also widens the gap between the eventual high BC and low BC vertices.

Now, to provide the vertices sufficient representative share of their BC values before going for the dynamic scheme, we execute the initial few iterations of the Brandes' algorithm. The choice of the source vertices for these initial iterations is crucial since we want the vertices to have a sufficient share of the BC values as quickly as possible. In the entire execution, no node is picked more than once. We discuss the heuristics for selecting these initial sources. We empirically found that selecting 5% source vertices is reasonable for contributing a good share of BC values to each vertex (and also improves execution time).

Descending (Dyn)

We sort the vertices in descending order of their degrees. We then pick the top 5% vertices from this sorted list as the initial set of source vertices. After the 5% outer loop iterations, in the subsequent iterations, the vertex picked as the source is a node with the least BC score up to that iteration that has not been picked already.

Dynamic Round Robin (DynRR)

We observed that with the selection of fewer source vertices, *RRR* achieves better results than Dyn. However, if a large number of sources are selected, Dyn works better and has smaller error compared to *RRR* for the same number of sources. This suggests that picking the initial set of source vertices using *RRR* is beneficial.

For the first 5% iterations, we select the vertices as sources in a round-robin fashion, selecting only those that have a degree less than a threshold (which is set to average node degree). Further, for the selection of sources in the iterations following the first 5%, we

consider neighbors of vertices having degree greater than the average degree. From this subset of vertices, the one having the least BC value up to that iteration is selected as the source for the next iteration. This helps limit the search space further to only those vertices which, when picked as source, contribute significantly to boosting the values of the important vertices. Selecting the sources based on BC values computed until then increases the BC values of their neighbors (the high degree vertices), which improves accuracy.

DynRR is found to perform the *best* among all the techniques, that is, it either has the lowest error for the same number of source vertices compared to other techniques, or it achieves better speedup than the other techniques for similar accuracy. It works well for all types of graphs and produces uniform results for all values of k.

Additionally, the number of low- and moderate-order neighbors of high-order nodes picked as the source using Dyn and DynRR technique is substantially high by design. So, these methods provide improved accuracy and speedups compared to earlier techniques. Further, the stopping condition requires us to keep track of only up to 5 highest BC nodes across iterations, which is computable in $\mathcal{O}(1)$. Hence, the overhead of computing the stopping criteria is negligible at each iteration.

6.5 Experimental Evaluation

We evaluate the performance and effectiveness of ParTBC's techniques for estimating top-k BC vertices.

Machine Configuration. We use the same machine as described in Section 4.4.

Input Graphs. We use input graphs (Table 6.1) from SNAP (Leskovec and Sosič (2014)) and KONECT (Kunegis (2017)), with different characteristics, to study the efficacy of our approach. These include social networks (such as Pokec) having small-world property, road networks (such as San Francisco) having large diameters, RMAT graphs which are synthetically generated scale-free graphs (Madduri and Bader (2006)), and random graphs which do not exhibit any specific structure.

Note: Determining the exact top-k BC-vertices (for establishing the ground-truth) re-

Graph	$\|V\|$	$\|E\|$	Graph type
fb-Friendships (FB)	63,731	817,035	Facebook friendship graph
soc-Pokec (SP)	1,632,803	30,622,564	Online social network
loc-Gowalla (LG)	196,591	950,327	Location-based social network
roadnetSF (RNSF)	174,424	221,802	San Francisco road network
usroad48 (RNUS)	102,615	147,656	Continental US road network
rmat17 (RMT)	130,977	2,091,451	R-MAT using GTgraph
random17 (RNM)	131,072	2,096,902	Random graph using GTgraph

Table 6.1: Input graphs

Graph	**Time (sec)**		**Speedup**	**Graph reordering**
	NVR	**VR**	**(NVR/VR)**	**Time(sec)**
fb-Friendships	1020	836	1.22×	5
soc-Pokec	214578	174453	1.23×	32
loc-Gowalla	26760	22677	1.18×	9
roadnetSF	2900	2566	1.13×	5
usroad48	3810	3371	1.13×	4
rmat17	1550	1302	1.19×	6
random17	616	592	1.04×	7

Table 6.2: Effect of vertex numbering on exact version
(NVR: no vertex renumbering, VR: with vertex renumbering)

quires computing the exact BC values of the vertices. Given the high time complexity of betweenness centrality computation, the exact computation takes very long (multiple days) on large graphs. Thus, we chose relatively small-sized graphs for our evaluation.

Baselines. We use two baselines to evaluate our techniques. First, we compare the performance of ParTBC against the exact parallel Brandes' algorithm on GPU for computing top-k vertices. Second, we compare the performance of ParTBC with the approximation algorithm in ABRA (Riondato and Upfal (2018)), which is the state-of-the-art in approximate BC top-k computation.

Effect of graph reordering. The execution times for the exact BC computation (which dominates top-k computation) on the graphs for Baseline-I are presented in Table 6.2. In Table 6.2 we report the time taken in the exact BC computation with and without the graph reordering (Phase-I of Algorithm 9). Note that the graph reordering technique is independent of Phase-II of Algorithm 9 and depends only on the input graph. We also

Graph	**Global Memory Load Efficiency (%)**	
	Without Vertex Renumbering	**With Vertex Renumbering**
fb-Friendships	19.5	59.4
soc-Pokec	23.3	70.7
loc-Gowalla	24.2	64.6
roadnetSF	25.2	66.3
usroad48	26.3	57.8
rmat17	22.1	63.5
random17	13.8	39.2

Table 6.3: Effect of vertex renumbering on global memory coalescing

Graph	Speedup w.r.t.		Speedup breakdown (w.r.t. exact parallel)	
	Exact Parallel	ABRA	VR	DynRR
fb-Friendships	2.80×	4.28×	1.22×	2.29×
soc-Pokec	2.76×	4.31×	1.20×	2.30×
loc-Gowalla	2.48×	4.16×	1.18×	2.10×
roadnetSF	2.71×	4.72×	1.13×	2.40×
usroad48	2.68×	4.63×	1.13×	2.37×
rmat17	2.65×	4.18×	1.19×	2.22×
random17	1.67×	1.92×	1.04×	1.61×
geomean	**2.5**×	**3.88**×	**1.15**×	**2.17**×

Table 6.4: Performance of ParTBC w.r.t. exact parallel Brandes' algorithm and ABRA (VR: vertex renumbering) Error $\sim 6\%$.

Graph	Error ~10% Speedup w.r.t.		Error ~20% Speedup w.r.t.		Error ~50% Speedup w.r.t.	
	Exact Parallel	ABRA	Exact Parallel	ABRA	Exact Parallel	ABRA
fb-Friendships	4.01×	4.49×	8.68×	8.13×	19.94×	8.98×
soc-Pokec	3.94×	4.52×	8.55×	8.18×	19.65×	9.05×
loc-Gowalla	3.54×	4.36×	7.68×	7.90×	17.65×	8.74×
roadnetSF	3.88×	4.95×	8.40×	8.96×	19.28×	9.91×
usroad48	3.83×	4.86×	8.31×	8.79×	19.08×	9.72×
rmat17	3.79×	4.38×	8.21×	7.94×	18.68×	8.77×
random17	2.38×	2.01×	5.17×	3.64×	11.89×	4.32×
geomean	**3.58**×	**4.08**×	**7.76**×	**7.38**×	**17.82**×	**8.16**×

Table 6.5: Performance of ParTBC w.r.t. exact parallel Brandes' algorithm and ABRA Error $\sim 10\%$, 20%, 50%.

report the time taken for graph reordering, which is observed to be negligible (less than 1%) compared to the total execution time of the BC computation. In order to measure the effect of vertex renumbering on global memory coalescing, we measure the global memory load efficiency (*gld_efficiency*) collected using NVIDIA's visual profiler. Table 6.3 presents the *gld_efficiency* (expressed as a percentage) in parallel Brandes' algorithm on the graph before and after vertex renumbering. Higher the *gld_efficiency*, better is the memory coalescing. We observe that the *gld_efficiency* increases as a result of vertex renumbering. This reaffirms our time-investment in vertex renumbering.

We observe that the performance of the exact parallel version on the modified graph layout (resulting from vertex-renumbering) is consistently better than that on the original layout, for all types of graphs, resulting in an average speedup of $1.15\times$. The improvement is primarily due to better global memory coalescing. We also observe that power-law graphs (such as FB), get benefited more due to coalescing. This happens due to a high number of active nodes in an iteration (due to small diameter), in the level-synchronous BFS traversal in the forward-pass. This leads to more coalesced memory accesses once the graph is made more structured. For road networks (e.g., RNSF), the gains are limited as only a few nodes are active in each iteration, and the number of

accesses to nearby nodes (after renumbering) is limited. Due to a lack of structure, random graphs are not sensitive to vertex renumbering.

6.5.1 Overall Results

We evaluate the inaccuracy of ParTBC techniques as follows: Let bc_G be the exact set of top-k vertices, and $\widetilde{bc}_G$ be the set of top-k reported by ParTBC. Then the error incurred for each of the techniques is measured as $|bc_G - \widetilde{bc}_G|$.

Comparison with Baseline-I and Baseline-II.

Table 6.4 compares the performance of *DynRR* from ParTBC with the two baselines at 6% inaccuracy. We report the speedups averaged across $k \in \{100, 500, 1000, 2000, 3000, 5000\}$ for error $\sim 6\%$ for both ABRA and ParTBC.

We observe that for 6% inaccuracy the geomean speedup of ParTBC w.r.t. Baseline-I is 2.5$\times$ while that w.r.t. ABRA is 3.88$\times$. We also present a breakup of the contribution of each of the two phases of Algorithm 9 in the speedup achieved w.r.t. Baseline-I. Table 6.5 presents the performance of *DynRR* from ParTBC w.r.t. the two baselines at 10%, 20% and 50% inaccuracy. The results show that for the size and type of graphs in our test suite, the ParTBC outperforms ABRA consistently without failing the accuracy constraint. This is because ABRA uses an iterative progressive-sampling based approximation algorithm. Its execution time is contingent on the sample size, the number of samples, and the number of iterations, all of which grow with the size of the graph for a specified value of accuracy. Additionally, the sample size is updated in each iteration. In contrast, ParTBC scales well with the size of the graph. There is a very gradual increase in the overhead (which primarily includes time for graph reordering in Phase-I, time for online selection of sources, and time for computing the termination condition) with an increase in the graph size. In addition, ParTBC reduces the amount of work done by the same factor irrespective of the size of the graph, for obtaining the specified accuracy. Hence, even for moderate-sized graphs, as in our test-suite, ABRA takes longer than ParTBC across graphs and different values of k. All techniques in ParTBC consistently outperform ABRA on moderate- and large-sized graphs. ABRA performs better than ParTBC with better accuracy on small graphs such as Enron-email

($|V|$ = 36,682 $|E|$ = 183,831).

As observed, ParTBC outperforms the Baseline-I in exchange of lower accuracy. This is expected due to the lower overall work done by the techniques in ParTBC.

With ParTBC, we obtain appreciable speedups for error $\geq 6\%$. For achieving error below 6% using the proposed techniques, we need to select a very large fraction ($> 80\%$) of the vertices as sources, thus tending towards the exact version of Brandes' algorithm. As a result, the performance gains for error below 6% are diminished. In our experiments we do not show results for error below 6% because we do not observe any noticeable performance improvement. Using ParTBC would be helpful in applications where an error of $\sim 6\%$ or higher is acceptable in top-k betweenness centrality vertex computation.

Evaluation of Source Selection Techniques in ParTBC. Next, we evaluate the effect of the approximate techniques for the selection of sources (Section 6.4). We do so by using the parallel implementation of Brandes' algorithm (from Baseline-I) executed on the modified graph layout, post vertex-renumbering as the new baseline. We call this Baseline-III. In Table 6.6, we report the comparison of the ParTBC techniques with Baseline-III. This is the geomean speedup and error of the different techniques in ParTBC computed across $k \in \{100, 500, 1000, 2000, 3000, 5000\}$ for all graphs in our testbed. α denotes the fraction of vertices chosen as sources.

We observe that the techniques achieve high speedups for smaller α. This is due to fewer outerloop iterations. On the other hand, as one would expect, the approximation error is also high. As α increases, both the speedup and the error reduce. Out of the six variants, *Random* and *DynRR* achieve the least error. The effect-difference across techniques, however, reduces with increasing α; for instance, beyond $\alpha = 0.8$, all the techniques achieve similar accuracy.

We observe that for $\alpha \geq 0.5$, the overall mean error is less than 6% for techniques *Random, RRR, Dyn, DynRR*, with decent speedups. Thus, these techniques bear the potential to estimate the set of top-k vertices faster with a small accuracy loss for a variety of graphs.

To study the robustness of the proposed techniques, we study the effect of varying

	α	**Mean speedup**	**Mean error**		α	**Mean speedup**	**Mean error**
Random	0.1	7.6×	9.7%	**Dyn**	0.1	8.3×	16.4%
	0.2	3.3×	7.3%		0.2	3.9×	11.9%
	0.3	2.6×	5.4%		0.3	2.7×	8.6%
	0.4	1.8×	4.9%		0.4	2.2×	7.1%
	0.5	1.3×	3.7%		0.5	1.8×	5.5%
	0.6	1.2×	3.0%		0.6	1.2×	4.2%
	0.7	1.1×	2.4%		0.7	1.1×	3.0%
	0.8	1.1×	2.0%		0.8	1.1×	2.0%
Ascending	0.1	7.4×	27.7%	**Descending**	0.1	7.3×	17.3%
	0.2	3.2×	20.1%		0.2	3.2×	13.0%
	0.3	2.4×	14.6%		0.3	2.4×	9.6%
	0.4	1.6×	12.0%		0.4	1.7×	8.7%
	0.5	1.3×	9.3%		0.5	1.4×	6.6%
	0.6	1.1×	4.9%		0.6	1.2×	5.0%
	0.7	1.1×	3.6%		0.7	1.1×	4.7%
	0.8	1.1×	2.2%		0.8	1.1×	3.5%
RRR	0.1	8.1×	11.9%	**DynRR**	0.1	8.7×	15.8%
	0.2	3.6×	10.0%		0.2	3.8×	10.5%
	0.3	2.4×	7.9%		0.3	2.8×	8.1%
	0.4	2.1×	7.1%		0.4	2.1×	6.9%
	0.5	1.6×	5.8%		0.5	1.7×	5.3%
	0.6	1.2×	4.2%		0.6	1.3×	4.1%
	0.7	1.1×	3.0%		0.7	1.2×	2.8%
	0.8	1.1×	2.5%		0.8	1.1×	1.6%

Table 6.6: Overall results

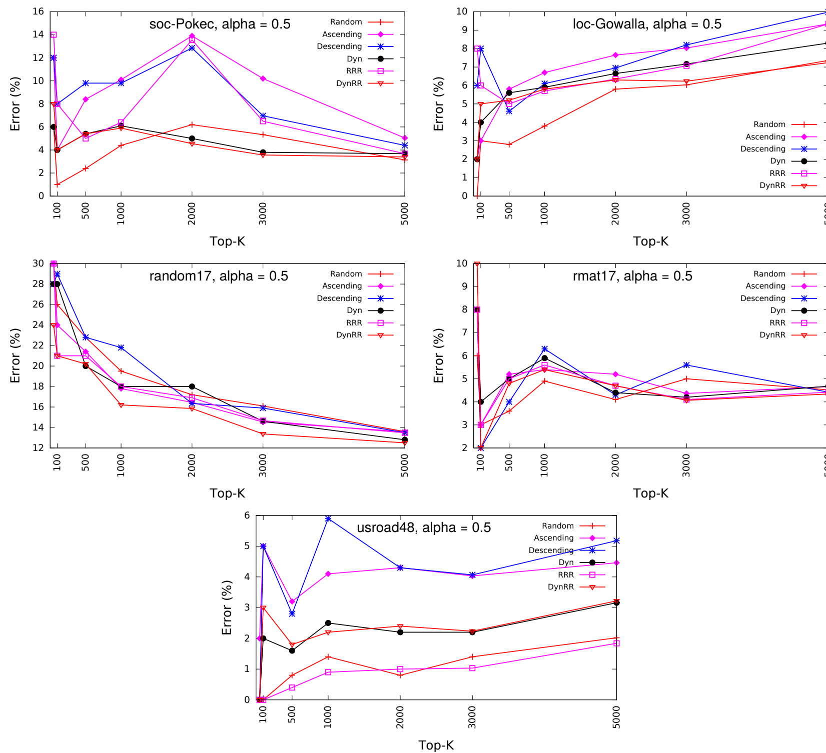


Figure 6.5: Graph-wise effect of varying k on the error for $\alpha = 0.5$

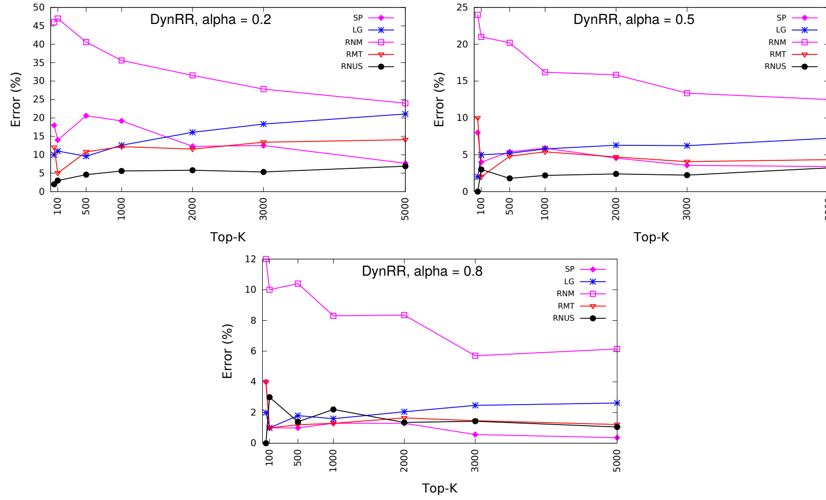


Figure 6.6: Effect of varying k on the error for DynRR

k on error. Figure 6.5 shows the variation in error with k on different graphs for various proposed techniques for $\alpha = 0.5$. As a general trend across all graphs, the absolute error in top-k increases with k.

For $\alpha = 0.5$, the error is least for the road-network graph (RNUS) – below 6% across all techniques, while the error is highest for the random graph (RNM). In the case of RNUS graph, *RRR* maintains the least error (below 2%) for all k. *Random*, *Dyn*, and *DynRR* follow closely with error less than 3%. The increase in error with k is slow for these four techniques. *RRR* leads to the least error for road-networks because a road-network has low, uniform vertex degrees. Thus selecting the nodes in a round-robin fashion selects favorable source nodes. The *Random* technique performs well on road-networks since nodes have similar characteristics.

We note that for power-law graphs (e.g. SP, RMT), the error is small (below 6%) for the *Random*, *Dyn* and *DynRR* techniques. The error for these three techniques decreases rapidly with an increase in k, and the errors are also very similar for large k. Uptil $k = 2000$, *Random* has the least error (below 4%), but for higher k, the *Dyn* and *DynRR* have the least errors (below 4%). In general, *DynRR* performs well for larger k for all graphs because the initial set of seed nodes selected using the round-robin technique provides a fair share of the BC values to the various important nodes.

Among all graphs, random graph (e.g. RNM) exhibits high error. It is because of a lack of well-defined structure. It is noteworthy that for *DynRR* (Figure 6.6), the

error does not exceed 7% for any of the graphs for $\alpha = 0.5$ for reasonably large k in our setup. The deviation in error is also small. Overall, *DynRR* emerges as a very stable technique that works consistently well across all but random graphs, for all k. We observed similar patterns in variation of k for $\alpha = 0.8$.

6.5.2 Effects of the fraction of source nodes

Figure 6.7 shows the variation in error with α for various proposed techniques for $k = 500$. For each graph, for each technique, the error decreases with increase in α since a higher value of α translates to performing more work and moving closer to the exact version ($\alpha = 1$).

For $k = 500$, we observe that for road-network graphs (e.g. RNUS), beyond $\alpha = 0.3$ the change in error is small and very gradual for *Dyn* and *DynRR*. The error is also low. The slight slope of the curve suggests little change in the relative ordering of the BC scores of the nodes on choosing more source nodes. It also hints at the scalability of these techniques for road-networks. We may terminate Brandes' algorithm after 30% iterations, without incurring high error, and gain immensely in execution time.

The error tends to decrease slowly in case of SP and RMT graphs with *DynRR*, *RRR*, *Dyn* and *Random* for $\alpha \geq 0.5$. For RNM graph, it is seen that there is a steep decrease in error values with increase in α. This shows that the relative ordering of nodes continues to change even for high α. Thus, for random graphs, exact BC needs to be computed for top-k. Overall, we observe that the technique *DynRR* has quite a good accuracy for most of the graphs (Figure 6.8) for $\alpha \geq 0.5$. At $\alpha = 0.5$, the error is less than 5% for most of the graphs, and as α increases, the percentage error decreases, which indicates the robustness and the scalability of the technique. We observed similar trends for higher values of k.

While the results suggest that the techniques *DynRR* and *Random* are comparable, *DynRR* trumps *Random* in practice on two accounts: 1) *DynRR* has lower overhead in computing the stopping condition compared to *Random*. 2) *DynRR* is deterministic, while in case of *Random*, an execution may result in higher inaccuracy than usual due to the selection of *unfavorable* source nodes.

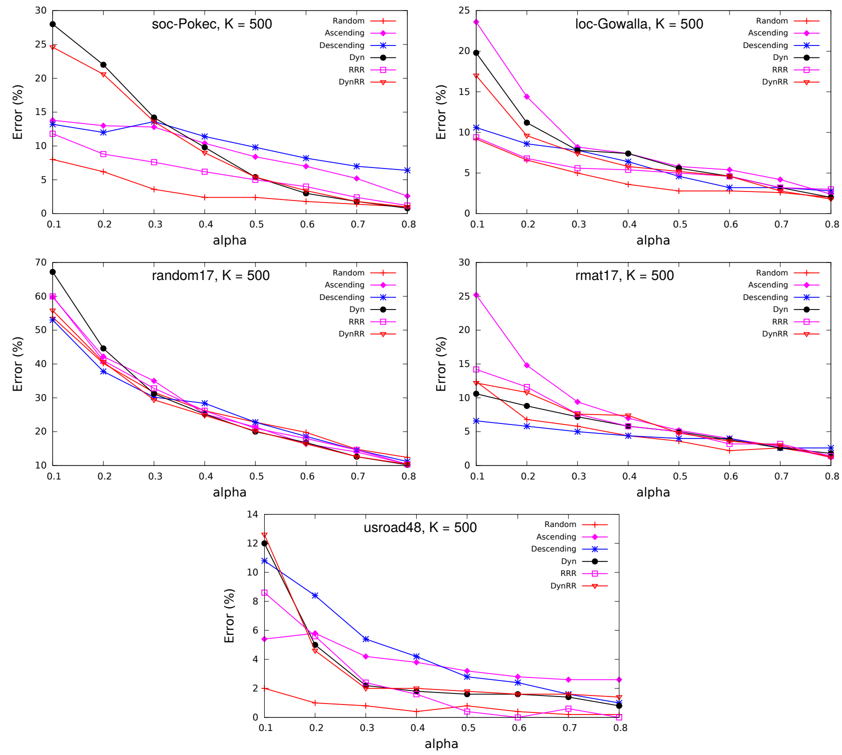


Figure 6.7: Graph-wise effect of varying α on the error for k = 500

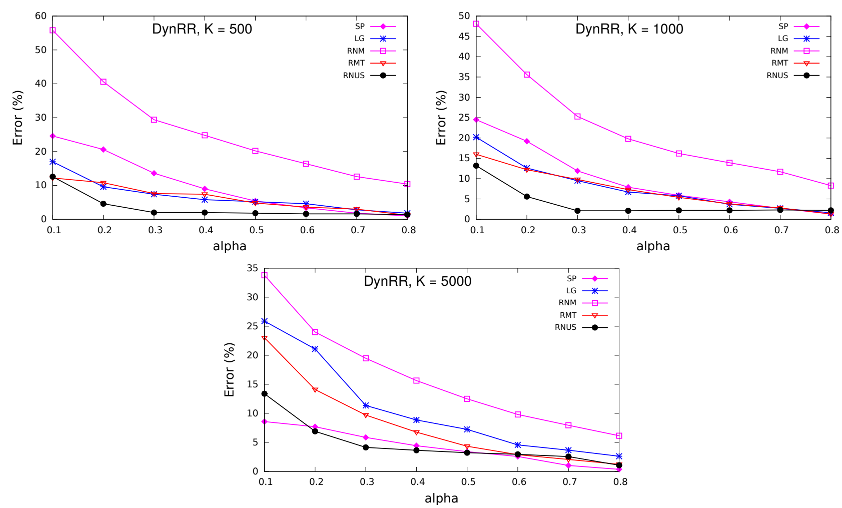


Figure 6.8: Effect of varying α on the error for DynRR

6.5.3 Controlling the number of outerloop iterations

For all the techniques in ParTBC, the number of outerloop iterations can be tuned by setting an appropriate stopping criterion. For every technique, the speedup and error are tied to the number of outerloop iterations. As discussed in Section 6.4, the stopping criterion is a function of C_t and t. In general, t dominates C_t in governing the number of iterations as the top-t vertices typically stabilize in P ($\gg C_t$) iterations.

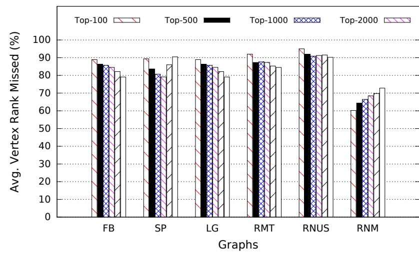
The trends in Figure 6.7 can be used as a guideline for setting C_t and t for different types of graphs. In general, to get good speedups and accuracy across all the proposed techniques, C_t and t should both be set to a small value for road-networks. For power-law graphs, C_t and t should be higher.

In our experiments, for *DynRR*, we assign $C_t = 5$ and $t = 5$ for power-law graphs, and $C_t = 4$ and $t = 3$ for road-networks. This resulted in the execution of 55% outer-loop iterations for LG graph and 28% iterations for the RNUS graph using *DynRR*. On the other hand, for *Random*, we assign $C_t = 10$ and $t = 50$ for power-law graphs, and $C_t = 5$ and $t = 40$ for road-networks. This resulted in the execution of 60% outerloop iterations for LG graph and 35% iterations for the RNUS graph using *Random*.

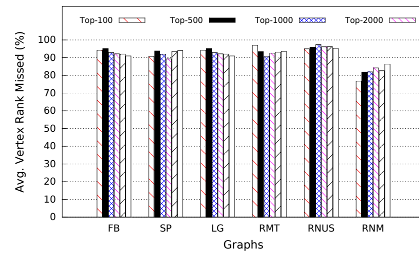
6.5.4 Discussion on quality of the reported top-k

To assess the quality of the top-k nodes reported by the proposed techniques, we consider two measures: (i) the mean exact rank of the nodes in top-k that are not reported using the approximate techniques. (ii) the mean exact rank of the nodes that are not in the exact top-k but are erroneously included.

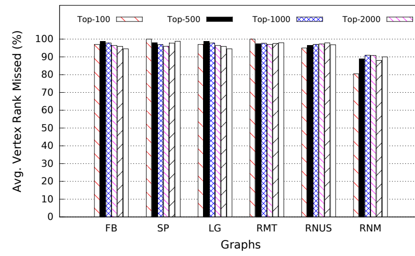
We argue, on empirical evidence, that the nodes we report as top-k are indeed *important*. We observe that the top-k nodes we fail to include in the set of top-k vertices are the ones that have their true ranks close to k, based on their exact BC scores. So, our techniques miss out on less important nodes among the exact top-k. Figure 6.9 shows the average true rank of the nodes we missed for *DynRR*. The average rank is expressed as a fraction of k. The nodes having lower ranks are more important. In the plot, values close to 100% depict that the nodes we missed have an actual rank closer to



(a) $\alpha = 0.2$

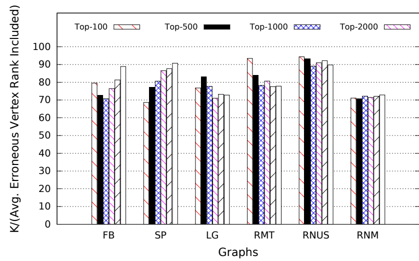


(b) $\alpha = 0.5$

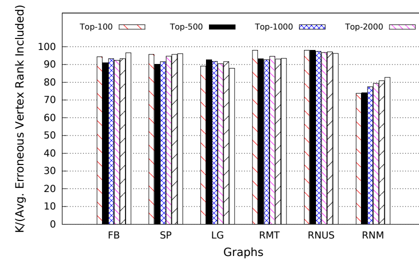


(c) $\alpha = 0.8$

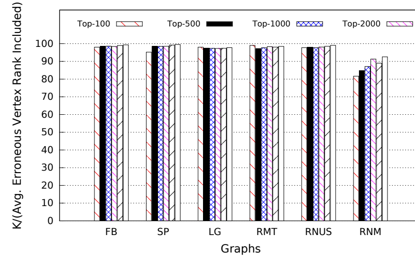
Figure 6.9: Average rank of top-k vertices missed (expressed as a percentage of k)



(a) $\alpha = 0.2$



(b) $\alpha = 0.5$



(c) $\alpha = 0.8$

Figure 6.10: Average rank of vertices erroneously included in top-k (expressed as k / Avg. rank)

k. We observe that for low α (Figure 6.9a), we miss more number of important nodes for all types of graphs. This is due to a high error in the reported top-k. For $\alpha = 0.5$ (Figure 6.9b), the values lie between 90% and 100% for all graphs except for random graphs, for which the error in top-k is high. Following a similar trend, the accuracy approaches 100% for $\alpha = 0.8$ (Figure 6.9c) for all the graphs. Among the graphs, the road network is the most stable for all k for all α, while the random graph is the most unstable. In general, we observe that average rank of the nodes missed (as a percentage of k) is proportional to the accuracy of the technique. This shows that we miss the nodes having true ranks in the range (accuracy% of k, k).

We further observe that the nodes we erroneously include in the top-k vertices are also among the important nodes although they are not in the exact top-k vertices. In order to verify this, we observed the ratio $R = \frac{k}{\text{true avg. rank of erroneous nodes}}$. Figure 6.10 shows this ratio, expressed as a percentage, for *DynRR*. Higher this value, closer is the average rank of the erroneous nodes to k. From the plot, we can compute the range of the true ranks of the erroneously included nodes within $((1 - 1/R) \times k)$ away from k on an average. For $\alpha = 0.5$ (Figure 6.10b), the values are in the range 90% – 100% for all the graphs except the random graph. The true rank of the nodes erroneously included in the top-k are in the range [$k+1$, $\frac{1}{0.9} \times k$), that is, [$k+1$, $1.1 \times k$). For higher α, the range shrinks. For $\alpha = 0.8$ (Figure 6.10c), the range reduces to [$k+1$, $1.02 \times k$) on an average for all graphs except the random graph. Since the true rank of the erroneous nodes is close to k, the nodes we include in the top-k are important.

6.5.5 ParTBC techniques are platform-independent

We believe that the ParTBC techniques are platform-independent. The source selection techniques in ParTBC exploit the fact that in Brandes' algorithm, the subset of the vertices picked as sources, and their order is instrumental in distributing a sizeable fraction of their respective BC scores in the early iterations of the algorithm, facilitating quicker estimation of the top-k vertices. These do not depend on or exploit any GPU-specific details such the GPU architecture. Hence, while we have evaluated the techniques only on GPU, we believe that ParTBC would work on CPUs too.

6.6 Summary

In this chapter, we presented a systematic study of heuristics for selecting source vertices in Brandes' algorithm that enable us to establish the relative ordering among the nodes quicker. Further, we proposed a novel graph-reordering scheme to make the graph layout more regular to allow efficient coalesced access of data to/from global memory in parallel Brandes' algorithm on GPU. The renumbering scheme is beneficial to other parallel graph algorithms on GPUs that employ vertex-centric push-based implementation. We demonstrated empirically that our proposed techniques compute the top-k BC vertices with a speedup of $2.5\times$ compared to the exact parallel Brandes' algorithm in exchange for a mean error of less than 6% on graphs of varying characteristics. Our techniques are robust and work well for larger values of k.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we proposed several i) algorithm- and architecture-independent; ii) algorithm-independent but architecture-specific, and iii) algorithm-specific but architecture-independent techniques for enhancing the efficiency of parallel graph processing using approximate computing. As evidenced by the experimental evaluation, our proposals are effective in improving the performance of popular graph algorithms for various classes of graphs encountered in practice, at the expense of accuracy.

With Graprox, we studied the effect of various algorithmic approximations on graph algorithms on GPUs. It is believed that for irregular computations such as graph algorithms, the effectiveness of a technique depends primarily upon the input. For instance, there exist algorithms that target specially power-law graphs and which do not work well with large diameter graphs. On the contrary, our study showed that while the amounts of performance improvement and inaccuracy vary, approximations are consistently helpful in achieving the trade-off well. Our techniques are general and applicable to other graph algorithms as well.

We next proposed Graffix— a set of graph transformation techniques for making graph processing on GPUs performant using approximate computing. Our techniques are targeted at improving memory coalescing, memory latency, and thread divergence. In order to improve memory coalescing, we reorder the graph vertices and replicate a select set of vertices to make the graph more *structured*. We reduce memory latency by processing the *well-connected* sub-graphs, iteratively, inside shared memory. We alleviate thread divergence, resulting from skewed degree distribution, by normalizing degrees across nodes assigned to a warp. We illustrated that our proposed techniques reduce execution times of parallel implementations of graph algorithms appreciably in exchange for small inaccuracies in the final output.

ParTBC proposed techniques for faster estimation of top-k betweenness centrality vertices in a graph. Our techniques are aimed at picking a subset of the vertices as sources in a specific order such that all graph vertices receive a sizeable fraction of their respective BC scores in the early iterations of Brandes' algorithm, facilitating quicker estimation of the top-k vertices. We demonstrated, empirically, the effectiveness of our techniques on graphs of varying characteristics and sizes.

A key feature of all the techniques presented in the thesis is that they provide *tunable knobs* to change the degree of approximation injected and control the performance-accuracy trade-off in graph applications. Our approximate computing techniques complement (and do not compete with) the existing optimization techniques and could be applied on top of these optimizations to enhance the execution performance further.

Overall, we showed that approximate computation of graph algorithms is a robust way of dealing with irregularities. Approximate computing combined with parallelization promises to make heavy-weight graph computation practical, as well as scalable. We envision that with rapidly growing data sizes, our techniques would be a building block for (i) versatile devices performing a range of precision-efficiency trade-offs, and (ii) software models that predict the right amount of approximation to be added to computation, to achieve a required service.

7.2 Limitations

The results presented in this thesis are empirical. For all the works – Graprox, Graffix and ParTBC, we do not prove theoretical bounds on the quality of the results, i.e., the magnitude of the error incurred upon applying our approximate methods.

We do not provide a cost model that outputs the appropriate values of the tunable parameters for achieving the desired performance-accuracy tradeoff. While we provide guidelines to assist in getting to the desired values of the tunable parameters faster, determining the correct values would require running extensive experimentation for a new algorithm-graph pair. The results we present in the thesis could be helpful in designing such a cost model.

7.3 Future Work

We discuss a few interesting directions that could be explored building on the ideas presented in the thesis.

Parallel approximate processing on dynamic graphs. The ideas presented in the thesis have been shown to be applicable and effective in improving the efficiency of graph analytics on static graphs. However, many real-world graphs, such as social networks and communication networks, are observed to be dynamic, evolving over time. A major challenge in the processing of dynamic graphs is to minimize re-computation, after each update and to apply updates incrementally. It would be interesting to explore techniques for incremental updates and approximate computing for further improving the scalability of dynamic graph processing on GPU. In cases where the changes are frequent, approximate computing may be used to consider only a subset of the changes to improve the performance while incurring small errors. Further, updates may be considered to arrive in batches or as a stream (with a limited buffer). This may be exploited to effect fewer re-computations by taking into account the cumulative effect or overlapping computations of the batched updates.

Framework for high performance approximate graph processing. As we have shown, combining parallelization with approximate graph processing promises significantly higher performance in graph analytics. However, applying and devising approximate techniques for getting the desired accuracy and efficiency is non-trivial and often requires considerable effort, both in terms of programming and in arguing about the error bounds on the solution. This is primarily because the performance and accuracy of approximate techniques for graph analytics are contingent on the nature of the graph algorithm, the input graph and the underlying hardware. There are several frameworks that target exact graph analytics on parallel platforms. More recently there have been efforts to design frameworks for specific graph approximations, such as lossy graph compression. However, there are no generalized frameworks that facilitate design and analysis of techniques for approximate graph processing. It would be useful to develop a generalized framework for parallel approximate graph analytics. A key feature of the framework would be a performance cost model that would enable the comparison of

different applicable approximate computing techniques using an accuracy metric. The accuracy metric may either be graph-specific, or algorithm-specific or both. The framework would include a set of graph-specific and algorithm-specific accuracy measures. The user may also specify a custom accuracy measure. For example, in case of algorithms that output a vector of vertex attributes (e.g. betweenness centrality, pagerank), the accuracy may be measured as a count of the number of vertices for which the deviation is more than the acceptable threshold. Further, given an approximation budget (in terms of acceptable accuracy and desired speedup) and the accuracy metric, the system would report the most suitable approximate computing technique. For example, a technique with the highest speedup to error ratio may be deemed suitable if it respects the specified budget. Designing mechanisms to evaluate the performance tradeoff of a specific approximate technique is another challenge. Another important feature of the framework would be to facilitate applying multiple approximate techniques, if feasible. This would enable the user to develop techniques as a composition of existing ones. Further, different techniques may perform well for different metrics and it may be unclear how they may perform together. The efficacy of the resulting technique may be studied using the performance model of the framework. Introducing these functionalities into existing high performance graph frameworks for exact graph processing would require making major changes to the design and the existing code base.

Locality Sensitive Hashing based approximate parallel graph processing.

Locality Sensitive Hashing (LSH) (Indyk and Motwani (1998); Gionis *et al.* (1999)) allows us to hash objects into values such that objects that are similar will be hashed to the same value with high probability. The notion of similarity needs to be defined appropriately for different applications. A formal definition of LSH is the following:

A family of hash functions, $\mathcal{H}$ is called (r_1, r_2, P_1, P_2)–locality sensitive if for any two items p and q

- if dist(p, q) $\leq r_1$, then $\mathbb{P}_{\mathcal{H}}[h(p) = h(q)] \geq P_1$
- if dist(p, q) $\geq r_2$, then $\mathbb{P}_{\mathcal{H}}[h(p) = h(q)] \leq P_2$; where $r_1 < r_2$, $P_1 > P_2$.

LSH-based methods provide asymptotic theoretical properties which could be leveraged for approximate parallel graph processing. A possible formulation could be the

following:

For an algorithm (and for a class of graphs), define a notion of *distance*. This distance measure will let us decide the extent of similarity between the entities of interest. More the distance, lesser the similarity between objects. Further, we would want this distance to be a *metric distance* to make use of the theoretical guarantees provided by LSH. The formulation of a locality sensitive hash function, for the decided notion of distance, will be such that similar entities will be hashed in to the same bucket and dissimilar entities hash in to different buckets.

The entities/objects of interest would be either edges or vertices, which will be governed by the quantity being computed by the algorithm. For instance, for finding strongly connected components, the LSH formulation would hash the nodes belonging to the same strongly connected component in one bucket and so on. In the case of top-k BC-vertex computation, the nodes whose BC scores are nearby would be hashed in to the same bucket.

The approximations in such techniques would lie in the formulation and quality of the locality sensitive hash function. We can then parallelize these LSH techniques for parallel execution on GPUs. Since these techniques would be far more generic and would apply to a large subset of graph algorithms and input graphs, their parallelization would enable faster graph processing with theoretical bounds on the quality of the result.

REFERENCES

1. **Ahn, K. J.**, **S. Guha**, and **A. McGregor**, Analyzing graph structure via linear measurements. *In Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12. Society for Industrial and Applied Mathematics, USA, 2012.

2. **Ashari, A.**, **N. Sedaghati**, **J. Eisenlohr**, **S. Parthasarathy**, and **P. Sadayappan**, Fast sparse matrix-vector multiplication on gpus for graph applications. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14. IEEE Press, Piscataway, NJ, USA, 2014. ISBN 978-1-4799-5500-8. URL `https://doi.org/10.1109/SC.2014.69`.

3. **Bader, D. A.**, **S. Kintali**, **K. Madduri**, and **M. Mihail**, Approximating betweenness centrality. *In Proceedings of the 5th International Conference on Algorithms and Models for the Web-graph*, WAW'07. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-77003-8, 978-3-540-77003-9. URL `http://dl.acm.org/citation.cfm?id=1777879.1777889`.

4. **Bader, D. A.** and **K. Madduri**, Parallel algorithms for evaluating centrality indices in real-world networks. ICPP '06. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2636-5. URL `http://dx.doi.org/10.1109/ICPP.2006.57`.

5. **Balaji, V.** and **B. Lucia**, Combining data duplication and graph reordering to accelerate parallel graph processing. *In Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450366700. URL `https://doi.org/10.1145/3307681.3326609`.

6. **Bandyopadhyay, B.**, **D. Fuhry**, **A. Chakrabarti**, and **S. Parthasarathy**, Topological graph sketching for incremental and scalable analytics. *In Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16. Association for Computing Machinery, New York, NY, USA, 2016. ISBN 9781450340731. URL `https://doi.org/10.1145/2983323.2983735`.

7. **Benczúr, A. A.** and **D. R. Karger**, Approximating s-t minimum cuts in Õ(n2) time. *In Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96. ACM, New York, NY, USA, 1996. ISBN 0-89791-785-5.

8. **Besta, M.**, **M. Podstawski**, **L. Groner**, **E. Solomonik**, and **T. Hoefler**, To push or to pull: On reducing communication and synchronization in graph computations. *In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17. Association for Computing Machinery, New York, NY, USA, 2017. ISBN 9781450346993. URL `https://doi.org/10.1145/3078597.3078616`.

9. **Besta, M.**, **S. Weber**, **L. Gianinazzi**, **R. Gerstenberger**, **A. Ivanov**, **Y. Oltchik**, and **T. Hoefler**, Slim Graph: Practical Lossy Graph Compression for Approximate Graph

Processing, Storage, and Analytics. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19. ACM, New York, NY, USA, 2019. ISBN 978-1-4503-6229-0. URL `http://doi.acm.org/10.1145/3295500.3356182`.

10. **Boldi, P.**, **M. Rosa**, **M. Santini**, and **S. Vigna**, Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. *In Proceedings of the 20th International Conference on World Wide Web*, WWW '11. Association for Computing Machinery, New York, NY, USA, 2011. ISBN 9781450306324. URL `https://doi.org/10.1145/1963405.1963488`.

11. **Borassi, M.** and **E. Natale** (2019). Kadabra is an adaptive algorithm for betweenness via random approximation. *J. Exp. Algorithmics*, **24**(1). ISSN 1084-6654. URL `https://doi.org/10.1145/3284359`.

12. **Brandes, U.** (2001). A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, **25**(2), 163–177.

13. **Burtscher, M.**, **R. Nasre**, and **K. Pingali**, A Quantitative Study of Irregular Programs on GPUs. *In Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '12. IEEE Computer Society, Washington, DC, USA, 2012. ISBN 978-1-4673-4531-6. URL `http://dx.doi.org/10.1109/IISWC.2012.6402918`.

14. **Carrillo, S.**, **J. Siegel**, and **X. Li**, A control-structure splitting optimization for gpgpu. *In Proceedings of the 6th ACM Conference on Computing Frontiers*, CF '09. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-413-3. URL `http://doi.acm.org/10.1145/1531743.1531766`.

15. **Chakrabarti, D.**, **Y. Zhan**, and **C. Faloutsos**, R-MAT: A recursive model for graph mining. *In Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*. SIAM, 2004. URL `https://doi.org/10.1137/1.9781611972740.43`.

16. **Chen, R.**, **J. Shi**, **Y. Chen**, **B. Zang**, **H. Guan**, and **H. Chen** (2019*a*). Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Trans. Parallel Comput.*, **5**(3), 13:1–13:39. ISSN 2329-4949. URL `http://doi.acm.org/10.1145/3298989`.

17. **Chen, R.**, **J. Shi**, **Y. Chen**, **B. Zang**, **H. Guan**, and **H. Chen** (2019*b*). Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Trans. Parallel Comput.*, **5**(3). ISSN 2329-4949. URL `https://doi.org/10.1145/3298989`.

18. **Coffman, T.**, **S. Greenblatt**, and **S. Marcus** (2004). Graph-based technologies for intelligence analysis. *Commun. ACM*, **47**(3), 45–47. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/971617.971643`.

19. **Dathathri, R.**, **G. Gill**, **L. Hoang**, **H.-V. Dang**, **A. Brooks**, **N. Dryden**, **M. Snir**, and **K. Pingali**, Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. *In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018. Association for

Computing Machinery, New York, NY, USA, 2018. ISBN 9781450356985. URL `https://doi.org/10.1145/3192366.3192404`.

20. **Devshatwar, S.**, **M. Amilkanthwar**, and **R. Nasre**, GPU Centric Extensions for Parallel Strongly Connected Components Computation. *In Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, GPGPU '16. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-4195-0.

21. **Dhulipala, L.**, **G. E. Blelloch**, and **J. Shun**, Theoretically efficient parallel graph algorithms can be fast and scalable. *In Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18. Association for Computing Machinery, New York, NY, USA, 2018. ISBN 9781450357999. URL `https://doi.org/10.1145/3210377.3210414`.

22. **Gharaibeh, A.**, **L. Beltrão Costa**, **E. Santos-Neto**, and **M. Ripeanu**, A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. *In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1182-3. URL `http://doi.acm.org/10.1145/2370816.2370866`.

23. **Gill, G.**, **R. Dathathri**, **L. Hoang**, **R. Peri**, and **K. Pingali** (2020). Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proc. VLDB Endow.*, **13**(8), 1304–1318. ISSN 2150-8097. URL `https://doi.org/10.14778/3389133.3389145`.

24. **Gionis, A.**, **P. Indyk**, and **R. Motwani**, Similarity search in high dimensions via hashing. *In Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-615-7. URL `http://dl.acm.org/citation.cfm?id=645925.671516`.

25. **Girvan, M.** and **M. Newman** (2002). Community structure in social and biological networks. *PNAS*, **99**(12), 7821 – 7826. ISSN 0027-8424.

26. **Goldberg, A. V.** and **C. Harrelson**, Computing the shortest path: A search meets graph theory. *In Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005. ISBN 0-89871-585-7.

27. **Gonzalez, J. E.**, **Y. Low**, **H. Gu**, **D. Bickson**, and **C. Guestrin**, Powergraph: Distributed graph-parallel computation on natural graphs. *In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12. USENIX Association, USA, 2012. ISBN 9781931971966.

28. **Grossman, S.**, **H. Litz**, and **C. Kozyrakis**, Making pull-based graph processing performant. *In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18. Association for Computing Machinery, New York, NY, USA, 2018. ISBN 9781450349826. URL `https://doi.org/10.1145/3178487.3178506`.

29. **Gubichev, A.**, **S. Bedathur**, **S. Seufert**, and **G. Weikum**, Fast and accurate estimation of shortest paths in large graphs. *In Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0099-5.

30. **Guha, S.** and **A. McGregor** (2012). Graph synopses, sketches, and streams: A survey. *Proc. VLDB Endow.*, **5**(12), 2030–2031. ISSN 2150-8097. URL https://doi.org/10.14778/2367502.2367570.

31. **Haghir Chehreghani, M.**, An efficient algorithm for approximate betweenness centrality computation. CIKM '13. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2263-8. URL http://doi.acm.org/10.1145/2505515.2507826.

32. **Han, W.**, **D. Mawhirter**, **B. Wu**, and **M. Buland**, Graphie: Large-scale asynchronous graph traversals on just a gpu. *In 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2017.

33. **Harshvardhan**, **A. Fidel**, **N. M. Amato**, and **L. Rauchwerger**, Kla: A new algorithmic paradigm for parallel graph computations. *In Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14. Association for Computing Machinery, New York, NY, USA, 2014. ISBN 9781450328098. URL https://doi.org/10.1145/2628071.2628091.

34. **Harshvardhan**, **B. West**, **A. Fidel**, **N. M. Amato**, and **L. Rauchwerger**, A hybrid approach to processing big data graphs on memory-restricted systems. *In 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2015. ISSN 1530-2075.

35. **Hoang, L.**, **M. Pontecorvi**, **R. Dathathri**, **G. Gill**, **B. You**, **K. Pingali**, and **V. Ramachandran**, A round-efficient distributed betweenness centrality algorithm. *In Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450362252. URL https://doi.org/10.1145/3293883.3295729.

36. **Hong, C.**, **A. Sukumaran-Rajam**, **J. Kim**, and **P. Sadayappan**, Multigraph: Efficient graph processing on gpus. *In 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2017.

37. **Hong, S.**, **S. Depner**, **T. Manhardt**, **J. Van Der Lugt**, **M. Verstraaten**, and **H. Chafi**, Pgx.d: A fast distributed graph processing engine. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC âĂŹ15. Association for Computing Machinery, New York, NY, USA, 2015. ISBN 9781450337236. URL https://doi.org/10.1145/2807591.2807620.

38. **Hong, S.**, **S. K. Kim**, **T. Oguntebi**, and **K. Olukotun**, Accelerating CUDA graph algorithms at maximum warp. *In PPoPP'11*. ACM, 2011. ISBN 978-1-4503-0119-0.

39. **Indyk, P.** and **R. Motwani**, Approximate nearest neighbors: Towards removing the curse of dimensionality. *In Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98. ACM, New York, NY, USA, 1998. ISBN 0-89791-962-9. URL http://doi.acm.org/10.1145/276698.276876.

40. **Jin, S.**, **Z. Huang**, **Y. Chen**, **D. Chavarría-Miranda**, **J. Feo**, and **P. C. Wong**, A novel application of parallel betweenness centrality to power grid contingency analysis. *In 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010.

41. **Jun, S.-W.**, **A. Wright**, **S. Zhang**, **S. Xu**, and **Arvind** (2017). Bigsparse: High-performance external graph analytics.

42. **Karantasis, K. I.**, **A. Lenharth**, **D. Nguyen**, **M. J. Garzarán**, and **K. Pingali**, Parallelization of reordering algorithms for bandwidth and wavefront reduction. *In SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014. ISSN 2167-4329.

43. **Khorasani, F.**, **K. Vora**, **R. Gupta**, and **L. N. Bhuyan**, Cusha: Vertex-centric graph processing on gpus. *In Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2749-7. URL `http://doi.acm.org/10.1145/2600212.2600227`.

44. **Kim, H.**, **J. Tang**, **R. Anderson**, and **C. Mascolo** (2012). Centrality prediction in dynamic human contact networks. *Computer Networks*, **56**(3), 983 – 996. ISSN 1389-1286. URL `http://www.sciencedirect.com/science/article/pii/S1389128611003975`. (1) Complex Dynamic Networks (2) P2P Network Measurement.

45. **Kim, K.-H.** and **Q.-H. Park** (2012). Overlapping computation and communication of three-dimensional FDTD on a GPU cluster. *Computer Physics Communications*, **183**(11), 2364 – 2369. ISSN 0010-4655. URL `http://www.sciencedirect.com/science/article/pii/S0010465512002044`.

46. **Koutis, I.** and **S. C. Xu** (2016). Simple parallel and distributed algorithms for spectral graph sparsification. *ACM Trans. Parallel Comput.*, **3**(2). ISSN 2329-4949. URL `https://doi.org/10.1145/2948062`.

47. **Kunegis, J.** (2017). KONECT: The koblenz network collection. `http://konect.uni-koblenz.de`.

48. **Kyrola, A.**, **G. Blelloch**, and **C. Guestrin**, Graphchi: Large-scale graph computation on just a pc. *In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12. USENIX Association, USA, 2012. ISBN 9781931971966.

49. **Lakhotia, K.**, **S. Singapura**, **R. Kannan**, and **V. Prasanna**, Recall: Reordered cache aware locality based graph processing. *In 2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. 2017. ISSN null.

50. **Lasalle, D.** and **G. Karypis**, Multi-threaded graph partitioning. *In Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13. IEEE Computer Society, USA, 2013. ISBN 9780769549712. URL `https://doi.org/10.1109/IPDPS.2013.50`.

51. **LeBeane, M.**, **S. Song**, **R. Panda**, **J. H. Ryoo**, and **L. K. John**, Data partitioning strategies for graph workloads on heterogeneous clusters. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'15. Association for Computing Machinery, New York, NY, USA, 2015. ISBN 9781450337236. URL `https://doi.org/10.1145/2807591.2807632`.

52. **Leskovec, J.** and **C. Faloutsos**, Sampling from large graphs. *In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06. Association for Computing Machinery, New York, NY, USA, 2006. ISBN 1595933395. URL `https://doi.org/10.1145/1150402.1150479`.

53. **Leskovec, J.** and **R. Sosič** (2014). SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`.

54. **Liljeros, F.**, **C. Edling**, **L. Amaral**, **H. Stanley**, and **Y. Aberg** (2001). The web of human sexual contacts. *Nature*, **411**, 907 – 908.

55. **Lim, Y.**, **U. Kang**, and **C. Faloutsos** (2014). Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, **26**(12), 3077–3089. ISSN 2326-3865.

56. **Liu, W.-H.** and **A. H. Sherman** (1976). Comparative analysis of the cuthill–mckee and the reverse cuthillâĂŞmckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, **13**(2), 198–213. URL `https://doi.org/10.1137/0713020`.

57. **Low, Y.**, **D. Bickson**, **J. Gonzalez**, **C. Guestrin**, **A. Kyrola**, and **J. M. Hellerstein** (2012). Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, **5**(8), 716–727. ISSN 2150-8097. URL `https://doi.org/10.14778/2212351.2212354`.

58. **Luo, L.**, **M. Wong**, and **W.-m. Hwu**, An effective GPU implementation of breadth-first search. *In Proceedings of the 47th Design Automation Conference*, DAC '10. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0002-5.

59. **Maass, S.**, **C. Min**, **S. Kashyap**, **W. Kang**, **M. Kumar**, and **T. Kim**, Mosaic: Processing a trillion-edge graph on a single machine. *In Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys âĂŹ17. Association for Computing Machinery, New York, NY, USA, 2017. ISBN 9781450349383. URL `https://doi.org/10.1145/3064176.3064191`.

60. **Madduri, K.** and **D. A. Bader** (2006). GTgraph: A suite of synthetic random graph generators. `http://www.cse.psu.edu/~madduri/software/GTgraph/`. [Online; accessed May 28, 2013].

61. **Madduri, K.**, **D. Ediger**, **K. Jiang**, **D. A. Bader**, and **D. Chavarria-Miranda**, A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. *In Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09. IEEE Computer Society, Washington, DC, USA, 2009. ISBN 978-1-4244-3751-1. URL `https://doi.org/10.1109/IPDPS.2009.5161100`.

62. **Malewicz, G.**, **M. H. Austern**, **A. J. Bik**, **J. C. Dehnert**, **I. Horn**, **N. Leiser**, and **G. Czajkowski**, Pregel: A system for large-scale graph processing. *In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10. Association for Computing Machinery, New York, NY, USA, 2010. ISBN 9781450300322. URL `https://doi.org/10.1145/1807167.1807184`.

63. **McLaughlin, A.** and **D. A. Bader**, Scalable and high performance betweenness centrality on the gpu. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14. IEEE Press, Piscataway, NJ, USA, 2014. ISBN 978-1-4799-5500-8. URL `https://doi.org/10.1109/SC.2014.52`.

64. **Merrill, D.**, **M. Garland**, and **A. Grimshaw**, Scalable GPU Graph Traversal. *In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1160-1.

65. **Mittal, S.** (2016). A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.*, **48**(4), 62:1–62:33. ISSN 0360-0300.

66. **Mumtaz, S.** and **X. Wang**, Identifying top-k influential nodes in networks. *In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17. ACM, New York, NY, USA, 2017. ISBN 978-1-4503-4918-5. URL `http://doi.acm.org/10.1145/3132847.3133126`.

67. **Nasre, R.**, **M. Burtscher**, and **K. Pingali**, Atomic-free irregular computations on GPUs. *In Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6. ACM, 2013*a*. ISBN 978-1-4503-2017-7.

68. **Nasre, R.**, **M. Burtscher**, and **K. Pingali**, Data-Driven Versus Topology-driven Irregular Computations on GPUs. *In Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13. IEEE Computer Society, Washington, DC, USA, 2013*b*. ISBN 978-0-7695-4971-2. URL `https://doi.org/10.1109/IPDPS.2013.28`.

69. **Nasre, R.**, **M. Burtscher**, and **K. Pingali**, Morph Algorithms on GPUs. *In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13. ACM, New York, NY, USA, 2013*c*. ISBN 978-1-4503-1922-5. URL `http://doi.acm.org/10.1145/2442516.2442531`.

70. **Nodehi Sabet, A. H.**, **J. Qiu**, and **Z. Zhao**, Tigr: Transforming irregular graphs for gpu-friendly graph processing. *In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18. ACM, New York, NY, USA, 2018. ISBN 978-1-4503-4911-6. URL `http://doi.acm.org/10.1145/3173162.3173180`.

71. **Pai, S.** and **K. Pingali**, A compiler for throughput optimization of graph algorithms on gpus. *In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-4444-9. URL `http://doi.acm.org/10.1145/2983990.2984015`.

72. **Pingali, K.**, **D. Nguyen**, **M. Kulkarni**, **M. Burtscher**, **M. A. Hassaan**, **R. Kaleem**, **T.-H. Lee**, **A. Lenharth**, **R. Manevich**, **M. Méndez-Lojo**, **D. Prountzos**, and **X. Sui**, The tao of parallelism in algorithms. *In PLDI'11*. ACM, 2011. ISBN 978-1-4503-0663-8.

73. **Prountzos, D.** and **K. Pingali**, Betweenness centrality: Algorithms and implementations. *In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-1922-5. URL `http://doi.acm.org/10.1145/2442516.2442521`.

74. **Riondato, M.** and **E. Upfal** (2018). Abra: Approximating betweenness centrality in static and dynamic graphs with rademacher averages. *ACM Trans. Knowl. Discov. Data*, **12**(5). ISSN 1556-4681. URL `https://doi.org/10.1145/3208351`.

75. **Roy, A.**, **I. Mihailovic**, and **W. Zwaenepoel**, X-stream: Edge-centric graph processing using streaming partitions. *In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13. Association for Computing Machinery, New York, NY, USA, 2013. ISBN 9781450323888. URL `https://doi.org/10.1145/2517349.2522740`.

76. **Sengupta, D.**, **S. L. Song**, **K. Agarwal**, and **K. Schwan**, GraphReduce: Processing Large-Scale Graphs on Accelerator-Based Systems. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15. Association for Computing Machinery, New York, NY, USA, 2015. ISBN 9781450337236. URL `https://doi.org/10.1145/2807591.2807655`.

77. **Seo, H.**, **J. Kim**, and **M.-S. Kim**, Gstream: A graph streaming processing method for large-scale graphs on gpus. *In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015. ACM, New York, NY, USA, 2015. ISBN 978-1-4503-3205-7.

78. **Sha, M.**, **Y. Li**, and **K.-L. Tan**, Gpu-based graph traversal on compressed graphs. *In Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19. ACM, New York, NY, USA, 2019. ISBN 978-1-4503-5643-5. URL `http://doi.acm.org/10.1145/3299869.3319871`.

79. **Shang, Z.** and **J. X. Yu** (2014). Auto-approximation of graph computing. *Proc. VLDB Endow.*, **7**(14), 1833–1844. ISSN 2150-8097. URL `https://doi.org/10.14778/2733085.2733090`.

80. **Shin, K.**, **A. Ghoting**, **M. Kim**, and **H. Raghavan**, SWeG: Lossless and Lossy Summarization of Web-Scale Graphs. *In The World Wide Web Conference*, WWW '19. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450366748. URL `https://doi.org/10.1145/3308558.3313402`.

81. **Shun, J.** and **G. E. Blelloch**, Ligra: A Lightweight Graph Processing Framework for Shared Memory. *In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13. ACM, 2013. ISBN 978-1-4503-1922-5.

82. **Shun, J.**, **L. Dhulipala**, and **G. E. Blelloch**, Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. *In 2015 Data Compression Conference, DCC*

2015, Snowbird, UT, USA, April 7-9, 2015. IEEE, Washington, DC, USA, 2015. URL http://dx.doi.org/10.1109/DCC.2015.8.

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

84. **Solomonik, E.**, **M. Besta**, **F. Vella**, and **T. Hoefler**, Scaling betweenness centrality using communication-efficient sparse matrix multiplication. SC '17. ACM, New York, NY, USA, 2017. ISBN 978-1-4503-5114-0. URL http://doi.acm.org/10.1145/3126908.3126971.

85. **Spielman, D. A.** and **S.-H. Teng**, Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. *In Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04. ACM, New York, NY, USA, 2004. ISBN 1-58113-852-0.

86. **Spielman, D. A.** and **S.-H. Teng** (2011). Spectral sparsification of graphs. *SIAM J. Comput.*, **40**(4), 981–1025. ISSN 0097–5397. URL https://doi.org/10.1137/08074489X.

87. **Turk, A.** and **D. Turkoglu**, Revisiting wedge sampling for triangle counting. *In The World Wide Web Conference*, WWW '19. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450366748. URL https://doi.org/10.1145/3308558.3313534.

88. **van der Grinten, A.** and **H. Meyerhenke** (2019). Scaling betweenness approximation to billions of edges by mpi-based adaptive sampling. *ArXiv*, **abs/1910.11039**.

89. **Wang, T.**, **Y. Chen**, **Z. Zhang**, **T. Xu**, **L. Jin**, **P. Hui**, **B. Deng**, and **X. Li**, Understanding graph sampling algorithms for social network analysis. *In 2011 31st International Conference on Distributed Computing Systems Workshops*. 2011.

90. **Wang, Y.**, **Y. Pan**, **A. Davidson**, **Y. Wu**, **C. Yang**, **L. Wang**, **M. Osama**, **C. Yuan**, **W. Liu**, **A. T. Riffel**, and **J. D. Owens** (2017). Gunrock: Gpu graph analytics. *ACM Trans. Parallel Comput.*, **4**(1), 3:1–3:49. ISSN 2329-4949. URL http://doi.acm.org/10.1145/3108140.

91. **Wei, H.**, **J. X. Yu**, **C. Lu**, and **X. Lin**, Speedup graph processing by graph ordering. *In Proceedings of the 2016 International Conference on Management of Data*, SIGMOD âĂŹ16. Association for Computing Machinery, New York, NY, USA, 2016. ISBN 9781450335317. URL https://doi.org/10.1145/2882903.2915220.

92. **Wu, B.**, **Z. Zhao**, **E. Z. Zhang**, **Y. Jiang**, and **X. Shen**, Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. *In PPoPP '13*. ACM, 2013. ISBN 978-1-4503-1922-5.

93. **Yazdanbakhsh, A.**, **D. Mahajan**, **H. Esmaeilzadeh**, and **P. Lotfi-Kamran** (2017). Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design Test*, **34**(2), 60–68.

94. **Zhang, E. Z.**, **Y. Jiang**, **Z. Guo**, and **X. Shen**, Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data remapping. *In Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0018-6. URL http://doi.acm.org/10.1145/1810085.1810104.

95. **Zhang, E. Z.**, **Y. Jiang**, **Z. Guo**, **K. Tian**, and **X. Shen**, On-the-fly elimination of dynamic irregularities for GPU computing. *In Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0266-1.

96. **Zhong, J.** and **B. He** (2014). Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, **25**(6), 1543–1552. URL http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.111.

97. **Zhu, X.**, **W. Chen**, **W. Zheng**, and **X. Ma**, Gemini: A computation-centric distributed graph processing system. *In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16. USENIX Association, USA, 2016. ISBN 9781931971331.

LIST OF PAPERS BASED ON THESIS

1. Somesh Singh and Rupesh Nasre. Scalable and Performant Graph Processing on GPUs using Approximate Computing. In *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)*, Volume 4, Number 2, 190 – 203, (2018). DOI: `https://doi.org/10.1109/TMSCS.2018.2795543`.

2. Somesh Singh and Rupesh Nasre. Optimizing Graph Processing on GPUs using Approximate Computing: Poster. In *Proceedings of the 24th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP 2019)*, 395 – 396. DOI: `https://doi.org/10.1145/3293883.3295736`.

3. Somesh Singh and Rupesh Nasre. Graffix: Efficient Graph Processing with a Tinge of GPU-Specific Approximations. In Proceedings of the *49th International Conference on Parallel Processing (ICPP 2020)*, 23:1 – 23:11.
DOI: `https://doi.org/10.1145/3404397.3404406`.