Engineering Parallel Graph Algorithms

Kishore Kothapalli

IIIT
Hyderabad

Contents

Preface

Graph algorithms continue to attract research attention for a vareity of reasons. In the parallel setting, this attention has led to the development of a variety of algorithms in the Parallel Random Access Machine (PRAM) model, huge efforts in engineering parallel algorithms for graphs,fff libraries of graph centric algorithms, and the like. The challenges tackled in this body of work has several dimensions including algorithmic issues, choice of data structures, load balancing, synchronization and runtime issues, library

In recent years, there has been tremendous interest in understanding the difficulty of computing on large graphs in parallel. These difficulties are accentuated by the inability of the memory systems of modern parallel architectures to handle the irregular read/write patterns typical of graph computations. Such a scenario presented researchers with an opportunity to apply principles from algorithm engnieering.

The discipline of algorithm engineering studies how an algorithm interacts with its environment including the system architecture and identifies heuristics

that help improve the performance of algorithms in practice. Such heuristics can sometimes be used to also offer a theoretical explaination to the behavior of the algorithm. When applied to graph algorithms, this allows for the possibility that existing PRAM based algorithms can be revisited for identifying suitable heuristics and algorithmic optimizations on modern parallel architectures such as multi-core CPUs and GPUs.

Indeed this has been in research focus over the last decade with various success stories. Notable among them include the GPU based algorithms for shoretst paths and spanning trees by Harish and Narayanan [32, 80], graph exploration on CPUs and GPUs [84, 109], strong connectivity of directed graphs [18].

A specific line of work that emerged in recent years targets specific domains of graphs and sees how the knowledge of the input can help identify further optimizations and heuristics to parallel algorithms.

This book attempts to capture this massive body of work from one specific view point that is motivated by algorithm engineering for graphs in the parallel setting.

How to Use this Book

This book is aimed at supporting a upper level project-based graduate elective or a seminar style elective on parallel graph algorithms. The material in Chapters 1 and 2 is of broad based nature that quickly aims to introduce the relevant ideas. Chapter 2 provides a primer on modern parallel architectures. Readers uninitiated in this detail will require supplementary reading material that is widely available on the web.

The rest of the chapters can be taken up collectively or as individual case studies. There are also no dependencies between Chapters 3 to 7 and efforts have been made to make these chapters as self-contained as possible. For this reason, the seminar group or the class can tailor dsicussion around individual chapters or a collection of chapters.

Acknowledgements

I am grateful to the research environment at IIIT Hyderbad that promotes excellence and stimulates passion. The small teaching load helped in setting aside several hours on endeavors of this kind. The leadership provided by P. J. Narayanan motivates people to set goals

and plan on ways to meet those goals. The advice by Kamalakar Karlapalem at various crucial points in this journey is often timely and exemplary. One cannot also forget the dsicussions with colleagues Suresh and Srinathan on a vareity of matters.

Many of the student co-authors on the papers have been quite resourceful in designing, planning, and executing the creation of large CUDA software sources for the various algorithms used in the papers and this book.

Some material in Chapters 3 and 4 of the book is based on the work presented at the International Conference on High Performance Computing (HiPC), 2016 where the paper received the best paper award as well. I would like to use this opportunity to thank the reviewers of the paper as well as the program committee for bestowing this honor on our paper.

Our group at IIIT Hyderabad is also fortunate to garner research support of various kind from a variety of governmental, strategic, and industry bodies including the Department of Science and Technology (DST), Governement of India, the Defense Research and Development Organization (DRDO), NVIDIA, Intel, and IBM, and Applied Materials. Interactions with representatives of the above bodies has always helped me keep abreast of the various developments in the ever expanding field of parallel computing in the current era.

Several teachers at various levels have made a lasting impact on me by their manners, disciplines, challenges posed, and the knowledge imparted. Many interactions with such teachers are still etched in my memory and have been great learning experiences. To all those teachers, I always remain deeply indebted.

Chapter 1

Introduction

In this chapter, we introduce the field of algorithm engineering starting in the sequential computing setting and parallel algorithm engineering along with suitable examples. We then describe the notion of domain-driven algorithm engineering that is gaining attention in the parallel computing community. we use standard notation, see e.g., [46], in describing the asymptotic run time of an algorithm. All the graphs we mention are simple and undirected. For a graph $G = (V, E)$, we let the number of vertices (or nodes), $|V|$, be n and the number of edges, $|E|$, to be m. Other terminology with respect to graphs, such as paths, diameter, degree, cycle, subgraph, will be defined as needed in the rest of the book. For notation on graphs that we use without any explicit definition, we use standard notation on graphs [159].

We assume that the reader is familiar with some of the basic graph algorithms such as those for traversals, shortest paths, spanning trees, and the like. The book by Cormen et al. [46] is a good source of such material.

1.1 Algorithm Engineering

Algorithm design and analysis tries to analyze the behavior of the algorithm on worst-case inputs, best-case, and occasionally on average-case inputs. Moreover, these analyses are communicated via the asympototic behavior of the algorithm. Of these, the worst- and best-case analysis sometimes corresponds to an input, or a class of inputs for which the algorithm indeed meets its worst, and best-case behavior respectively. For average case analysis, the big question that algorithm analysis has to grapple with is what defines the average case. Is it that every input is equally likely?, or that a specific behavior of the input in the context of the algorithm is equally likley?. These troubling questions mean that the average behavior of an algorithms is less studied in general. Such gaps are also not likely to satisfy the inquisitiveness of practitioners as to the practicality of an algorithm with sound theoretical gaurantees, or the basis to choose one algorithm over another in practice.

A closely related discipline is algorithm engineering that deals with the study of algorithms and the computers that are used to execute the algorithms. One often tries to understand and evaluate the options available and make the algorithm benefit from the underlying architectural characteristics of target machines. In a departure from average case analysis, algorithm engineering can indicate reasons for the behavior of algorithms on particular classes of inputs based on the behavior of the input, compare algorithms for their differences in behavior across input classes, study the practicalities of various algorithms beyond asymptotics on input of varying scales, and the like. Often times, it is also possible to observe phenomenon in practice that can help one formulate conjectures and hypotheses that result in better algorithms. This is captured by Figure 1.1 that indicates the scope for algorithm engineering[1].

We now provide a concrete example of algorithm engineering via the work of Crescenzi et al. [49] for finding the diameter of a graph.

[1]Figure 1.1 is credited to G. Italiano from his slides.

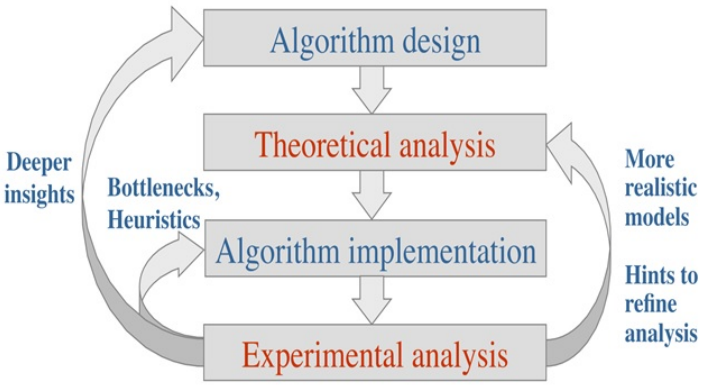


Figure 1.1: An illustration of the PRAM model.

Traditional algorithms that run in $O(mn)$ time are easy to design for this problem. One can reduce the time to $O(n^{\omega})$ where ω refers to the exponent of matrix multipliation. In an interesting result, Crescenzi et al. show that on most real-world graphs one can reduce the number of BFS traversals required by observing the impact of prior BFS traversals. In paritcular, denote a BFS tree rooted at node u of a graph G by T_u. Let the height of T_u be denoted as eccentricity of u, $\mathbf{ecc}(u)$. Define $F(u)$, the fringe set at u as the set of nodes at maximum distance from u. In other words, $F(u) := \{v | d(u,v) = \mathbf{ecc}(u)\}$. If we further define $B_i(u)$ as the maximum eccentriticy of nodes in $F(u)$, then it can be observed that for any nodes v, w at distance i and j from u, then $d(v,w) \leq B_i(u)$. Based on this observation, Crescenzi et al. show that for any x, i and k such that $x \in F_{i-k}(u)$ for $1 \leq i < \mathbf{ecc}(u)$ and $1 \leq k < i$ with $\mathbf{ecc}(x) > 2(i-1)$, there exists a node y_x in $F_j(u)$ such that $d(x, y_x) = \mathbf{ecc}(x)$ with $j \geq i$. This last observation in turn helps in specifying a good termination condition for the algorithm for finding the diameter of G as follows.

Let y be a node in $\cup_{j=i}^{\mathbf{ecc}(u)} F_j(u)$ with maximum eccentricity $\mathbf{ecc}(y) >$

$2(i-1)$. Then the eccentricity of all nodes in $\cup_{j=1}^{i-1} F_j(u)$ is not greater than $\mathbf{ecc}(y)$. This indicates that as one traverses the BFS tree T_u in a bottom-up fashion, at each level i, we can compute the eccentricities of all its nodes. if the maximum eccentricity e is greater than $2(i-1)$ then we can skip traversing the remaining levels as the eccentricities of all the nodes in such levels cannot exceed e. The running time of the algorithm based on these ideas is $O(nm)$ in the worst case, as in the worst case, we have to perform a bfs starting from "almost" every node of the graph. On most real-world graphs, the algorithm performs under 1% of the BFSs. For more details, we refer the reader to [49].

Two important observations are due about the algorithm of Crescenzi et al. Firstly, the run time of the algorithm is a function of the number of BFS traversals required which varies significantly based on the input instance and its properties. Secondly, however, the run time of the algortihm of Crescenzi et al. [49] meets the worst-case of $O(mn)$ for various classes of graphs as identified in [49, Section 3]. This situation best illustrates the scope for algorithm engineering as specific heuristic approaches can help run the algorithms fast but such behavior may not be analyzed in the traditional realm of asymptotic analysis of algorithms. Some may posit that algorithm engineering should provide algorithms that not only execute faster in practice but also base the faster run times via theoretical guarantees. In this book, we take a lenient view that some of the speed of the algorithms may not be applicable to all instances, and may not be explained by sound theoretical guarantees. This view agrees to the view espoused via Figure 1.1. We feel therefore that algorithm engineering as a field lies at the boundary of theoretical and practical Computer Science.

1.2 Parallel Algorithms and Parallel Algorithm Engineering

The recent decades have opened up exciting possibilities for algorithm engineering with the emergence of parallel computing. One of the

reasons for this is the widespread availability of parallel architectures such as the multi-core computers, accelerators such as the GPUs, and other emerging models such as the Intel Xeon fused CPU-FPGA architecture, and the like. In addition, parallel computing is now seen as an essential tool to handle scale issues arising out of ever-increasing size of data sets that correspond to real-world phenomenon. Parallel computing is now practiced in various domains starting with fundamental problems such as sorting to applied areas such as deep learning and bio-informatics.

This situation has meant that studying the performance of several parallel algorithms on modern parallel architectures has gained pace. In the context of parallel algorithms, design and analysis of parallel algorithms flourished in the earlier decades predominantly in the Parallel Random Access Machine (PRAM) model. The PRAM model, an illustration of which is shown in Figure 1.2 is a natural extension to the standard RAM model of computation. In the RAM model, we consider a computer connected to a memory unit via a bidirectional bus. The PRAM model, see also Figure 1.2, extends this notion by having several processors share a common pool of read/write memory via individual bidirectional buses. The PRAM model also assumes that all the processors are synchronous in nature, and the latency to read/write any memory cell by any processors is identical. Further fine-grained categorization of the PRAM model is introduced by the support for concurrent reads and writes. In the simplest case, the model prohibits any concurrent reads and writes, resulting the EREW PRAM model. The CREW PRAM model supports concurrent reads but forbids concurrent writes. The CRCW PRAM models allow concurrent writes also subject to additional semantics such as COMMON, ARBITRARY, PRIORITY and the like. For a more detailed background on this model, we refer the reader to the excellent treatise on parallel algorithms by JaJa [91].

The PRAM model provides wide leverage to algorithm designers. However, the model is not close to practical realizations in many aspects. To address this issue, several bridging models such as the Bulk Synchronous Parallel (BSP) model of Valiant [151], the LoGP

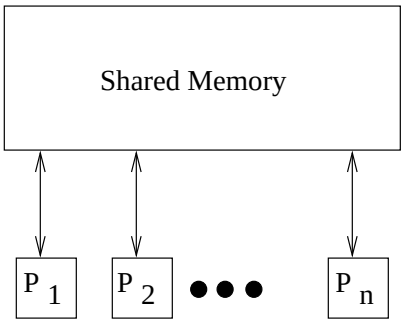


Figure 1.2: An illustration of the PRAM model.

model [52], the QRQW model [71] are also studied. The bridging models bring in aspects of network latency, thread level work imbalance, and penalty for concurrent operations into the algorithm design and analysis.

PRAM algorithms have been successful at opening up the inherent as well as latent parallelism in a variety of computations. For instance, Wyllie [164] highlights a new technique to design an efficient parallel algorithm for the seemingly sequential problem of prefix sum/scan. However, as the PRAM model hides a lot of details such as communcation delays, latency of memory reads/writes, syncrhonization overheads across the processors, it is often not clear as to the practical efficacy of the algorithms designed in the PRAM model. It was quickly observed that assumptions concerning the PRAM model are not realized in most parallel architectures. For instance, when one considers the list ranking algorithm of Wyllie [164] via pointer jumping on a computer that can support only T threads running simultaneously, one needs to be watch our for inconsistencies due to thread scheduling decisions. These can be mitigated by using expensive constructs such as locking or atomic operations which do not find a place in PRAM algorithms per se. Such additional steps and other efficient ways to counter these difficulties are part of several studies [129, 135].

To summarize, PRAM algorithms offer a good way to expose the

parallelism in a problem. However, as the number of concurrent and simultaneously executing threads do not scale beyond a fixed number, implementing PRAM algorithms on modern parallel architectures requires incorporating additional techniques that assure correctness, efficiency in practice, and the like. These issues are addressed in several ways over a large number of prior works for a variety of problems including sorting [105] and matrix multiplication [31].

Parallel algorithm engineering has been helpful in identifying several aspects of PRAM based parallel algorithms that have to be reinterpreted in how they are implemented so as to ensure correctness, account for multi-threaded nature of execution and the like. It is however realized that further gains are possible when one can also reinterpret steps of the PRAM algorithms depending on the practical cost of these steps. As an example, consider the optimal list ranking algorithm designed in the PRAM suggests using an optimal PRAM algorithm for finding a maximal independent set of a linked list. This requires $O(\log n)$ parallel time and $O(n)$ work. However, Banerjee and Kothapalli [15] argue that a similar effect can be achieved by using a Fractional Independent Set (FIS) of a linked list instead of an MIS given that an FIS can be obtained in $O(1)$ parallel time and $O(n)$ work. It is true that an FIS has a size that is much smaller than an MIS but the overall approach works since the smaller size of an FIS is offset by the quick computation to get an FIS.

1.3 Domain-Driven Algorithms

From the earlier sections, we understand that traditional algorithm design did not take structural properties of the input into account. Such properties turn out to be extremely interesting for graph algorithm design and implementation. A recent development to assess this opportunity is to understand the relation between the structural properties of the input and the impact on the computation. The basic premise of such a step is to view the design and development of a parallel algorithm as being impacted by three parameters: (i) the

input characteristics, (ii) The architecture of the computing platform, and (iii) the problem. For instance, if the problem is supposed to be solved on an accelerator, the choice of algorithm may differ compared to if the problem is supposed to be solved on a multi-core CPU.

So far, only (iii) and (ii) are the only aspects that were considered important. The input characteristics, item (i) above, were rarely considered. The only exceptions are the algorithms for sparse graphs designed by Johnson (cf. Johnson). There were instances in the literature where item (ii) above has been studied exclusively. Greiner (SPAA 1994) considers modifications to the parallel connected components algorithm of Shiloach and Vishkin (Shiolach and Vishkin,) when implemented on symmetric multi-processor systems. Similar approaches can be found in Soman et al. [145].

In this book, we make the case for the nature of input to be also used as an aid in designing and implementing algorithms. We call such algorithms as domain-driven algorithms where the best algorithm for a given problem depends also on the nature of input in addition to the usual parameters of the algorithm and the target architecture.

Domain-driven algorithms can add specific steps to the computation to address the nature of input they are designed to work for. For instance, the algorithm may preprocess the input suitably followed by the computation, and a possible post-processing. A domain-driven algorithm may not only alter the input, but also can call for a different computation on the altered input with the guarantee that the result of the new computation on the altered input is identical to the output of the actual computation on the original input. It has to be noted that such algorithms may exhibit a good performance only for inputs coming from specific classes for which it is designed for. For other inputs, at best, these algorithms may have the same runtime as the best possible algorithm for the problem. These algorithms may be more suitable in either the asymptotic sense or in the practical efficiency or both for the domain of inputs they are aimed at and possibly meet, or even exceed, the asymptotic worst case run time on inputs coming from other domains.

Domain driven algorithms typically benefit from the following

techniques.

- *Reduce*: An algorithm can *reduce* the amount of computation on inputs from specific domains. In this scenario, an existing algorithm that has a large worst-case asymptotic bound will now has a better bound on the run time owing to the reduced computation.

- *Reuse*: An algorithm may identify elements of the computation that can be deduced from the results of other computations performed by the algorithm. In this case, the results of such computation can be reused to obtain the results of computation that has not been performed explicitly. This technique is useful when computation is expensive but deducing the result of a computation as a function of the results of other computations is easy.

- *Reinterpret*: In this scenario, the domain-driven algortihm may replace certain steps that are typically expensive in practice with alternative less expensive steps. It is likely that the result of the less expensive alternatives do not exactly correspond to the output of the steps that they replace. In this case, one has to resort to a post-processing phase that aims to remove the errors induced by using the results of the non-exact computation. Such al algorithm will be faster in practice if the post-processing step also is efficient in practice.

There has been some recent progress in this direction in the field of parallel computing. One aspect that is making an impact in recent years is to take into account some of the structural properties of graphs arising from real-world phenomenon such as road networks, social networks, and web grpahs. Examples of some of these properties may include being sparse, having a particular distribution of the degrees of the vertices, having a large number of k-connected components for some small k such as 2 or 3, the way distances between vertices behave, having long maximal paths, and the like. One or

more of these properties may help in arriving at better algorithms by either reducing the size of the graph or the space of exploration thereby reducing the amount of computation, reusing computation on nodes of degree at least 3 for nodes of degree at most 2, and replacing breadth-first spanning trees with randomly chosen subgraphs. Throughout the book, we will provide more specific examples of such techniques.

1.4 Relation to Other Fields

We also would like to distinguish the field of algorithm engineering with other related disciplines. We consider fileds such as programming for performance, cacghe-oblivious algorithms, and resource-oblivious algorithms.

1.4.1 Programming for Performance

Programming for performance aims to implement algorithms on target architectures by also taking into account the behaviour of the architecture. To this end, one often considers a host of tehcniques at various levels of the program while most of the algorithm remains unchanged. Some of these techniques are:

- Program Space Optimizations: To achieve good performance one has to understand the number of threads to use, thread group size if applicable, thread work assignment in terms of affinity and the like. The space of these optimizations could be highly unstructured so much so that one may have to explore the entire space to find the best possible values for the parameters considered. Recently, a few techniques such as roofline optimization [41] that can explore space efficiently are presented.

- Compiler Optimizations: These techniques aim to address issues in code transformation such as loop unrolling, loop switching, dead code/instruction elimination, tail call elimiation, function inlining, and the like. (A list of such features the LLVM

Compiler supports is available at [106].) Additional optimizations such as register allocation, instruction scheduling, auto-vectorization are also included in this set but are very dependent on the target machine.

- Memory Hierarchy Optimizations: These techniques attempt to understand how to increase the memory system throughput by addressing issues related to the cache size and cache line size, memory read and write behavior, cache associativity, inter-thread read/write affinity and shared caches, and the like.

  A good example of illustrating memory hierarchy based optimizations is the tiled approach for matrix multiplication as demonstrated by the CUDA source code [1]. In this case, for computing the product of two square matrices A and B, the program fetches parts of the matrices into the shared memory, uses these parts to compute partial output, and then brings in the next parts of the matrices into the shared memory. The tile size is dictated by the amount of shared memory available on the device.

It can be noticed that many of the above optimizations are often highly specific to individual architectures and hence the optimizations cannot be ported to other comparable architectures directly. Further, there is also a significant dependence on the sequence these optimizations are applied and the resulting gains in performane [124]. For these reasons, programs that include the above optimizations are often handcrafted. It is therefore not surprising that such programs are often created as part of architecture supported libraries such as the Intel Math Kernel Library for matrix computations [2], the NVIDIA `cublas` library for BLAS routines [21, 118], and the like. This situation poses a great difficulty for also practitioners as the program has to change significantly across machine models.

1.4.2 Cache Oblivious and Resource-Oblivious Algorithms

In a related area, the work on cache-oblivious algorithms aims to design and implement algorithms that can work well for any memory hierarchy of a given target architecture. These algorithms are analyzed by parameterizing the sizes of the individual members of the memory hierarchy. For instance, M and B are used to denote the sizes of any two consecutive levels of the memory hierarchy with the assumption that $M \geq B^2$. (M is the size of the *slower* memory.) It is further assumed that the slower level always transfers B words of data together to the faster level. Under these mild assumptions, one is often interested in designing and analyzing algorithms for the asymptotic number of cache misses incurred by the algorithm in terms of the input size and B. One popular design technique in the algorithm design process is divide and conquer. When using divide and conquer, the algorithm recurses on the sub problems and at some point, the size of the subproblems reduces to fit into B inducing no further cache misses.

Algorithms for problems such as sorting, matrix transpose, and matrix multiplication, are desgined in this model by Frigo et al. [63]. While the model is useful for such algorithmic explorations, experimental studies [104] notes that cache-aware algorithms tend to outperform the cache-oblivious algorithms. The reasons for this phenomenon include factors beyond the scope of the model such as hyperthreading, TLB misses, cache associativity, and the like.

1.4.3 Massively Parallel Algorithms

Distributed computing is one of the traditionally closest fields to parallel algorithms. The big difference between the two is the provision of shared memory in parallel computing whereas in distributed computing processors that are typically assumed to be identical and communicate via messages to perform a computation. In most distributed computing models, the parameter of comparison is the number of rounds of communication required. More formally, in the distributed

computing model, we have n processors that are connected with communication links. Computation proceeds in rounds where in each round, each processor can send messages to each of its neighboring processors, perform local computation, and receive messages from its neighbors. The number of such rounds required to perform the computation is measured as the round complexity of the algorithm.

The LOCAL variant of the distributed computing model does not put any restriction on the size of the messages that can be sent/received in any one round. It assumes that therefore each link has unlimited bandwidth and hence the network congestion is completely ignored. The aspect of locality of information is what is of interest in this model. In this model, the main question to study is to see how many rounds of communication are required to solve a given problem by *collecting* all the required pieces of information. On the other hand, the CONGEST model restricts the bandwidth of each link to $O(\log n)$ bits where n is the size of the input.

More recent and practical variants of the distributed computing include the Map-Reduce model [54], the k-machine model [101], and the Massively Parallel Compting (MPC) model [97, 165], among others.

The Map-Reduce model requires all computations to be expressed via two functions: a mapper and a reducer. The framework allows for automatically distributing the computation across a set of machines. The *mapper* in the Map-Reduce model reads the input data and processes the data to create a set of <key,value> pairs. These <key,value> pairs are then fed to the *reducer* applies another user defined function on these pairs to produce the output. The framework includes routines to manage the runtime and also resilience to faults. The Map-Reduce framework is a good way to create a large cluster out of commodity machines.

The k-machine model can be seen as formalizing the Map-Reduce model where there are k machines that are connected as a clique with each link having a bandwidth of B bits. The input is assumed to be partitioned across the k machines usually in equal volume according to some model: uniform where each piece of input is mapped to any

of the k machines with uniform probability, or an adversarial model where the mapping of input to machines is assumed to be in the control of an adversary. The parameter of interest is the number of communication rounds required to solve a given problem.

The Massively Parallel Computing (MPC) model is similar to the k-machine model in some respects. The MPC model is defined by a set of machines with each machine limited to at most S words of memory. The machines are connected pairwise interconnected resembling an all-to-all communication network. Communication and computation in this model are synchronous. In each round, each machine can receive up to S words from other machines, performs local computation, and sends up to S words to other machines. A key element of this model is that both the memory upper bound S and the number of machines used are assumed to be strongly sublinear in the input size N. In other words, $S = O(N^{1-\epsilon})$, for some constant ϵ, $0 < \epsilon < 1$. This restriction allows the model to study modern large-scale computational problems where the input is simply too large to fit in a single machine and is much larger than the number of available machines.

For graph problems with n and m denoting the number of nodes and edges respectively, notice that $N = \tilde{O}(m + n)$ where m is the number of edges and n is the number of nodes of the input graph. The number of rounds needed to solve many graph problems of interest varies significantly based on how S relates to the number of nodes (n) of the input graph. Specifically, three regimes for S have been considered in the literature.

- **Strongly superlinear memory ($S = O(n^{1+\epsilon})$):** In this regime, it is assumed without loss of generality that the input graph is highly dense, i.e., $m \gg S \gg n$ such that S is strongly sublinear in m. (Otherwise a single machine can solve the problem using a space-efficient sequential algorithm.) Even though the input graph is dense, the fact that each machine has $O(n^{1+\epsilon})$ local memory makes this model quite powerful. For example, in this model, problems such as minimum spanning tree, MIS, and 2-approximate minimum vertex cover, all have $O(1)$-round algo-

rithms [81,97]. Input filtering is a main technique that is useful in this regime. In filtering, one iteratively sparsifies the input until the entire problem fits on one machine. Once such a size reduction is achieved, the problem is solved using a standard sequential algorithm. For example, for connected-components, one can just throw out edges locally, preserving the global connectivity, until the graph has size at most s.

- **Near-linear memory ($S = \tilde{O}(n)$):** Problems become harder in this regime, but symmetry breaking problems such as MIS, approximate minimum vertex cover, and maximal matching can still be solved in $O(\log \log n)$ rounds [9, 50, 67, 68]. Furthermore, recently Assadi, Chen, and Khanna [10] presented an $O(1)$-round algorithm for $(\Delta + 1)$-vertex coloring.

- **Strongly sublinear memory ($S = O(n^{\epsilon})$):** Given the restriction on the space, problems seem to be much harder to solve in this regime. There are also several open questions related to the existence of sublogarithmic-round algorithms for certain graph problems in this regime. For example, it is conjectured that the problem of distinguishing if the input graph is a single cycle vs two disjoint cycles of length $n/2$ requires $\Omega(\log n)$ rounds [69, 165]. However, even in this regime, Ghaffari and Uitto [70] have recently shown that MIS does have a sublogarithmic-round algorithm, running in $\tilde{O}(\sqrt{\log \Delta})$ rounds, where Δ is the maximum degree of the input graph.

1.5 This Book

There exists today a vast body of knowledge that fits in the theme of semantic parallel graph algorithms across architectures. Various resaerchers have used a range of techniques motivated by the properties of graphs such as having a large number of biconnected components, having a short diameter, having a scale-free nature of degree distributions, and the like. Some of the work done by the author in the last

five years is centered around the design and development of parallel semantic algorithms for a variety of problems on both graphs and matrices.

The book considers a wide spectrum of graph problems such as decompositions, connectivity, shortest paths, metrics, and enumerative problems. The intent of this book is to discuss the specific properties of graphs that enable algorithms for particular problems to work well in practice.

The rest of the book is organized in the following manner. In Chapter 2, we provide a brief overview of the data structures used for representing graphs in parallle programs and also summarize the features of the parallel archtiectures that are used in the experiments reported in the rest of the chapters.

In Chapter 3 we start by describing parallel algorithms for arriving at an ear decomposition and ONE MORE of a graph. The algorithms we present here outperform the current best-known parallel algorithms based on the PRAM model.

In Chapter 4, we the study of graph k-connectivity algorithms for $k = 1, 2$, and 3. In this case, we also show how even the result of Cheriyan and Thuirmella [38] that uses multiple breadth-first traversals is not very conducive in terms of practical performance. The algorithms presented in Chapter 5 try to use sampling and post-processing to arrive at the k-connected components of the graph.

In Chapter 6, we study how one can improve the practical performance of shortest path algorithms. The solution presented here uses ideas from Chapters 3 and 4 to speedup existing algorithms for finding the shortest paths of a weighted graph in parallel. Interestingly, these techniques can be applied to *any* parallel algorithm for shortest paths.

In Chapter 7, we focus on various graph metrics such as pagerank, diamter, and centrality. We propose techniques for computing the pagerank of the nodes of a directed graph that aim to identify nodes that are identical with respect to the computation, halt computation on nodes that coverge, and also propose an ordering of the computation that helps in faster convergence. The section on comput-

ing the diameter of an undirected graph is based on the techniques of Crescenzi et al. [49]. For finding the betweenness-centrality of nodes in a graph, we will once again use ideas from Chapters 3 and 4 and from the work of Sariyuce et al. [134].

Chapter 8 discusses work in the direction of enumerating small sized subgraphs of a given graph such as counting the number of triangles, or small induced subgraphs, and the like. This chapter is based on the works of Madduri et al.

The book ends with concluding remarks in Chapter 9.

Chapter 2

Preliminaries

2.1 A Short Primer on Modern Architectures

Parallel architectures have been in wide use for the last 15 years owing to multiple factors. The conventional wisdom in computer architectures that fuelled the growth of microprocessors could not be continued any further. In particular, (i) frequency could not be scaled up any more in what is termed as the "frequency wall", (ii) the power dissipated was growing at an alarming rate in what is termed as the "power wall", and (iii) the time it takes to transfer data from the memory to the CPU is increasing enormously in what is termed as the "memory wall'. Architects hence had to explore other possibilities such as multi-core CPUs to address the three challenges. These multi-core CPUs are typically single board designs with multiple cores on each board. The multiple cores are interconnected via particular topologies in some cases, and also often share one (or more) cache(s). We will discuss these in more detail in the subsequent sections.

Around the same time, the growth of accelerators led to their massive adoption in various applications and domains. NVIDIA GPUs stood out for reasons such as performance per Watt of power and low physical cost. Using GPUs for general-purpose computations picked pace along with sophistication in programming platforms, libraries,

tools, and techniques over the years.

In this chapter, we review concepts related to the high-level architectures of CPUs and GPUs that we use in experiments in the rest of the book. As many of these architectures are revised every year or sooner, we will highlight commonalities across these revisions. Such a view will help us also help us relate the experiments across machine models.

2.2 Multi-core Architectures

Multi-core architectures refer to CPUs that house multiple independent cores on a single chip. In this chapter, we will focus mostly on the Intel CPU architecture. From 2007 to 2016, Intel used a tick-tock model to enhance its multi-core architectures line since the 65 nm fabrication model. Each tick corresponds to a new microarchitecture absorbing advances in manufacturing process technology, reducing the fabrication width. The corresponding tock uses the new microarchitecture to support additional hardware features in terms of instructions, thereby improving application performance at the architecture level, energy efficiency, and the like. Typically, a tick and a tock span two years. Below, we summarize the features introduced at the various tock cycles.

The Intel architecture Yona used the 65 nm fabrication technology to house two cores with a shared L2 cache of 2 MB. The Intel Penryn architecture developed with 45 nm fabrication technology CHECK

The 4-core Nehalem architecture developed using the 45 nm fabrication technology introduced a simultaneous multi-threading technique, or hyper-threading such that each core can support two physical threads of execution. This architecture also introduced SIMD extensions to the instruction set that benefit programs that have SIMD nature of computations. The four cores have separate L_1 and L_2 caches of size 32 KB and 256 KB respectively and share an L_3 cache of size 8 MB.

Using the 32 nm process technology, the Sandybridge, with four

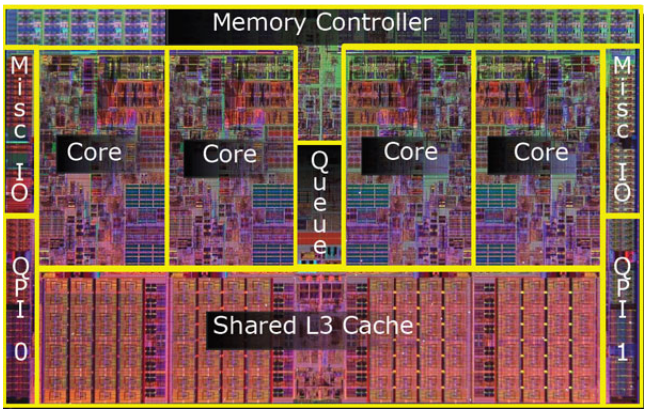


Figure 2.1: The block diagram of the Intel Nehalem microarchitectures.

cores, included an integrated graphics processor, support for vector operations, and a shared L3 cache of 8 MB. Besides, it introduced several instructions that affected the throughput of cryptographic operations that arise in algorithms such as AES [1] and RSA [131]. This architecture's cores are interconnected with a ring-based interconnect containing separate rings for data, request, acknowledgment, and snooping.

The 22nm based microarchitecture named Haswell, released in 2013, introduced further enhancements, including support for advanced vector instructions. These instructions, often called AVX2 Haswell instructions, expand vector operations to 256 bits apart from scatter/gather and support for shift operations. The microarchitecture also comes with an advanced power-saving mechanism by adding additional low-power states. Such mechanisms also made it possible to create form factors supporting thin and light ultrabooks.

Broadwell The Skylake microarchitecture uses the 14 nm manufacturing process technology introduced by the Broadwell architecture.

[1] See https://www.nist.gov/publications/advanced-encryption-standard-aes

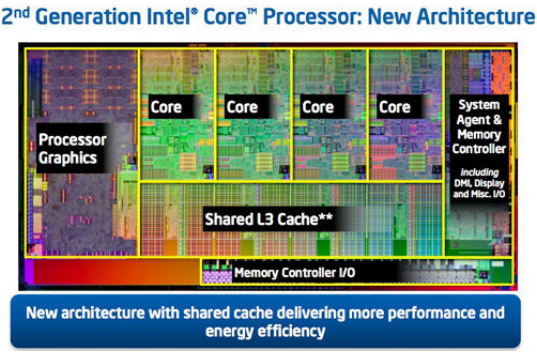


Figure 2.2: The block diagram of the Intel Sandybridge microarchitectures.

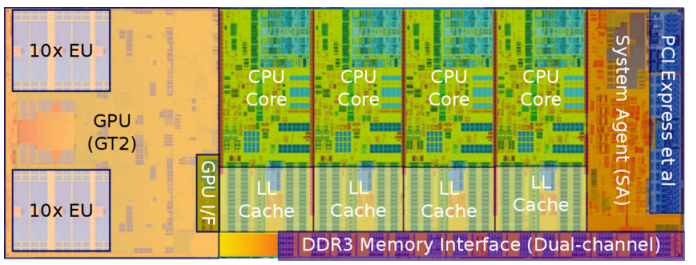


Figure 2.3: The block diagram of the Intel Haswell microarchitectures.

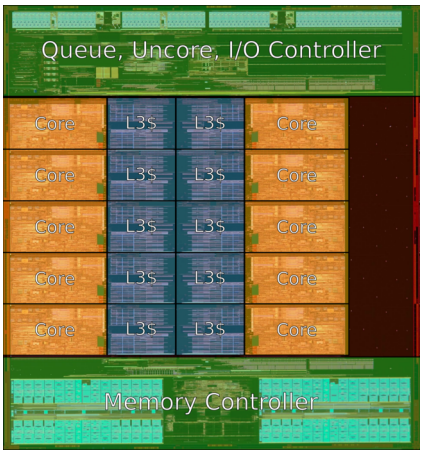


Figure 2.4: The block diagram of the Intel Broadwell microarchitectures.

While the default configuration is for four cores, the architecture extends up to 18 cores. This architecture extended support for AVX2 instructions to 512 bits compared to 256 bits for Haswell. Further refinements to Skylake include the Coffee Lake and Kaby Lake models.

From 2016 onward, the tick-tock model is replaced with a three step cycle called the process – architecture – optimization model. The first two steps of the three step cycle are manufactured in the same process technology regime while the third step introduces improvements in the process technology. Further progress along the 10 nm process technology is scheduled to release in the second half of 2019. The code name for this architecture is Cannon Lake.

We use the Intel(R) Xeon(R) E5-2650 CPU in our experiments in later chapters of this book. The Intel Xeon E5-2650 CPU has 128 GB RAM and a memory bandwidth of 68 GB/s. It is a dual-processor where each processor has 10 cores, and each core can process two threads using hyperthreading. Each core operates at 2.34 GHz, which can be boosted to 3 GHz using turbo boost technology. It has 64 KB L1 cache per core, 256 KB L2 cache per core, and a shared 25 MB L3

cache. It uses PCI Express 3.0 with lane combinations x4, x8, and x16 that connects processors, PCI lanes, and PCI devices. We use this CPU for our experiments on multi-core CPUs.

2.2.1 Programming the Intel Multicore

We use OpenMP 3.0 to program the CPU. The OpenMP 3.0 specification inlcudes three main components: a set of compiler directives, a set of runtime routines, and a set of environment variables. Compiler directives allow the programmer to indicate parts of code that can run in parallel, synchronization of work among multiple threads, distributing the loop iterations across threads, and the like. The runtime routines allow the programmer to query the (unique) identifier of a thread, lock handling, querying wall clock time, and the like. The environment variables allow the programmer to set the number of threads the program can use, thread wait policy, stack size and the like. An example OpenMP program using the C language is shown in Algorithm 2.2.1.

```
1: #include <omp.h>
2: main(int argc, char *argv[]) {
3: int numthreads, myid;
4: #pragma omp parallel private(myid) {
5: myid = omp_get_thread_num();
6: printf("Welcome to the book, Happy Reading, from thread %d\
   n", myid);
7: }
8: }
```

2.3 Accelerators

In this section, we will review existing works that have used a variety of accelerators, and move on to GPUs and GPU computing.

2.3.1 Accelerator Based Computing

With the availability of general-purpose programming languages, mature system software stack consisting of compilers, debuggers, and related utilities, the CPUs have occupied the core of the computing space. However, the nature of the CPU execution does not suit all computational patterns. Using special-purpose hardware helps better realize certain computational patterns that frequently arise in specific domains. The name given to such special-purpose hardware units designed for specific computation purposes is coprocessors. Coprocessors serve as an auxiliary processor that is separate from the CPU and extend the CPU functionality. Allowing for coprocessors helps specialized users enhance their hardware by spending extra money on custom hardware components. In some cases, with advances in technology, it is possible to integrate the coprocessor and the main processor. Often, a coprocessor is transparent to the user, but users could write instructions that execute directly on a coprocessor in some cases. These instructions can be in assembly language or some high-level language. A coprocessor can also sometimes provide a computing engine that can surpass that of the main processor on a range of computations because of significant architectural differences. The floating-point coprocessor is an excellent example of the former, and the graphics coprocessor is a stand-out example of the latter, as we shall describe in more detail in Subsection 2.3.1.

Popular hardware coprocessors include floating-point coprocessors, network coprocessors, graphics coprocessors, and more recently cryptographic coprocessors, and a coprocessor for regular expression matching.

Graphics Coprocessors

Producing images on a computer screen requires a significant number of graphics operations such as rendering, shading, coloring, culling, and the like. These requirements also grew as computer gaming started to become popular. Currently, the demand is in rendering complex game scenarios at resolutions exceeding that of 60 frames per

second. Fortunately, over the last four decades, graphics algorithms and their execution models went through significant improvements so that real-time rendering is possible at a minimal cost to the user.

In the early days, graphics operations such as drawing lines, wire-frame diagrams, and bitmaps were made possible by NEC via the 82720 graphics controller and is usable on Intel machines. Over the 1980's and the 90's, their functionality includes support for 2D and 3D graphics to hardware-accelerated graphics. Advances in manufacturing technologies helped increase the range of tasks that graphics coprocessors could handle. Software libraries such as OpenGL [162] were also developed around the same time and, in some cases, supported by graphics coprocessors.

The nature of graphics computations helped the emergence of graphics coprocessors immensely. Graphics operations usually are done in a pipelined manner, starting with pixels, composing them into triangles, applying surface shading, textures, colors, and the like. The hardware of early NVidia graphics processors

At present, three key players, Intel, AMD and NVidia dominate the GPU space. Intel also has been able to achieve a tighter integration of the GPU onto the CPU motherboard in its Westmere (CHECK) architecture. Doing so has advantages, including reducing the communication time between the CPU and the GPU. AMD also offers the APU (expand), which is also a fused CPU+GPU model. On the other hand, at present, NVidia is still designing increasingly more computationally powerful stand-alone GPUs. The latest GPUs from NVidia called the Kepler or GTX 680, can offer up to 2 TFLOPs of computing power at its peak.

2.3.2 General Purpose Computing on Graphics Processing Units (GPGPU)

In the previous section, we have provided a brief description of various Intel multi-core CPU micro-archtiectures. This section briefly describes the graphics coprocessors that gained significant research attention in the past decade or more. Notably, today's NVIDIA

graphics coprocessors offer a raw computational power that exceeds that of general-purpose CPUs by order of magnitude.

The vast difference is partly due to the fundamental differences in GPUs' design principles and CPUs. As GPUs aim at the video games industry, they require high throughput. Therefore, GPUs need to feed the data from memory to processors. Therefore GPUs support memory bandwidth that is far higher than CPUs and includes small caches that allow faster data access to multiple threads. The CPUs, on the other hand, are designed to minimize the latency of a single thread. Such support is possible due to large caches and is called a latency oriented design.

Typical NVIDIA GPU Architecture The NVIDIA GPUs consist of a set of Streaming Multiprocessors (SMs), each with multiple Scalar Processors (SPs). A core is another name for an SP. (The name SMXs replaced the earlier name of SMs). The cores within each SM have access to a large register set and shared memory. Each SM is also equipped with multiple functional units to perform arithmetic operations, control units, and an instruction cache. Each SMX has a hardware scheduler which schedules 32 threads at a time. This group is called a warp, and a half-warp is a group of 16 threads that execute in a SIMD fashion. Figure 2.5 shows a typical GPU architecture. The latest GPU offering from NVIDIA called the V100, has 5,376 cores arranged as 64 cores each in 84 SMXs. Table 2.1 shows the number of cores and SMXs in some of NVIDIA's recent GPUs.

The Tesla K40C GPU has 2880 compute cores arranged as 192 cores each in 15 SMXs. It has 12 GB onboard memory and 64 KB of on-chip memory per each SMX. All SMXs share an L2 cache of 1.5 MB.

Memory Hierarchy The NVIDIA family of GPUs comes with a deep memory hierarchy and offers various memory resources with varying read and write costs and characteristics. Figure 2.6 shows the various memory resources of a typical GPU. The entire GPU card has a global memory that every SM reads/writes. Typical sizes

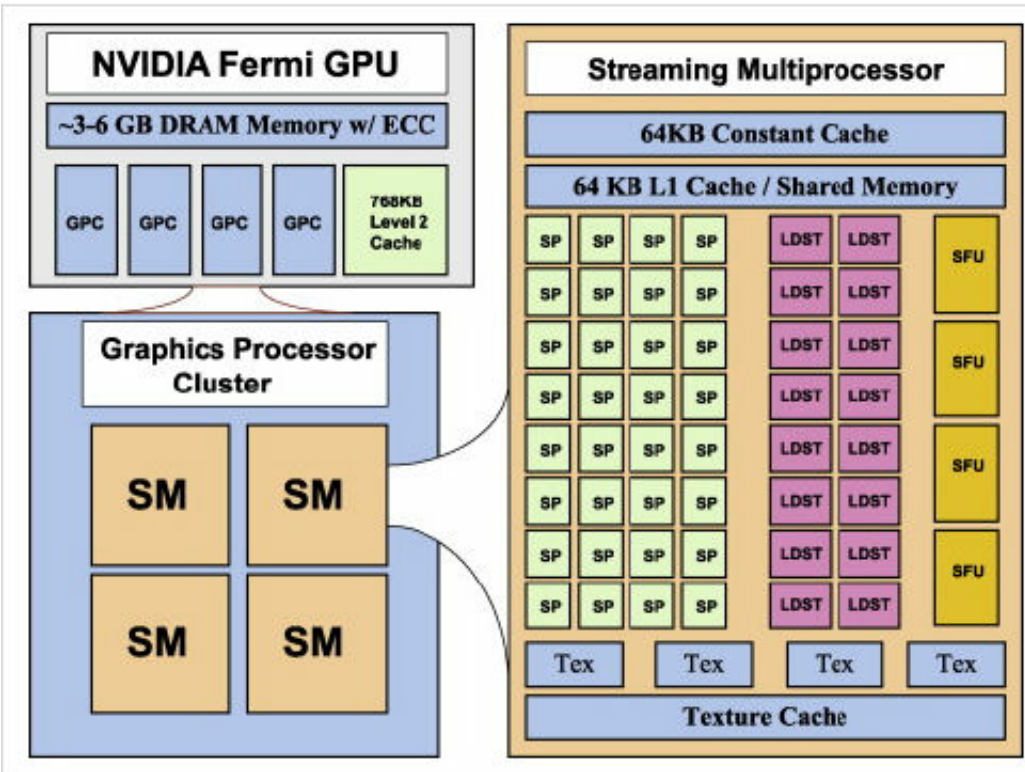


Figure 2.5: Memory hiearchy of a typical GPU.

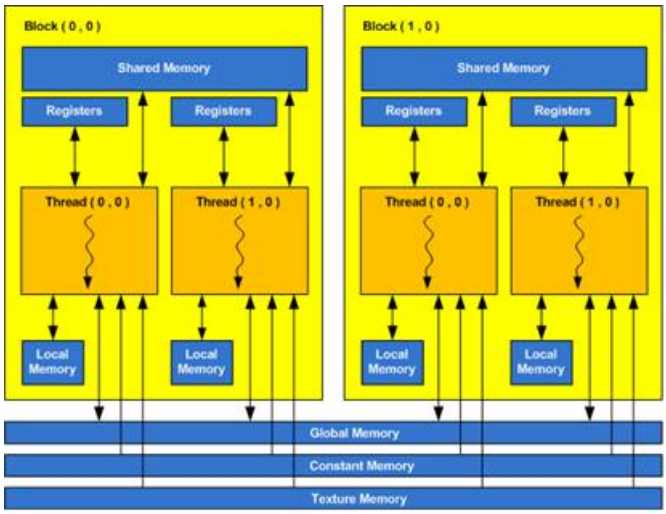


Figure 2.6: Memory hiearchy of a typical GPU.

of this global memory are of the order of GBs. However, the latency of this memory depends on the nature of access. Access to this global memory by a half-warp of threads to a contiguous 128 Byte block is termed as *coalesced* access. Under a coalesced access, the average latency suffered by any thread in a half-warp is small as one transaction from the memory controller suffices to satisfy the requests of the entire group of threads. On the other hand, if threads in a half-warp request arbitrary locations in the global memory, then the memory controller has to perform multiple transactions in serial order, thereby increasing the latency incurred by such thread groups. This situation refers to *uncoalesced* access.

Apart from the global memory, cores in each SMX have access to a large register file and a common shared memory. The register file has a large number of 32-bit registers with an access latency of a few cycles. This large register file allows multiple threads to store several variables in the registers for faster access. Across generations of GPUs, the number of registers in the register file have increased

from a model 32K registers in the Fermi GPU to a massive 256 K registers in the Tesla family of GPUs.

The shared memory is useful for storing data required in common by a half-warp of threads in execution. This shared memory is usually of a few MB orders and arranged in the form of logical banks. Access to locations in different banks of this memory by a half-warp of threads consumes only a few cycles. However, if the locations accessed by multiple threads in the same half-warp are in the same bank, such accesses are serialized.

Other members of the memory hierarchy are constant memory and texture memory. The texture memory and the constant memory are of the size, usually of multiple KB. These are useful for data that does not change too often over the execution of a kernel. Data read from these memories are cached on-chip for faster access. Reads to the same location in the constant memory by a warp of threads result in the best possible outcome, whereas reads to different locations by a warp of threads happen serially. Misses on the cache are served from the global memory. Texture memory is useful for applications where memory access patterns of a group of threads exhibit a great deal of spatial locality.

Programming Model Further, the vast majority of operations that graphics coprocessors, GPUs in short, provided were dealing with matrices and vectors. This led researchers to use the operations supported by a GPU to perform non-graphics based general purpose computations. This essentially masks general purpose computations as graphics operations. Given that GPUs had massive data parallelism to offer, in some cases such as sorting [74], database querying [62], these implementations offered a significantly higher throughput compared to corresponding CPU implementations.

For purposes of illustration, we consider the Terasort work of Govindaraju et al. [74]. Govindaraju et al. engineered the standard bitonic sorting algorithm [91] on NVidia 7800 and 7900 GPUs. One of the main steps of bitonic sorting algorithm is to rearrange pairs of elements according to the outcome of their comparison. This

step is achieved by a clever use of texture memory, whereas texture memory is an artefact of graphics processing. Similar efforts can be seen in other early works on general computing [62, 65] and the like apart from computer vision related works such as object culling [76], collision detection [75] and so on.

It can be seen that these early efforts made a strong case for enhancing the programmability of GPUs via direct support of programming languages and associated environments. In general, the big question that faced the community at this stage is to find suitable high-level abstractions that do not required programmers to write programs that do not have to involve low-level hardware access.

Stream based programming models offered an intermediary step in this process. Academic projects such as Brook [30] and Sh [110], and related commercial versions such as Microsoft Accelerator and RapidMind[2] in general attempt to view the GPU as a streaming processor. In this model, a program consists of data elements, streams, and kernels. Kernels are the functions that are applied to a set of data elements in an input stream, thereby producing an output stream.

The CUDA API allows a user to create many threads to execute code on the GPU. Groups of these threads form logical *blocks*, and a collection of blocks make up a *grid*. Tuples of up to three dimensions provide numbers to blocks in a grid. This numbering can be useful to number each thread uniquely and use such a thread number to map to data elements. Each SM gets blocks to execute. The number of blocks that an SMX gets to execute varies across generations of GPUs. Table 2.1 shows the numbers for three generations of GPUs. The blocks themselves consist of logical SIMD groups called *warps*, each containing 32 threads on current hardware. An SM executes one warp at a time. CUDA uses zero-overhead scheduling, enabling warps that stall on memory fetches to swap with another warp.

A kernel specifies the computations that are to be performed by the GPU. Before launching a kernel, all the kernel data must be made available on the GPU global memory by an explicit transfer from the

[2]acquired by Intel in 2009.

Resource	Tesla	Fermi	Kepler
SM or SMX	15	16	84
Cores	2880	512	5376
Global memory	24 GB	6 GB	12 GB
Shared Memory	16 KB	48 KB	48 KB
Registers	16384	32768	65536
Blocks per SMX	8	8	16

Table 2.1: Features of generations of NVidia GPUs

CPU memory. This data transfer is done using the PCIExpress links or the NVLink on custom models. A kernel invocation will hand over the control to the GPU, and the GPU executes the specified GPU code on this data. The control returns to the CPU only after the GPU finishes executing the kernel. Contents of the global memory are not guaranteed to remain unchanged across kernel invocations.

CUDA v3.1 onward supports recursive kernels. Concurrent execution of two or more kernels is also supported only from CUDA v? Concurrent execution enables two or more kernels to share GPU resources via a left-over policy []. Under a typical left-over policy, after the maximum possible number of thread blocks from a grid have been assigned to all SMXs, another kernel can make use of any leftover resources by having blocks from the second kernel to be assigned to SMXs for execution.

Synchronization The user can define barrier synchronization for all the threads in a block in the kernel code. Apart from this, all the threads launched in a grid are independent, and the user cannot control their execution order. Global synchronization of all threads is possible only across separate kernel launches. For more details, we refer the interested reader to [1].

Example CUDA Program In this section, we show a quick example of a CUDA program.

```
__global__ void transposeCoalesced(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

2.4 Graph Representations

In the case of sequential computing, graphs are typically represented in a computer program using the adjacency matrix or the adjacency list representation. The adjacency matrix of a graph of n vertices will be a square matrix of size $n \times n$. The entries of the matrix are either $0, 1$ indicating the presence of an edge or a parameter such as the weight of the corresponding edge. This requires a space in $O(n^2)$ which is prohibitive for large graphs. To remedy the space issue, one often uses the adjacency list representation of a graph where the neighbors of a vertex are arranged in a (singly) linked list. These linked lists of neighbors are then stored in an array of size n with the ith element of the array storing the linked list of the neighbors

of vertex numbered i. The space used by this data structure is in $O(m+n)$ and is therefore space-efficient.

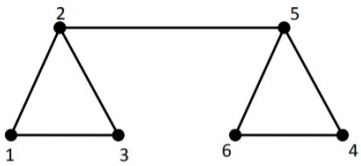
In the parallel setting, one often cannot use pointer based data structures since memory references may not be all local to a core. Therefore, the adjacency list representation is not convenient to use. For space efficiency reasons, the adjacency matrix representation is also not viable. In the following, we illustrate some of the representation mechanisms for stroing graphs in parallel programs.

Edge List

The edge list representation of a graph simply stores the list of edges of the graph. For the graph in Figure 2.7(a), the corresponding edge list is shown in Figure 2.7(b). Some algorithms may also keep the edge list in a sorted order as shown in Figure 2.7(c). The space used by this representaiton is in $O(m)$.

Compressed Sparse Row

In CSR, all the adjacency lists are packed into a single large array. An array E_a is used to store the adjacency lists where the list for vertex $i+1$ immediately follows vertex i, for all the vertices in $\mathcal{G}$. An array V_a, stores the starting indices of the corresponding adjacency lists in E_a. Each of the indices of V_a acts as the vertex number of of the graph. The key advantage of using this representation is that the graph is stored in a contiguous memory locations and no long strides are required to go from a neighbor of a certain vertex. This helps in reducing the memory access irregularity in general. An example is shown in Figure 2.8.



(a) A graph on six vertices.

1,2	5,6	2,5	1,3	4,5	3,2	4,6

(b) The corresponding edge list representation.

1,2	1,3	2,5	3,2	4,5	4,6	5,6

(c) The corresponding sorted edge list representation.

Figure 2.7: The edge list and the sorted edge list representation of a graph.

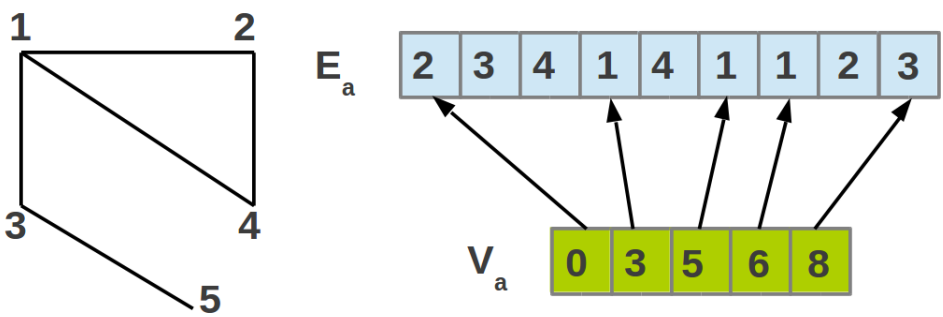


Figure 2.8: The CSR format for representation.

Chapter 3

Graph Decompositions

Graph decompositions are a useful concept in parallel algorithms as they indirectly help break the problem into smaller subproblems. In this chapter, we study the ear decomposition of graphs. In both these cases, we show how identifying critical properties of graphs and the decompositions can help arriving at faster algorithms in practice by reducing the amount of computation required.

3.1 Ear Decomposition

We start by defining an ear decomposition of a graph, which is a partition of the edges of the graph. Ear decomposition of graphs has found applications for testing 2-connectivity and outerplanarity [99].

Definition 1 *An ear decomposition of a biconnected graph $G = (V, E)$ is an ordered partitioning of the edges of G into simple paths (ears) $P_0, P_1, \cdots$ as follows (see also [126]).*

- *P_0 consists of a single edge uv,*
- *$P_0 \cup P_1$ is a simple cycle, and*

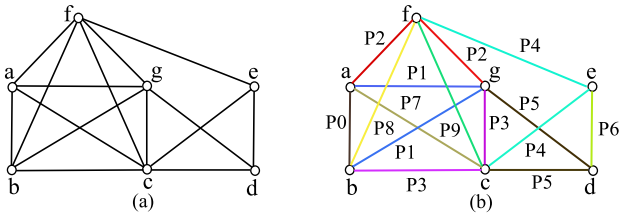


Figure 3.1: An example of an ear decomposition. The labels on edges in part (b) of the figure indicate the ear number they belong to.

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

Figure 3.1 shown an example where the labels on the edges indicate the ear number that the edge belongs to. Observe that an ear decomposition is not unique.

In the context of Definition 1, let us denote by u_i and v_i the end points of the ear P_i for $i \geq 0$. If for each $i \geq 0$, p_i and q_i are distinct, then the ear P_i is called an open ear. Otherwise, the ear P_i is closed. An open ear decomposition of a graph G is an ear decomposition of G in which every ear P_i for $i \geq 0$ is open.

Whitney [160] showed that a graph G has an open ear decomposition if and only if G is biconnected [160]. Tarjan [147] presents sequential algorithms for obtaining such a decomposition using a Depth-First-Traversal (DFS) of a graph. Since DFS is known to be P-Complete, such an approach is not readily amenable to efficient parallelism. Lov'asz showed that the problem of obtaining an open ear decomposition is in NC [35, 108].

Maon et al. [115], and Miller and Ramachandran [126] adapted the approach of Lovasz [108] to design an efficient parallel algorithm for obtaining an ear decomposition of a graph. The basic approach is as shown in Algorithm 1. The preorder numbers in Line 2 of Algorithm 1 can be computed, for instance, as described in [91] using the Euler tour technique. Given a graph G and a rooted spanning tree T of G

Algorithm 1 EarDecompose(G)($from$ [126])

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

and an edge $e = uv$, we use LCA(e) to denote the Least Common Ancestor (LCA) of the nodes u and v with respect to the tree T.

Using the PRAM model [91], Algorithm 1 has a runtime of $O(\log n)$ and uses $O(m + n)$ work. However, in a practical setting, operations indicated in the algorithm usually suffer from drawbacks mentioned below.

- Computing the preorder numbers of the nodes according to T requires one to use the Euler tour technique [91]. This computation on pointer-based data structures such as linked lists involve a lot of uncoalesced memory accesses that result in poor performance on most modern parallel architectures.

- To identify the labels of the non-tree edges, one needs to compute the LCA of the endpoints of every non-tree edge. To achieve an $O(\log n)$ parallel runtime, the algorithm suggests an $O(\log n)$ time and $O(n)$ work preprocessing based on range minima algorithms for LCA queries. When one considers sparse graphs where the number of LCA queries are small owing fewer

non-tree edges, such algorithms can increase the overhead on the computation.

- The labeling of tree edges also has practical difficulties similar to those mentioned above.

3.1.1 Our Approach for Ear Decomposition

Let $G = (V, E)$ be a biconnected graph. We start by identifying edges of G that are redundant to obtain an ear decomposition. These redundant edges are removed from G to get a subgraph G'. The graph G' will have n nodes and at most $2n - 2$ edges, making G' a sparse graph. We show that an ear decomposition of G' easily extends to an ear decomposition of G. To obtain an ear decomposition of G', we exploit its sparsity to improve on the practical performance of the algorithm of Ramachandran [126].

Identifying Redundant Edges

We consider an edge of G as redundant for obtaining an ear decomposition if e can be included as an ear containing just the edge e. We call such an ear as a *trivial* ear. (See also [126]). Cong and Bader [44] present a characterization for identifying redundant edges with respect to the biconnectivity of a graph. A necessary and sufficient condition for a graph to have an ear decomposition is that the graph should be biconnected. Intuitively, edges redundant for the biconnectivity of a graph can also be redundant for obtaining an ear decomposition. Indeed, as we show in the following lemma, biconnectivity and ear decomposition share the same notion of redundant edges.

Lemma 2 *Let T be a rooted BFS tree of a biconnected graph G and F be a spanning forest of the graph $G \setminus T$. Then, edges in $G \setminus (T \cup F)$ are redundant for the purposes of an ear decomposition.*

Proof: Consider the graph $T \cup F$. According to the characterization of Cong and Bader [44], if the graph G is biconnected, so is the

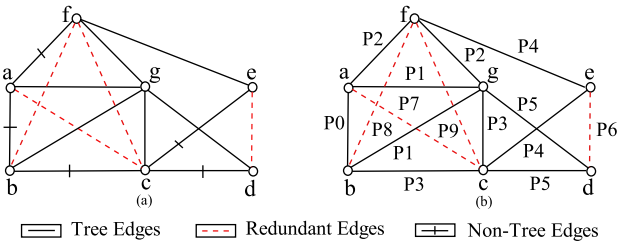


Figure 3.2: An example of using Lemma 2. In the graph on the left, we note that edges shown in dashed lines and red color are redundant. An ear decomposition, as ears P_0 through P_5, of the graph with the rest of the edges is shown in the right part of the figure. As the lemma shows, ears P_6 through P_9 correspond to trivial ears of the redundant edges.

graph $T \cup F$. Therefore, if G is biconnected, then $T \cup F$ has an ear decomposition. Let $(P_0, P_1, \cdots, P_s)$ be an ear decomposition of the graph $T \cup F$.

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

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

An example is illustrated in Figure 3.2. Since T contains $n-1$ edges and F has at most $n-1$ edges, the above lemma indicates that on a graph G of n nodes and m edges, the number of redundant edges is at least $m - 2n + 2$. The remaining graph has thus at most $2n - 2$ edges.

Algorithm for Ear Decomposition

The earlier section results indicate that obtaining an ear decomposition of a graph G can be achieved by obtaining an ear decomposition of a sparse subgraph G'. In this section, we use properties induced by the sparsity of G' to arrive at an ear decomposition of G' efficiently. Our algorithm uses a preprocessing step followed by employing the algorithm of Ramachandran [126] with some changes, followed by a post-processing step. Algorithm 2 presents our algorithm with a brief description in the subsequent paragraphs.

Algorithm 2 EarDecompose(G)

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

Phase I In Phase I, the pruning step requires a BFS of G and another spanning forest computation on the graph $G \setminus T$. For this, we use the optimized BFS implementation from [114]. As mentioned in Steps 2–4 of Algorithm 2, we first compute a BFS tree, T, of the graph G and a spanning forest F of the graph $G \setminus T$. As mentioned in Lemma

2, we remove all edges in $G \setminus (T \cup F)$ from further consideration. The remaining graph, G', has n nodes and at most $2n - 2$ edges.

Phase II Since the graph is sparse and has at most $2n - 2$ edges, the number of non-tree edges would be at most $n - 1$. Thus, we need to find the LCA for at most $n - 1$ pairs of nodes. Therefore, we note that a preprocessing for LCA queries may not be essential. Instead, the LCA of a pair of nodes u, v can be obtained by walking along the path from u and v to the root of the tree. This simple technique has the advantage that we can perform all the LCA queries in parallel. The one disadvantage of the method is that the time spent for each LCA query will depend on the LCA node's distance. However, we show in a later section that most LCA queries traverse a distance that is indeed small. (See Table 3.1).

Labeling of Tree Edges Notice that the graph for which we obtain an ear decomposition has at most $2n - 2$ edges of which $n - 1$ edges appear as tree edges. The remaining $n - 1$ fundamental cycles contain the $n-1$ tree edges. Therefore, we expect that the total lenght of fundamental cycles, and the number of cycles that pass through any given tree edge is small on average. This is also verified empirically, as shown in Table 3.1.

Hence, the routine LabelTreeEdge in Line 11 of Algorithm 2 proceeds as follows. We list the edges of each fundamental cycle in an array A. Given that we know the length of each fundamental cycle from Steps 7–9 of our approach, we reserve space for each fundamental cycle in the array and calculate the starting index in the array where we write the edges of each cycle. In this step, there would be no need for any synchronization operations.

Each element of the array A is of the form $\langle e, \ell, f \rangle$ where e is the id of a tree edge, ℓ is the level number of the LCA of the endpoints of the non-tree edge f whose fundamental cycle passes through e. We now sort the elements of A lexicographically [91]. In the lexicographic order, all the tuples corresponding to each tree edge e appear contigu-

ous. Hence, we find the minimum in each group using the segmented prefix operation [91].

Phase III – Post-processing In this phase, edges pruned in Phase I will be included in the ear decomposition of G as trivial ears.

3.1.2 Analysis

We now analyse the time taken by Algorithm 2 if it were run as a sequential algorithm. Phase I involves two BFS traversals and runs in $O(m + n)$ time. In Phase II, the bulk of the computation is to assign a label for the edges of the tree T'. For a non-tree edge uv, the time taken in this step depends on the maximum distance of LCA(u, v) to u and to v. This distance is no more than the depth of the tree T', which we denote by d'. Notice that the in the worst-case, $d' = O(n)$, however, our experimental results indicate that d' is usually very small. We refer the reader to [42] for a theoretical analysis of d' in random graphs. The time taken for assigning a label to the tree edges is proportional to the number of fundamental cycles that pass through a given tree edge, denoted c'. While theoretical estimates of c' are not known, in our approach the average value of c' is $O(1)$. Phase III runs in $O(m)$ time. So, the overall time for Algorithm 2 is $O(m + nd' + nc')$.

The pruning in Phase I of Algorithm 2 helps in two ways. Firstly, reducing the number of non-tree reduces to $O(n)$ from $O(m)$, minimizesthe impact of using a naive technique to find the LCA of the end points of non-tree edges. Secondly, the sparse nature of $T \cup F$ results in a small c'. A small c' helps in Algorithm 2 to resort to yet another simple way to label the tree edges. We show evidence for these observations in our experiments in the later section.

As a PRAM algorithm, the work done by Algorithm 2 is estimated as follows. The work done in Phase I is in $O(m+n)$ [22]. In Phase II, the work done is in $O(m+nd'+nc')$. Phase III requires work in $O(m)$. The time taken by Algorithm 2 in the PRAM sense is dominated by Phase I which requires $O(D)$ time, where D is the diameter of the

Graph name	$\|V\|$	$\|E\|$	**Edges Pruned**	**Avg. Dist.**	**ACE**
roadNet-CA	2.0 M	2.7M	15 K	10.42	8.18
roadNet-TX	1.4 M	1.9 M	11 K	10.44	7.77
soc-Epinions1	76K	508 K	294 K	2.17	1.89
patents_main	241K	560 K	185 K	5.07	5.59
coAuthorsDBLP	299 K	977 K	447 K	2.89	4.2
soc-Slashdot0902	82 K	474 K	371 K	2.44	2.72
caidaRouterLevel	192 K	609 K	284 K	3.4	4.3
scircuit	171 K	479 K	83 K	3.12	4.74
soc-sign-epinions	131 K	841 K	527 K	2.52	1.95
p2p-Gnutella31	62 K	147 K	52 K	4.16	4.17
Random Graphs $\mathcal{G}(n,p)$ [25]					
$\mathcal{G}(1M, 10 \times 10^{-6})$	1 M	10 M	8 M	4.86	9.31
$\mathcal{G}(1M, 20 \times 10^{-6})$	1 M	20 M	18 M	3.99	7.79
$\mathcal{G}(1M, 40 \times 10^{-6})$	1 M	40 M	38 M	3.68	6.86
$\mathcal{G}(1M, 80 \times 10^{-6})$	1 M	80 M	77 M	2.99	5.92
$\mathcal{G}(2M, 10 \times 10^{-6})$	2 M	10 M	6 M	5.15	7.6
$\mathcal{G}(2M, 20 \times 10^{-6})$	2 M	20 M	16 M	4.98	9.66
$\mathcal{G}(2M, 40 \times 10^{-6})$	2 M	40 M	36 M	4.13	8.03
$\mathcal{G}(2M, 80 \times 10^{-6})$	2 M	80 M	76 M	3.87	7.3

Table 3.1: List of sparse graphs that we use in our experiments. We round the number of nodes and the edges to the nearest thousand (K) or the nearest million (M). The notation $\mathcal{G}(n,p)$ refers to a random graph ([25]) with n nodes and an edge probability of p. The number in the column labeled "Edges Pruned" shows the number of redundant edges, according to Lemma 2. The column labeled "Avg. Dist." shows the average number of tree edges traversed to find the LCA of the endpoints of a non-tree edge. The column labeled "ACE" indicates the average number of fundamental cycles according to a BFS tree that pass through a tree edge.

input graph [22].

Notice that the time required by existing algorithms for obtaining an ear decomposition run in $O(m+n)$ time in the sequential setting, and have a work complexity of $O(m+n)$ in the PRAM model. While Algorithm 2 has asympototically larger complexity in both these settings, we note that the constants involved in the asymptotic analysis and also other system level issues play a role in the practical performance as we will see in the following section.

3.1.3 Experimental Results

In this section, we use an NVidia Tesla K40c GPU attached to an Intel(R) Xeon(R) E5-2650. See Chapter 2 for more details of these architectures to study the performance of Algorithm 2. We run our experiments on both the GPU and the CPU mentioned above and compare the performance of Algorithm 2 to that of existing algorithms.

Datasets

We use the graphs listed in Table 3.1 for our experiments. The graphs include both real-world graphs from [4] and also Erdos-Reyni random graphs [25] generated using the RMAT generator [37]. Since we require the graph to be biconnected to have an ear decomposition, we make the graphs in Table 3.1 biconnected by adding additional edges as needed. We also remove self-loops and make all the graphs undirected.

Results

We show the performance of our algorithm to that of [126] on a GPU and also on a multi-core CPU.Figure 3.3(a) and 3.3(b) show the absolute runtime as well as the speedup of our algorithm to that of [126] on the K40c GPU and the E5-2650 CPU, respectively. To better understand our algorithm's performance, we also add the pruning step from Algorithm 2 to the algorithm of Ramachandran [126]. We label

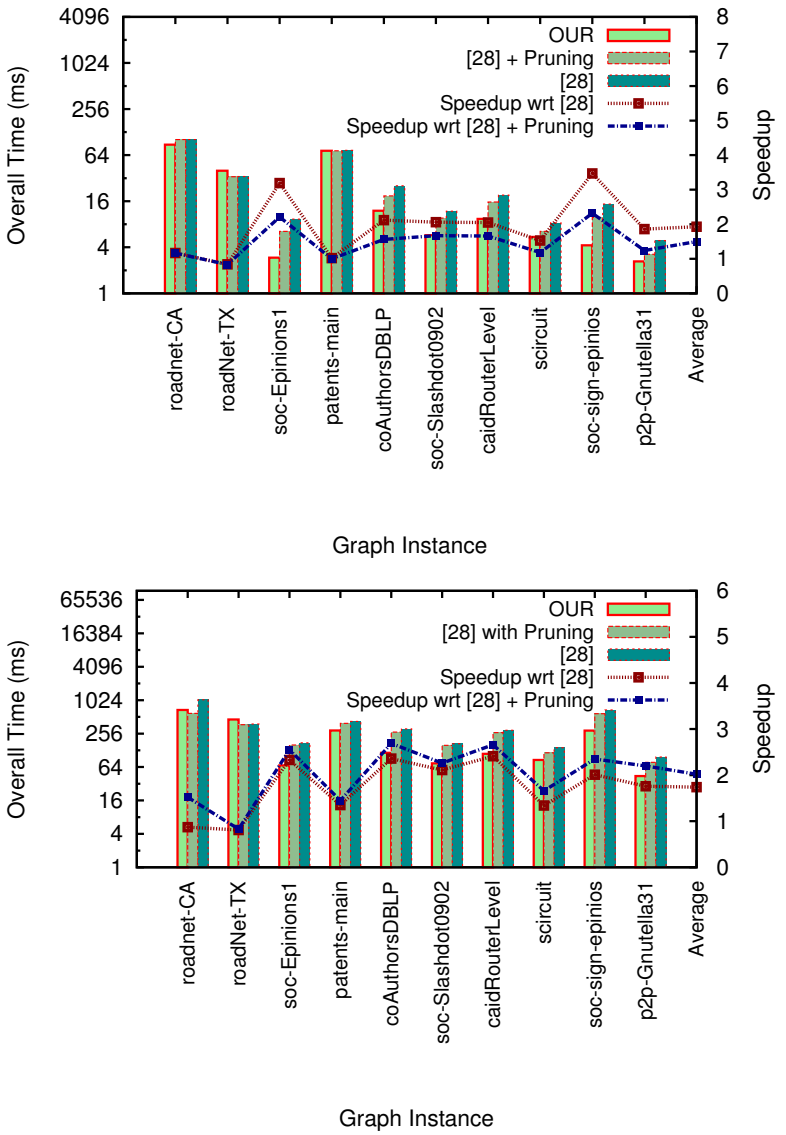


Figure 3.3: Performance of our ear decomposition algorithm on real-world graphs (a) using the K40c GPU and (b) on the Xeon E50-2650 CPU. The last label "Average" indicates the average speedup on the dataset from Table 3.1.

this modification as " [126] with Pruning" in the plot in Figure 3.3(a) and 3.3(b).

We show the speedup on the secondary Y-axis. Note that the Y-axes are on a logarithmic scale. Notice that our algorithm performs 2.3x and 2.02x better than the algorithm of [126] on GPU and CPU respectively. On adding the pruning step to the algorithm of [126], our algorithm outperforms this variation by a factor of 1.54 on GPU and 1.74 on CPU. This indicates that our performance gains are due to both the pruning step and other algorithmic enhancements to that of [126].

As the number of edges increases, Phase I of our algorithm removes a bigger number of edges, reducing the work in the latter phases resulting in a better speedup. Our experiments on random graphs in Figure 3.4(b) and also 3.4(b), where we keep the number of nodes fixed at 1 M and 2 M nodes and increase the number of edges, support this observation.

Comparison with respect to Bader et al. [13] Table 3.2 shows a comparison between our algorithm and the PRAM algorithm. For this comparison, we use a dataset similar to the one used by Bader et al. [13], which has the following types of graphs: regular lattices, regular triangulation graphs, and random planar graphs. Figure 3.5(a) and (b), respectively, show an example of the regular lattice and regular triangulation. The graph generator LEDA [113] is used to generate these graphs. We set $|V|= 8192$ number of vertices as in [13] and eight threads for processing to keep the comparison fair. As shown in Table 3.2, our algorithm is performing better than that of [13] except in the case of regular lattices. This behavior on regular lattices is due to the large average number of tree edges traversed to find the LCA of the endpoints of non-tree edges.

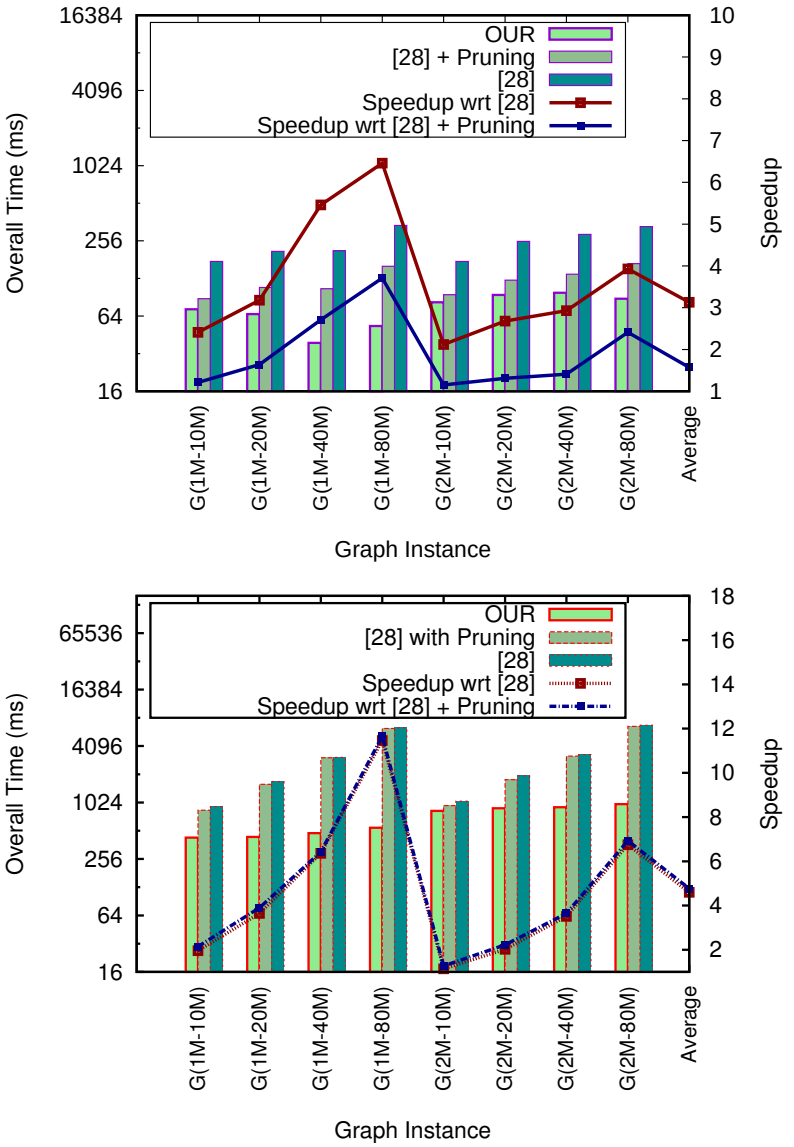


Figure 3.4: CPU Performance of our ear decomposition algorithm on random graphs (a) using the K40c GPU and (b) on the Xeon E50-2650 CPU. The last label "Average" indicates the average speedup on the dataset from Table 3.1.

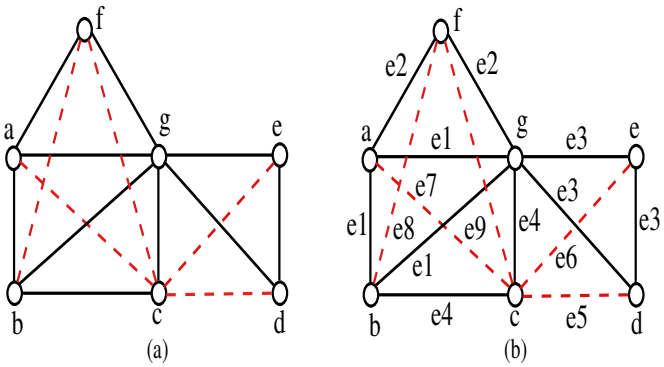


Figure 3.5: Part (a) shows regular lattice RL and part (b) shows regular triangulation RT graph.

Graph name	$\|V\|$	$\|E\|$	**OUR**	**[13]**	**Avg. Dist.**
Random Graph A (Planar)	5965	7686	0.791	1.9	4.21
Random Graph B (Planar)	8192	16375	1.030	1.9	3.67
Random Graph C (Planar)	8192	24570	1.345	1.9	3.39
Random Graph D (Planar)	8192	24570	1.349	1.9	3.38
Regular Triangulation	8192	24551	1.422	1.9	4.12
Regular Lattice	8192	16199	15.0128	1.9	90

Table 3.2: Table shows the comparison of our algorithm to that of [13]. Columns three and four show the time (in ms) for OUR approach and that of [13]. The last column indicates the average number of tree edges traversed to find the LCA of the endpoints of a non-tree edge.

Chapter 4

Graph Connectivity

Finding whether a graph is k-connected, and the identification of its k-connected components is a fundamental problem in graph theory. For this reason, there have been several algorithms for this problem in both sequential and parallel settings.

4.1 Introduction

Finding whether a graph is k-connected and obtaining the k-connected components of a graph is a fundamental problem with a variety of applications including planarity testing [79], isomorphism in planar graphs [86], network analytics [27, 73, 154], clustering [33] and data visualization [5]. It is therefore not surprising that several researchers have explored this problem in various settings such as sequential algorithms [38,47,87,96,147], parallel algorithms [82,96,100,116,137,147], and implementations [36,45,142,153], and also distributed algorithms [121]. Most of these algorithms use graph traversal techniques to create one (or more) spanning tree(s) and use the properties of the spanning trees to test the k-connectivity of the graph and obtain its k connected components.

In particular, in the parallel setting, PRAM algorithms that require poly-logarithmic time and work in $O(m + n)$ are known for

$k = 1, 2, 3$, and 4 [82, 96, 116, 137, 147]. However, in practice, the constants hidden in the big–O notation are significantly high for $k \geq 2$. Therefore, algorithms that run faster in practice are sought after.

In this chapter, we first review some of the existing parallel algorithms for connectivity and graph biconnecitivity. Due to its varied applications, testing graph k-connectivity has been a problem of immense research interest. Early PRAM algorithms for testing the connectivity of a graph were proposed by Hirschberg et al. [82] and Shiloach and Vishkin [137]. The algorithm of Shiloach and Vishkin is shown to run in $O(\log n)$ time using $O(n + m)$ work in the PRAM model. Several experimental studies on finding the connected components of a graph are based on this algorithm [77, 140, 145].

The first PRAM algorithm for finding the 2-connected components of a graph in parallel is given by Tarjan and Vishkin [128]. This algorithm reduces the problem of finding the 2-connected components of a given graph to finding the connected components of an auxiliary graph. The construction of the auxiliary graph is shown to be in $O(\log n)$ time using $O(m + n)$ processors. It is identified by Bader and Cong [45] that the process of constructing the auxiliary graph is however quite slow in practice. Bader and Cong [45] proceed to use a formulation akin to that of Cheriyan and Thurimella [38] and show a speed-up of up to 2x on a variety of graphs on multi-core CPUs. More recently, Slota and Madduri [142] proposed that one can test the biconnectivity of a graph by performing multiple BFS traversals on multi-core CPUs. This result has been subsequently improved by Chaitanya and Kothapalli [36]. The approach of Chaitanya and Kothapalli [36] is then adapted to work on GPUs by Wadwekar and Kothapalli [153].

4.1.1 Organization of the chapter

The rest of the chapter is organized as follows. Section 5.2 presents our technique in brief. Sections 5.4–5.5 discuss our approach applied to the problem of 2, and 3-connectivity respectively. The chapter then ends with concluding remarks in Section 5.6.

4.2 Existing Algorithms

We review in brief some of the parallel algorithms that are most relevant to our present work.

4.2.1 Shiloach-Vishkin Algorithm [138] for Graph Connectivity

A common feature of all parallel algorithms for connected components is to maintain a partition of the vertices of the graph so that nodes in the same partition belong to the same connected component. The partition is refined iteratively and termination is achieved when no further refinement to the partition is possible. Initially, each node is in its own connected component, and hence its own partition. The partitions are also typically maintained as trees with the root of the tree serving as a representative of that partition. Different algorithms [19, 83, 139] differ in the processing done during each iteration.

The Shiloach-Vishkin algorithm [139] involves iterative grafting and pointer jumping operations. In each iteration, if (u, v) is an edge in the graph, then, under certain condition, the trees containing nodes u and v are combined to form a single tree. This process is called *grafting*. Further, during each iteration, pointer jumping is applied to reduce the height of the resulting trees. The algorithm terminates when all the trees in the forest are stars, and each node is assigned to one star. In each iteration the following steps are performed:

- **Grafting trees:** For each edge uv so that parents of u and v are different, one node changes parent, if parent of either u or v is the root of its tree and the parent of the other node has a lower index than the former.

- **Grafting star trees onto other trees:** This is done to reduce the depth of the resultant trees. The trees are checked to ascertain whether they are stars or not by allowing nodes which are at a depth of 2 or larger with respect to the root of the tree to mark its parent and the parent of its parents as members of

non stars. Thus all the nodes which are not part of a star will be marked by this process. This step reduces the worst case complexity of the algorithm.

- **Single pointer jumping:** One step of pointer jumping is done to reduce the depth of the trees.

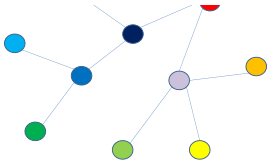
It is shown by Shiolach and Vishkin that this algorithm runs in $O(\log n)$ time using $O(m + n)$ operations. Figure 4.1 provides an illustration of the running of the Shiloach and Vishkin algorithm.

4.2.2 Tarjan-Vishkin Parallel Algorithm (TV) [149] for Graph 2-Connecitivity

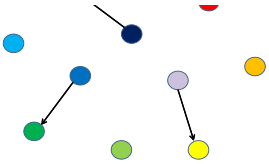
The algorithm of Tarjan and Vishkin for identifying the biconnected components in G is designed for the PRAM model [149] and requires $\mathcal{O}(\log(n))$ time with $\mathcal{O}(n + m)$ processors. The main steps in the Tarjan-Vishkin algorithm are as follows. A rooted spanning tree T for the input graph G is constructed (For notations interested reader can refer to [158]). Using T, two functions `low`, the lowest vertex that is either a descendant of v or adjacent to the descendant of v by an edge in $G \setminus T$, and `high`, the highest vertex that is either a descendant of v or adjacent to the descendant of v by an edge in $G \setminus T$, are computed for each vertex $v \in G$. These functions help define an auxiliary graph G' where the nodes of G' are the edges of G. The edges of G' are defined as follows.

- An edge connecting vertices uw and vw in G' is added to $E(G')$ whenever there is a tree edge $u \rightarrow w$ with u as the parent of w and a nontree edge vw with $\text{pre}(v) < \text{pre}(w)$ where pre(.) is the preorder number of a vertex according to the tree T.

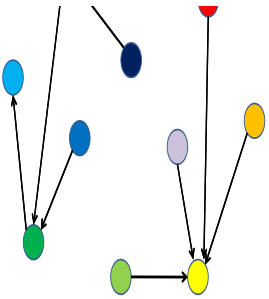
- An edge connecting vertices uv to xw in G' is added to $E(G')$ whenever there is a tree edge $u \rightarrow v$ with u as the parent of v and a tree edge xw with x as the parent of w and a nontree edge vw with v and w not having an ancestor-descendant relationship in T.



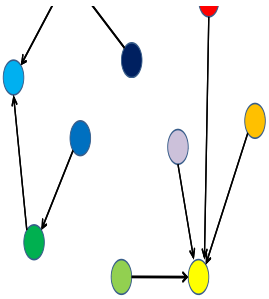
(a) Initial Graph



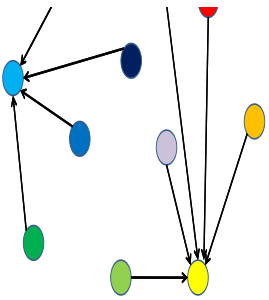
(b) First Hooking and Complete Pointer Jumping



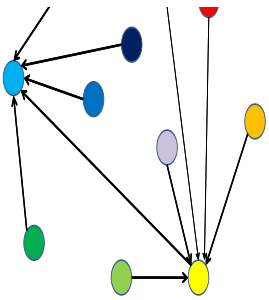
(c) General Hooking



(d) Pointer Jumping for inner nodes



(e) Pointer Jumping for leaf nodes



(f) Final Hooking

- An edge connecting uv to vw in G' whenever there is a tree edge $u \to v$ with u as the parent of v and a tree edge vw with v as the parent of w and a nontree edge joins a descendant of w to a non-descendant of v in T.

The idea behind the definition of G' is such that the connected components of G' are the biconnected components of G. To this end, Tarjan and Vishkin show that for edges that lie on some cycle of G, their corresponding vertices in G' are all in the same connected component of G'.

4.2.3 Cong and Bader [44] Improvement to TV (TV-filter)

An experimental study by Cong and Bader [44] provides an improvement to the Tarjan-Vishkin ([149]) by removing non-essential edges in its computation. This leads to a significant reduction in computing `low`, `high` values and connected components computation. In particular, they define an edge e as *non-essential* for biconnectivity if removing e does not change the biconnectivity of the component it belongs to. They show that edges of G that are not in a BFS tree T and are also not in a spanning forest F of $G \setminus T$ are non-essential. However, in this algorithm, T must be a BFS tree, which can be difficult to compute in parallel compared to a simple spanning tree. The runtime for the Cong-Bader approach is $\mathcal{O}(dia + \log n)$ where dia is the diameter of the graph. The algorithm is as follows.

4.2.4 BFS-BiCC

The BFS-BiCC algorithm [94] is an improvement over Cong-Bader's approach [44] on sparse graphs. This algorithm is similar to that used by Eckstein [51]. A rooted BFS tree T of the input graph G is constructed. To identify the articulation points, the BFS-BiCC algorithm considers every vertex u and performs a BFS on the graph after removing its parent $P(u)$ (according to T) from the graph G. During this process BFS-BiCC algorithm keeps track of the level of

Algorithm 3 CONG-BADER(G)

```
1: procedure CONG-BADER(Graph G)
2:     T ← BFS(G)  /*Phase I*/
3:     F ← BFS(G\T)
4:     G' ← (F∪T)
5:     Biconnected_Components = TARJANVISHKIN-PARALLEL(G')
   /*Phase II*/
6:     for all e = (w,v) ∈ G \ (F ∪ T) do  /*Phase III*/
7:         label e to be in biconnected component containing w and
   p(w)
8:     end for
9: end procedure
```

vertices reached from u. If any vertex w with level $L(P(u))$ or less is reached and u has a path to all its siblings after the removal of $P(u)$, then $P(u)$ is not an articulation point. The main drawback of this approach is that it performs BFS from all the vertices of the graph. While there are optimizations introduced to stop some of these breadth-first traversals, there exist graphs on which such early stopping cannot be done and on such graphs, Algorithm BFS-BiCC from [94] suffers heavily.

4.2.5 Color-BiCC

The Color-BiCC algorithm [94] is an improvement over the Cong and Bader approach [44] on sparse graphs. This is an iterative strategy that is similar to recursive doubling used to compute connected components in undirected graphs as well as weakly and strongly connected components in directed graphs [143]. Let $par(v)$ signify the parent articulation point that separates the vertex v from the root. This algorithm aqis based on the observation that any two vertices in a biconnected component will have their least common ancestor set to the $par(v)$. The goal of this approach is to color all vertices in the biconnected component with a parent level articulation point

that is separating the vertex from the root. As we will show in Figure 4.7 that the Color-Bicc algorithm is heavily dependent on the structure of the graph and can be slower than the sequential Tarjan [148] approach in some cases.

4.3 Our Approach for a GPU Algorithm for Graph Connectivity

Connected components is considered an irregular memory access algorithm (irregular algorithm), which is not a good fit for the GPU computational model which relies heavily on regularity of memory access. The CUDA Programming Guide [1] provides a detailed description of the effects of irregular and regular memory access on GPU performance.

The focus of any algorithm designed for the GPU relies on regular/coalesced memory accesses and increasing computation, focusing on data movement in the shared memory. The requirements for connected components and GPU computational model are thus orthogonal to each other. Thus mapping the connected components algorithm on to the GPU is non trivial.

The work presented here tries to reduce irregular memory access based on the guidelines of algorithms of [19, 78, 139]. While designing the algorithm, the following pronciples were considered,

- Reducing Atomiic operations
- Reducing the overhead caused by the 64 bit reads for the end points of the edges
- Allowing partial results from previous iterations to reduce redundant computations

We have focused on developing an algorithm that reduces irregular memory accesses, and allows transfer of information from one iteration to the next, hence streamlining the algorithm.

The Shiloach-Vishkin algorithm as proposed [139] may not be quite suitable on modern architectures such as the GPU. One of the reasons for this is the excessive number of irregular reads during the grafting and the pointer jumping steps. An important point to be noted here is that the average number of iterations till which an edge is active is large. Also, if an edge has taken part in a grafting step once, it is not necessary that it cannot participate in another iteration of grafting. Hence, we require a method that requires an edge to participate in grafting atmost once. The number of reads of the full edgelist should be minimal, and the pointer jumping step is further optimized. In this section, we devise ways to handle the given constraints, thereby improving the performance on the GPU.

Our adaptation follows three main steps: 1) Hooking 2) Pointer Jumping 3) Graph Contraction. Hooking is the time intensive operation in the algorithm. It tries to connect elements of the same component together. Pointer jumping tries to find the root node for each component after the hooking step. The fundamental building blocks of our algorithm are as follows:

1. **Hooking:** Hooking is the process of selecting a vertex as the parent of another. For each edge, if the two ends lie in different components, it tries to connect the root nodes of the two components. The orientation of the connection is based upon the labels of the indexes. The orientation is varied across iterations. As the hooking process is randomized, the sensitivity to vertex labels is reduced. We use the alternating orientation hooking process as mentioned in HY1 of [78]. In even iterations, the node with lower label selects the node with higher label as its parent and the reverse happens in odd iterations. Figure b,c and f in Figure 4.1, represent the hooking steps of the algorithm, note that in the first iteration, the node attaches to the neighbour instead of the parent of its neighbour.

2. **Pointer Jumping:** Instead of one step of pointer jumping as originally proposed in the Shiloach-Vishin algorithm [139], we

perform complete pointer jumping. In complete pointer jumping, each tree is reduced to a star in every iteration. Figure **??** illustrates the difference. INCLUDE A FIGURE HERE. We add that the HY1 algorithm of Greiner [78] also uses complete pointer jumping.

Further, when we use complete pointer jumping, notice that nodes in a tree are at only two levels: root node and internal nodes or leaf nodes. Also, after a tree is grafted onto another tree, only the root nodes in one tree have to perform pointer jumping. This is illustrated in Figure d and e of Figure 4.1. This helps in improving the performance on the GPU.

Though the number of iterations required remains the same as SV [139] and AS [19], we observe that the number of read write operations decrease significantly.

3. **Graph Contraction using Edge Hiding:** Explicit graph contraction requires large amount of data movement across the global memory of the GPU, which is a costly operation. The time taken by such a process only increases the run time of the algorithm. Hence an implicit method of hiding edges from the graph is used. As edges are only active in the hooking stage, edges which are known to be in the same component are inactivated in the next stage of hooking.

Theoretically complete algorithm is presented in Algorithm 4.2. Our algorithm can be seen as a modification of HY1 of [78] for the GPU. The algorithm presented here runs in worst case of $O(\log n)$ iterations, and $O(\log^2 n)$ time, doing $O((m+n)\log n)$ work on a PRAM model. The pointer jumping step takes a total of $O(n \log n)$ of work, using $O(\log^2 n)$ time.

4.4 Our Approach for BiCC on Sparse Graphs

In this section we present a simple yet efficient algorithm for identifying the biconnected components of graph G. Our algorithm is based

```
Begin
    Initialize Parent[i] = i;
    while All edges are not marked do
        for each edge (u, v)
            if edge is unmarked and Parent[u] ≠ Parent[v]
                max=max{Parent[u], Parent[v]};
                min=min{Parent[u], Parent[v]};
                if iteration is even Parent[min] = max;
                else Parent[max] = min;
            else Mark edge (u, v);
        end-for
    end-while
    Multilevel_pointer_jumping();
end.
```

Figure 4.2: Algorithm for Connected Components

on our experimental evidence that identifying bridges of a graph in a parallel setting is a much easier and simpler task. Based on the above observation, we initially decompose the graph into maximal 2-edge-connected components $G_1, G_2, \cdots$. For each such component, G_i, $i \geq 1$, we construct an auxiliary graph G'_i where articulation points in G_i translate to bridges in G'_i. Therefore, identifying the bridges of G'_i allows us to identify the articulation points of G_i, and hence those of G. Using this information, we then identify the biconnected components of G.

We develop two results (Lemmata 3, 4 and 5) below that will help us present our algorithm. Towards this, let T be a rooted BFS tree of G and LCA denotes the least common ancestor. Each non-tree edge (u, v) in $G \setminus T$ is a cross edge that connects two different branches of a tree. For an edge e to be a bridge, e must be part of BFS spanning tree and e cannot be on any cycle induced by the non-tree edges $(u, v) \in G \setminus T$.

Now, consider the graph G' obtained by removing the bridges of

G. The resulting graph consists of maximal 2-edge connected components $G_1, G_2, \cdots$, such that for each pair of vertices u, v in the same 2-edge-connected component, there are at least two edge disjoint paths between u and v. We can now treat each such component independently and in parallel to identify the articulation points within each component. These will also be articulation points of G.

Let G be a 2-edge-connected graph and T_G be a rooted BFS tree of G. We use the notation $V_{\text{lca}}(G)$ to denote the set of vertices that are the LCA of the end points of some non-tree edge of G according to a given BFS on G. We can classify the vertices of G into two categories as follows.

1. Potential articulation points: We will prove shortly that all the vertices $V_{\text{lca}}(G)$ belong to this category. A subset of the vertices in $V_{\text{lca}}(G)$ is the set of articulation points of G.

2. Non-articulation points: These are the set of safe vertices whose removal does not disconnect the graph. All the vertices $v \in V(G) \setminus V_{\text{lca}}(G)$ belong to this set.

The above categorization is supported by the following lemma which shows that vertices not in $V_{\text{lca}}(G)$ cannot be articulation points in G.

Lemma 3 *Let G be a 2-edge-connected graph and let T be a BFS tree of G. If v is an articulation point of G, then $v \in V_{lca}(G)$.*

Proof: On the contrary, assume that a vertex v is an articulation point and is not the LCA of any non-tree edge of G. If v is on only one cycle in G, then v cannot be an articulation point. So, we assume in the rest of the proof that v is on at least two cycles in G. Let $C_1, C_2, \cdots, C_k$ be the fundamental cycles induced respectively by non-tree edges $e_1, e_2, \cdots, e_k \in G \setminus T$ and pass through vertex v. Let C_i and C_j be *any* two cycles from the set $\{C_1, C_2, ..., C_k\}$ induced by non-tree edges e_i and e_j respectively. Let vertices x, y be LCA of the endpoints of e_i and e_j respectively. It is evident that x and y should

be the ancestors of v as v lies on both the cycles and $v \notin \{x, y\}$. The relation between x and y can be categorized as follows.

- $x = y$: In this case the two cycles C_i and C_j share the same LCA say x and also the vertex v. This implies that C_i and C_j share at least an edge (as there are at least two vertices, x and v, common to both C_i and C_j). So, even after the removal of v, all edges belonging to C_i and C_j remain in a single biconnected component. Hence, v is not an articulation point.

- $x \neq y$, $z = LCA(x, y)$, and $z \notin \{x, y\}$: As x and y are ancestors of v there is a path x to v and v to y in T. As z is the ancestor of x and y there is a path z to x and y to z in T. This concludes that there is a path from $z \rightsquigarrow x \rightsquigarrow v \rightsquigarrow y \rightsquigarrow z$ which leads to a cycle in T. However, T is a BFS tree and cannot have cycles. Therefore, our assumption that v is an articulation point is not valid.

- $x \neq y$ and $\text{LCA}(x, y) \in \{x, y\}$: Without loss of generality, we will assume that $y = \text{LCA}(x, y)$. Let C_i and C_j be any pair of cycles induced by non-tree edges e_i and e_j and pass through v with $\text{LCA}(e_i) = x$ and $\text{LCA}(e_j) = y$. Since y is a proper ancestor of x, there is a path from $x \rightsquigarrow v$ (in T and also in G) that is common to C_i and C_j. This ensures that there is at least an edge common between the cycles C_i and C_j. Similar to the case where $x = y$, this allows us to argue that even after the removal of v, all edges of C_i and C_j remain in a single connected component. Since the above holds for any pair of cycles passing through v, v is not an articulation point.

□

The above lemma indicates that we only have to check whether vertices in $V_{\text{lca}}(G)$ of G are articulation points in G. To find these articulation points, we now construct an auxiliary graph G' as follows. Let T_G be a BFS tree of G. We use the notation $\text{LCA}(e)$ to refer to the LCA of the end points of the edge e. (Such a notation is used in other earlier works too [126]).

Initialize $V(G') = V(G)$ and $E(G') = E(G)$. For every non-tree edge e in $G \setminus T_G$, compute the $LCA(e) = x$. Let b_1 and b_2 be the neighbours (also called as base vertices) of x in the fundamental cycle induced by e. We now remove the edges xb_1 and xb_2 from G', add an *alias* vertex for vertex x as x' to G', and add edges xx', $x'b_1$, and $x'b_2$ to $E(G')$. All other edges in G with end points as x, b_1 or b_2, remain unchanged. Also, if k cycles $C_1, C_2, \cdots, C_k$ induced by the non-tree edges $e_1, e_2, \cdots, e_k$, respectively share any of the base vertices, then their common LCA vertex, say v, cannot be an articulation point with respect to the cycles $C_1, C_2, \cdots, C_k$ as all these cycles share at least an edge. Thus, we create only one alias vertex v' in the auxiliary graph for v with respect to the above k cycles.

The alias vertices and edges with one end point as an alias vertex have the property that articulation points of G are transformed as bridges in the auxiliary graph G' as we will show shortly. An example of the construction of the auxiliary graph is shown in Figure 4.3. In Figure 4.3, we consider a BFS of the graph in Figure 4.3(a) with z as the source vertex, and edges wx and ts as the non-tree edges. In the following, we show that bridges in G' can be used to identify the articulation points of G.

Lemma 4 *Let G be a 2-edge-connected graph, T_G a rooted BFS tree of G with root as r, and G' be the auxiliary graph of G constructed as earlier. For vertices u in G' with $u \neq r$, u is an articulation point of G iff u is an end point of some bridge uv in G' with $u \in G$ and $v \notin G$.*

Proof:

(only − if) ⇐: Consider a vertex u which is not an articulation point in graph G with $u \neq r$. We will show that any edge of type uu', where u' is the alias of u, cannot be a bridge in auxiliary graph G'.

Let $\mathcal{C} := \{C_1, C_2, \cdots, C_k\}$ be the cycles that pass through vertex u in G. The relation between vertex u and the such cycles can be categorized as follows.

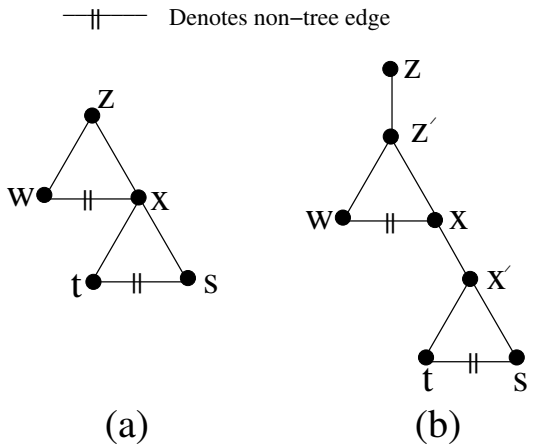


Figure 4.3: Figure (b) shows the auxiliary graph for Figure (a). Edges wx and ts are kept as non-tree edges and the rest of the edges are tree edges according to a BFS starting from vertex z as the source.

- u is not the LCA of any of the end points of non-tree edges that induces cycles in $\mathcal{C}$: In this case, no alias vertices are introduced in G' due to u. Therefore, bridges with u as one end points does not exist in G'. (Note that G is already 2-edge-connected and has no bridges).

- u is the LCA any two non-tree edges that induces cycles C_i and C_j in $\mathcal{C}$: According to the construction of G', two alias vertices u_i and u_j are introduced in the auxiliary graph G'. Further, two edges uu_i and uu_j are also added to G'. An example is illustrated in Figure 4.4.

  Let x and y be any two distinct vertices on the cycles C_i and C_j respectively. Since u is not an articulation point in G, there must be some path P_{xy} in G' between x and y that does not pass through u as shown in Figure 4.4. The path P_{xy} along with paths P_{yu_j}, $P_{u_ju_i}$, and P_{u_ix} forms a simple cycle in G'. This indicates that edges uu_i and uu_j on this cycle cannot be bridges

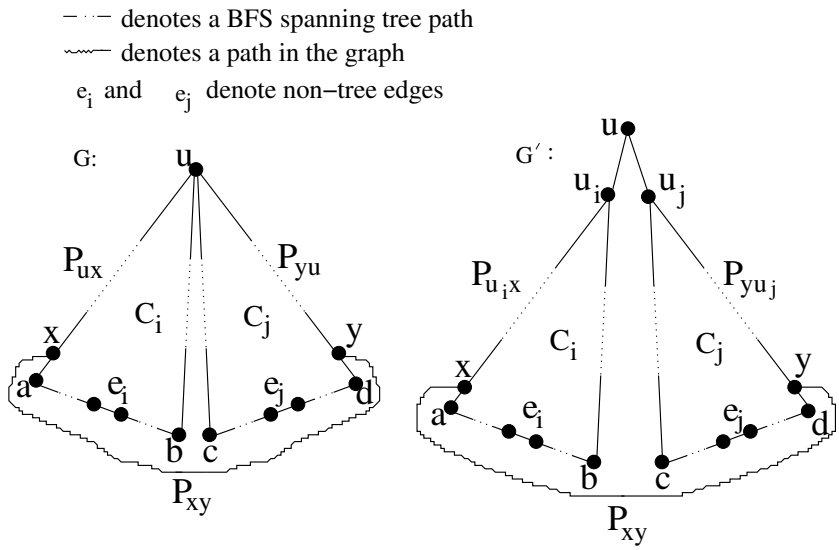


Figure 4.4: Figure shows the cycle created by paths P_{xy}, P_{yu_j}, $P_{u_ju_i}$, and P_{u_ix}. For ease of exposition, the auxiliary graph shown contains only the changes made with respect to u and not the changes induced with respect to other vertices.

in G'. So there is no bridge in G' with one of the endpoint as u pertaining to cycles C_i and C_j. The above property holds for any two cycles C_i and C_j.

- u is the LCA of some non-tree edge that induces cycle C_i in $\mathcal{C}$: Consider the case where the number of cycles through u is at least 2. By our assumption, u is not an articulation point. Let u_1 be the alias vertex of u. Hence, for some vertex x in C_i that is not equal to u, and another vertex, say the parent of u, there is a path that does not go through u. This path along with edges uu_1 and $uP(u)$ mean that the edge uu_1 is part of a cycle. Therefore, in G', the edge uu_1 will not be a bridge.

(**if**) $\Rightarrow$: Let u be an articulation point of a 2-edge-connected graph G

and let G' be the corresponding auxiliary graph. It holds that u has at least two cycles passing through it and not sharing any of the base vertices. Let b_i and b_j be one of the base vertices on two fundamental cycles. Let x and y be two vertices, both distinct from u, on two such cycles C_i and C_j that get disconnected by the removal of u. Since u is an articulation point in G, there exists only one path between x and y that passes through u, say, $x - b_i$, $b_i - u$, $u - b_j$ and $b_j - y$. Let u_1 and u_2 be any two alias points created for C_i and C_j. The corresponding path $P_{xy}(G')$ has the form $x - b_1$, $b_1 - u_1$, $u_1 - u$, $u - u_2$, $u_2 - b_2$, $b_2 - y$. Since u is an articulation point, there cannot be a path P_{xy} between x and y in G (and in its corresponding auxiliary graph G') that does not pass through u and its alias vertices. As a result uu_1 remains uncovered in any cycle of G'. Hence, uu_1 will be a bridge in G'.
□

Lemma 5 *Let G be a 2-edge-connected graph, T_G a rooted BFS tree of G with root as r, and G' be the auxiliary graph of G constructed as earlier. Vertex r is an articulation point in G iff r is the LCA of more than one non-tree edge of G' according to a BFS in G' from r, and r is also an end point of some bridge in G'.*

Proof: We use $P_{uv}(G)$ to denote a path between vertices u and v in the graph G.

(only − if) ⇐: Notice that since G is 2-edge-connected, vertex r is on at least one cycle. Further, since r is the root of the BFS tree of G, for every fundamental cycle that contains r, vertex r is the LCA of the non-tree edge that induces the cycle. We now make a case distinction as follows. Now consider the case that more than one cycle passes through r. Let C_i and C_j be any two cycles through r induced by non-tree edges e_i and e_j. In G', we now introduce two alias vertices r_i and r_j and also the edges rr_i and rr_j, along with edges between r_i and r_j to the base vertices of C_i and C_j. If r is not an articulation point, then we notice that there are any two distinct vertices x and y

in C_i and C_j respectively such that there is a path between x and y that does not go through r. This path between x and y, P_{xy}, along with paths P_{xr_i}, the edges $r_i r$ and rr_j, and path $P_{r_j y}$ creates a cycle that contains the edges rr_i and rr_j. Therefore, the edges rr_i and rr_j cannot be bridges in G'.

(**if**) $\Rightarrow$: The same argument from proof of (**if**) part of Lemma 4 holds true when u is an articulation point and is the root of the spanning tree.
□

Thus, bridges in the auxiliary graph are a good indicator of articulation points in the original 2-edge-connected graph. Further, it is relatively easy to find the biconnected components of a graph when the bridges and articulation points are identified. Our approach also indicates that the size of auxiliary graph in terms of both the number of vertices and the number of edges, is more than the size of original graph. But, we will see later that for real-world graphs that are sparse in nature, this increase in size is usually small, and the additional increase in run time can be offset by the simplicity of our algorithm.

4.4.1 Our Algorithm for Biconnected Components

Given Lemmata 3, 4 and 5 we now provide the following algorithm to identify the biconnected components of a graph G. Algorithm LCA-BiCC shown as Algorithm 4 describes our approach in a high level as consisting of 5 steps. In Step 1, we obtain a BFS tree of the input graph G. In Step 2, we find the bridges of G and also decompose G into its 2-edge-connected components $G_1, G_2, \cdots$. Step 3 onwards, each such 2-edge connected components is treated independently. In step 3, for each G_i, an auxiliary graph G'_i is constructed. Step 4 identifies non-tree edges of each G'_i, and Step 5 identifies the bridges of the auxiliary graph, and hence the articulation points of G. In Step 6, the bridges and the articulation points of G are used to identify the biconnected components of G.

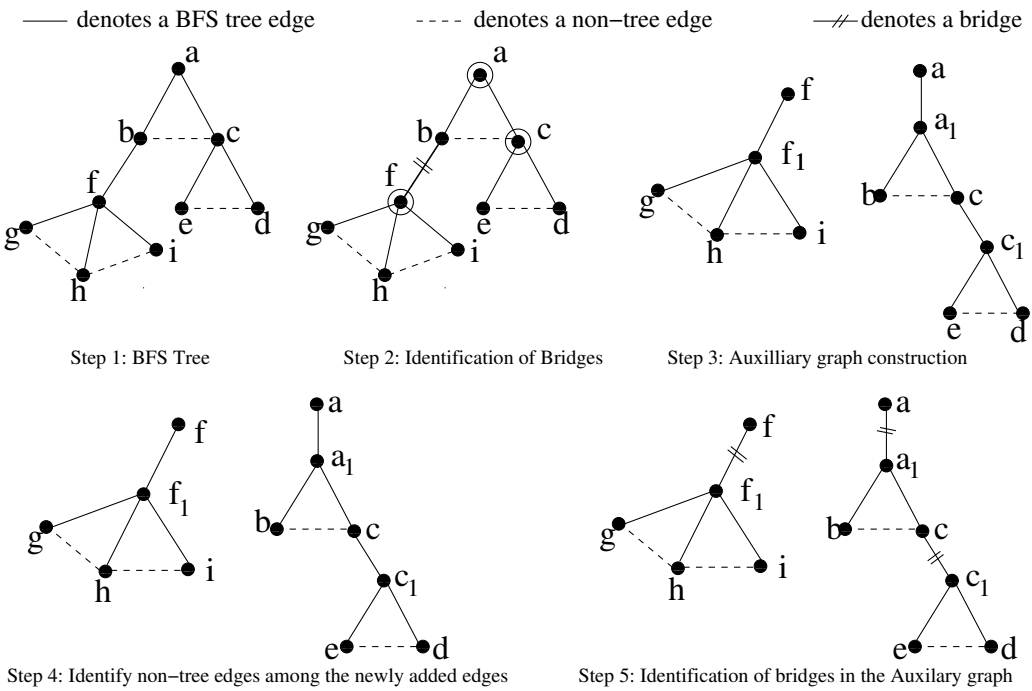


Figure 4.5: An example run of Algorithm LCA-BiCC. Graph G is BFS tree with non-tree edges (shown in step 1). Vertices marked in a circle in step 2 are the LCA vertices.

Algorithm 4 LCA-BiCC(G)

```
1: procedure LCA-BiCC(Graph G)
2:     T ← BFS(G)  /*Step I */
3:     {G_1, G_2, ···} = Bridges(G, T) /* Step II*/
4:     for all G_i i = 1, 2, ··· in parallel do
5:         Construct the auxiliary graph G'_i./*Step III*/
6:         Identify the non-tree edges in G'_i among the newly
           added edges to G_i from Step III/*Step IV*/
7:         {H_1, H_2, ···} = Bridges(G'_i, T'_i)/*Step V*/
8:         Check if r_i is an articulation point in G'_i
9:         Run a connected components algorithm on (G'_i \
   Bridges(G'_i, T'_i))  to identify the Biconnected Components of
   G_i/*Step VI*/
10:    end for
11: end procedure
```

The correctness of Algorithm 4 is immediate from Lemmata 3,4, and 5. An example run of the algorithm is presented in Figure 4.5. In the following, we elaborate on each step of Algorithm 4.

4.4.2 Step I: BFS on input graph G

We choose an arbitrary vertex r as the source vertex and perform BFS from r and also root the BFS tree at r. The output of BFS is stored in two arrays namely $L(v)$ and $P(v)$, where $L(v)$ signifies the level of the vertex in the BFS spanning tree and $P(v)$ stores the parent of v in the corresponding BFS tree. For the root r, we set $P(r) = -1$ and $L(r) = 0$.

4.4.3 Step II: Finding the Bridges of G

Recall that an edge in G is a bridge if and only if the edge is not on any cycle in G. The above property can be modified further to say that an edge is a bridge if it is not on any *fundamental cycle*, i.e., on cycles induced by non-tree edges according to a spanning tree. To

this end, we consider each non-tree edge e according to the BFS tree T from Step 1 and mark all edges in T that are in the cycle induced by e. Algorithm 5 explains the above steps. As shown in Algorithm 5, for each non-tree edge $e = xy$, we traverse up the tree edges from x and y till we reach the LCA of x and y. Each edge encountered in this process is marked. Edges of T that are not marked in the above process are the bridges of G. Also, end points of these bridges with degree at least two are articulation points in G.

For each bridge xy identified above with $x = P(y)$, we set $P(y) = -1$ that essentially decomposes G into its 2-edge-connected components. We return these 2-edge-connected components as the output of Algorithm Bridges.

Algorithm 5 BRIDGES(G)

1: **procedure** BRIDGES(*Graph* G, *Tree* T)
2: **for all** $e = (w,v) \in G \setminus T$ **do**
3: Mark the tree edges that we encounter in the process of computing the $LCA(w, v)$.
4: **end for**
5: **for all** $e \in E(G)$ **do**
6: If e is not marked then $B \leftarrow B \cup \{e\}$
7: **end for**
8: Return the connected components of $G - B$
9: **end procedure**

We choose to find the LCA of the end points of a non-tree edge by using a traversal from these end points while more robust algorithms exist for computing the LCA. In the parallel setting, such algorithms are studied by Soman et al. [144] as an application of range minima queries. Our choice is however justified by two reasons. Firstly, most real-world graphs have a low diameter as Table 4.1 shows. As the number of traversals is upper bounded by the diameter, such traversals do not pose a serious performance bottleneck. Secondly, using range minima query to compute the LCA points involves non-trivial steps that can be computationally intensive compared to simple

traversal.

4.4.4 Step III: Auxiliary Graph construction

Let $G_1, G_2, \cdots$, be the 2-edge-connected components of G. As described in Section 4.4, we create auxiliary graphs $G'_1, G'_2, \cdots$ corresponding to the 2-edge-connected components of G. (See also Figure 4.5 for an illustration.)

4.4.5 Step IV: Identify non-tree edges among the newly added edges

In this step, for each 2-edge-connected component G_i of G, $i \geq 1$, we do the following. We consider all the edges added to the auxiliary graph G'_i and mark them as either edges in the BFS tree or non-tree edges according to BFS. Notice that we do not have to run a BFS traversal again on G'_i, and can extend the BFS on G_i to identify the non-tree edges.

In particular, consider a vertex u which is the LCA of a non-tree edge e and vertices v and w are the children (neighbors) of u in the BFS tree T that are on the cycle induced by e. As part of the auxiliary graph construction, we add a vertex u', and edges uu', $u'v$, and $u'w$. We delete edges uv and uw. Such vertices v and w can now be end points of edges added during the construction of the auxiliary graph. Each such vertex v picks one of the alias vertices of its parent in the BFS of G'. The remaining edges between v and alias vertices will be marked as non-tree edges.

4.4.6 Step V: Identifying Articulation Points of G

In this phase we use the Algorithm 5 for identifying the bridges in each of the graphs $G'_1.G'_2, \cdots$. For each such bridge $e = xy$, notice that one of the end points is a vertex that is not in G and is added to the corresponding auxiliary graph during Step 3 of the algorithm. Such vertices are the articulation points of G.

4.4.7 Step VI: Finding the Biconnected Components of G

In this step, we utilize the information of bridges identified in Step V to identify the biconnected components of G. Let F' denote the forest of auxiliary graphs generated after the removal of bridges identified on auxiliary graph G'. We invoke connected components algorithm on F' and rename each alias vertex in F' to its original to get the biconnected components of G.

Analysis: We analyze the work done in Algorithm LCA-BiCC as follows. BFS requires $O(n + m)$ work, and finding the bridges in both original and auxiliary graph requires $O(d_T)$ per non-tree edge. where d_T is the depth of the BFS tree. So the total work done for all non-tree edges is $O(md_T)$. Rest of the steps such as constructing the auxiliary graph, finding the connected components (Step VI) all can be done in time $O(m + d_T)$ time. Thus, the algorithm in the worst case takes $O(md_T)$ time in a sequential setting.

However, as observed in Table 4.2, the average number of traversals towards the root of the tree required in identifying bridges is often much smaller than the depth of the BFS tree. This is the reason why our algorithm is a nearly linear (in m and n) in time.

4.5 Implementation Details

In this section, we describe the implementation details of our algorithm. We also justify our choices made during the implementation.

We use the compact adjacency list representation(CSR), to represent the graph in the memory. In this representation all the adjacency lists are packed together into a single large array. An array E_a is used to store the adjacency lists where the list for vertex i+1 immediately follows that of vertex i for all vertices in G. An array V_a, stores the starting indices of the corresponding adjacency lists in E_a. CSR representation helps in reducing the irregularity memory access. For a detailed description, an interested reader can refer to [17].

We perform the following program optimizations while implementing Algorithm LCA-BiCC. The performance of Algorithm LCA-BiCC is influenced by factors such as the depth of the BFS tree produced in Step I, the time taken to identify vertices in V_{lca} in Step II, and the size of the auxiliary graph constructed in Step III. We use the heuristic based approach of selecting the largest degree vertex as the source of the BFS to minimize the depth of the BFS spanning tree.

To minimize the time taken to identify the LCA vertices, we introduce the following optimization. Consider a non-tree edge, $e = (u, v)$ with both u and v performing a walk towards the root of the tree. If *both* u and v encounter a tree edge that is marked by another non-tree edge, say f, then it holds that $\text{LCA}(e) = \text{LCA}(f)$. Therefore, we stop the walks originating from u and v. These optimizations also reduce the size of the auxiliary graph.

We note that Lemmata 3, 4 and 5 can be modified suitably to work with any spanning tree. Using any spanning tree in Algorithm 5 requires one to associate level numbers to vertices in the tree. However, using a BFS tree allows to obtain the required level numbers as part of the BFS traversal and no additional computation is required for the same. Therefore, we chose to present the lemmata and the algorithms in terms of a BFS tree.

4.6 Experimental Results

4.6.1 Platform

We use an Intel i7 980x processor with 8 GB main memory as the experimental platform to test our results. The 980x is based on the Intel Westmere micro-architecture. This processor is from the Intel family with each core running at 3.4 GHz and with a thermal design power of 130 W. The i7-980X has six cores and with active SMT(hyper-threading) and can handle twelve logical threads. Beyond the memory, the i7-980x has a three level cache structure with an L1 cache per core of 64 KB split into two halves for instruction and data, an L2 cache per core of size 256 KB, and a shared L3 cache

of size 12 MB.

4.6.2 Datasets

We experiment on a variety of real-world and synthetic datasets. For simplicity purposes, directed edges were considered undirected. Multiple edges and self loops were removed. The input graphs are assumed to be a single connected component. If not they are made connected by adding explicit edges. We experiment with real world graphs from the dataset of University of Florida Sparse Matrix collection [4] and SNAP database [3]. These matrices can be converted to graphs naturally. A list of the instances we consider are given in Table 4.1. Synthetic graphs were generated with the GTGraph suite [12] using the default parameters.

Graph name	Source	$\|V\|$	$\|E\|$	Diameter
web-google	[89]	916,428	5,105,039	23
Webbase	[161]	1,000,005	3,105,536	27
Roadnet-PA	[89]	1,090,920	3,083,796	794
Roadnet-CA	[89]	1,971,281	5,533,214	863
web-Stanford	[93]	281,903	2,312,497	745
Wb-edu	[89]	9,845,725	57,156,537	511
amazon	[89]	262,111	1,234,877	29
Great-Britan	[89]	7,733,822	16,313,034	9340
asia-osm	[89]	11,950,757	25,423,206	48,126
Patents	[89]	3,774,768	16,518,948	29

Table 4.1: List of graphs that we use in our experiments.

4.6.3 Results on Real World Graphs

We study the results of our approach on the graphs mentioned in Table 4.1. We will demonstrate both the relative and absolute speedup compared to prior work on the same platform and analysis of our algorithm with respect to the graph computations involved in it.

Overall Improvement We consider the overall improvement obtained by our approach compared to the best known implementation for finding the biconnected components on the same platform. In Figure 4.6, the baseline we use for comparison is the *best* runtime achieved by either of Algorithm BFS-BiCC and Algorithm Color-BiCC reported in [94].

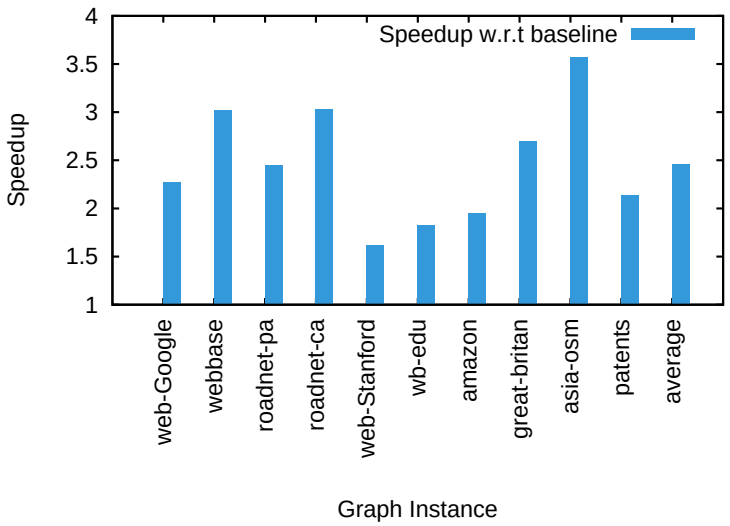


Figure 4.6: Comparing the overall performance improvement of our approach with respect to a baseline implementation. The Y-axis shows the ratio of the time taken by the baseline to our implementation.

As can be observed from Figure 4.6, Algorithm LCA-BiCC outperforms the baseline by a factor of 2.45x on average. As claimed in [94], the best of the above two algorithms is an improvement over the results of [44]. So, we expect that Algorithm LCA-BiCC too would outperform the results of [44].

Figure 4.7 shows the absolute times taken for all the three algorithms on the real world graphs. It is visible from Figure 4.7 that the run time of Algorithm BFS-BiCC and Algorithm Color-BiCC can

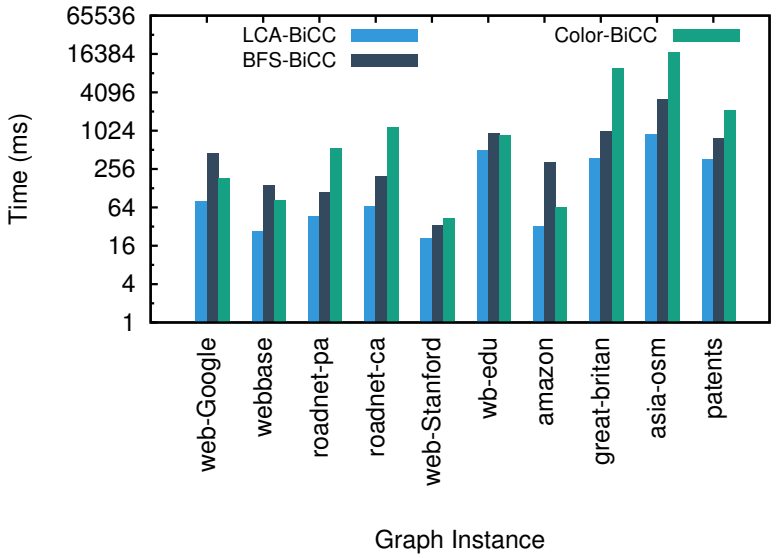


Figure 4.7: Figure shows the absolute runtime of Algorithm LCA-BiCC and Algorithms BFS-BiCC, Algorithms Color-BiCC from [94] (in left-to-right order of bars) on graphs listed in Table 4.1. Note that the Y-axis is in log-scale.

vary hugely across instances and it is not possible to determine a-priori which algorithm might perform better among the two as both are heavily dependent on the structure of the graph.

Understanding the Results One can analyze the improvement is in terms of the basic steps in each of the algorithms under comparison. Algorithm BFS-BiCC [94] does, in principle, n BFS computations. There are however several optimizations that Slota and Madduri [94] introduced to ensure that several of these BFS computations do not visit all the vertices. On the other hand, Algorithm LCA-BiCC performs only one BFS computation (Step 1) and one connected components computation (Step 6), apart from two calls

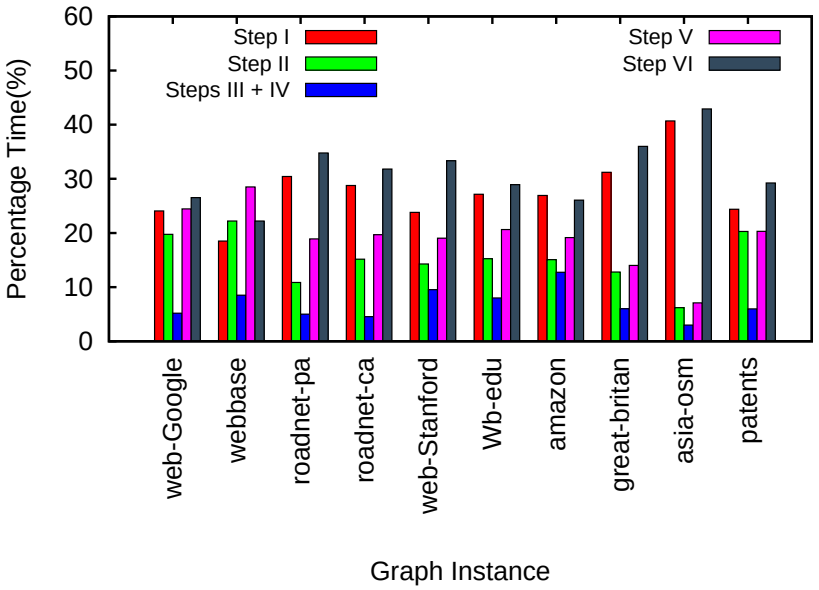


Figure 4.8: Profile of time taken across various steps of Algorithm LCA-BiCC.

to Algorithm Bridges (Steps 2 and 5) to find the bridges of a given graph. The rest of the computation is to construct the auxiliary graph G'. During Algorithm Bridges, the end points of each non-tree edge march up the tree using the parent pointers. As described before, the number of traversals for each end point of a non-tree edge is upper bounded by the depth of the BFS tree produced in Step 1. The depth of a BFS tree is also related to the diameter of the graph, and it is observed that most real-world graphs have a low diameter. The diameter of the graphs used in our experiments is also shown in Table 4.1.

Further, we also computed the average number of traversals needed by each end point to locate the LCA. The average number of traversals along with the depth of the BFS tree constructed in Step 1 are listed in Table 4.2 for the graphs from Table 4.1. It can be noticed that

the average number of traversals needed is smaller than the depth of the BFS tree and is under 10 in all the graphs. This small number of traversals on sparse graphs is what also helps in keeping the runtime of our algorithm low.

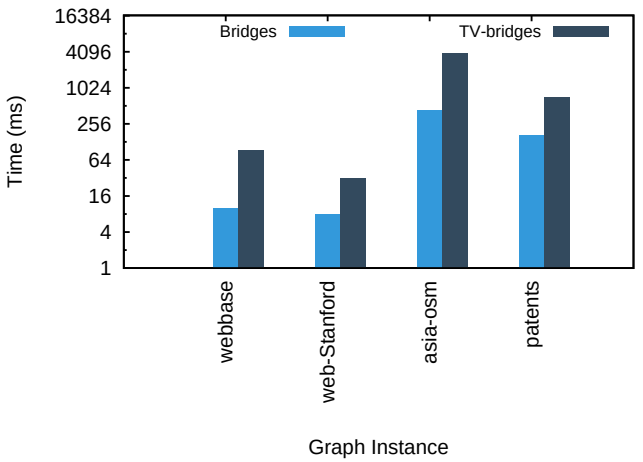


Figure 4.9: Figure shows the time taken to identify bridges in a graph G using our algorithm (Algorithm 5 and the algorithm of Tarjan and Vishkin [149]).

We study the size of the auxiliary graph constructed in Step III of Algorithm LCA-BiCC. Since an auxiliary graph is constructed for each 2-edge-connected component of G, we measure the sum of the sizes of the auxiliary graphs constructed with respect to each 2-edge-connected component of G. It is noted that the maximum increase in number of vertices at 25% and 20% occurred on the graphs roadnet-CA and roadnet-PA respectively. For all the other graphs from Table 4.1, the number of vertices increased by under 5%. For each vertex added to the auxiliary graph, there are two edges removed and three edges that are added to the auxiliary graph. Thus, the increase in the number of edges is equal to the number of newly created vertices. In terms of relative increase, the number of edges had a maximum

increase on the above instances again, at 8% and 7% respectively, and in all other instances from Table 4.1, the increase in the number of edges is under 3%. This shows that the actual increase in the size of the auxiliary graph is rather marginal and does not affect the performance of Algorithm LCA-BiCC in a significant manner.

Profiling and the Choice of the Source Vertex We finally show the time taken in each step of Algorithm LCA-BiCC as a percentage of overall time. Such a study shows the relative cost of each of the steps of the algorithm. The results of study are shown in Figure 4.8. Some of the improvement of Algorithm LCA-BiCC can be attributed to the fact that finding the bridges of a graph is a much easier task in a parallel setting. As shown in Figure 4.8, this step on the original graph G takes under 16% of the overall time on average, and takes under 20% on average on the auxiliary graph G'. These are labeled as Steps II and V respectively in Figure 4.8.

Note that Tarjan and Vishkin [149] also outline an approach to identify the bridges, their algorithm is usually more time consuming as it involves computation of functions low(v) and high(v) (as defined in Section4.2.2). Using the two functions low and high, a tree edge in T from $v \rightarrow w$ is marked as a bridge if and only if $w \leq low(w)$ and $high(w) \leq w + nd(w) - 1$ where $nd(w)$ refers to the number of descendants for a vertex w. To illustrate the simplicity of our approach for identifying bridges on sparse graphs, we consider some graphs from Table 4.1 and perform identification of bridges using our approach (Algorithm 5) and also the approach of Tarjan and Vishkin [149]. The time taken for both these approaches is plotted in Figure 4.9. As can be seen, our approach outperforms that of Tarjan and Vishkin.

In practice we have observed that unlike BFS-BiCC and Color-BiCC our LCA-BiCC approach is not heavily impacted by the choice of the source vertex while constructing the BFS spanning tree. The difference in the run time of Algorithm LCA-BiCC when a vertex of the highest degree is chosen as the source vertex versus an arbitrary source vertex is usually under 10% on the graphs from Table 4.1. On

the other hand, as reported in [94], and also as witnessed in our experiments too, Algorithm BFS-BiCC from [94] is significantly affected by the choice of the source vertex for all graphs listed in Table 4.1 and those considered in [94].

Graph name	%of LCA vertices	BFS depth	avg # LCA-traversals
web-google	3.31	12	1.09
Webbase	0.31	16	2.5
Roadnet-PA	15.02	534	9.91
Roadnet-CA	14.90	554	9.09
web-Stanford	1.90	727	2.82
Wb-edu	11.91	319	9.81
amazon	8.20	23	6.47
Great-Britan	2.96	6841	7.55
asia-osm	2.93	38,793	4.93
Patents	1.97	17	3.91

Table 4.2: List of graphs along with the average number of traversals made per non-tree edge. The column labeled "% of LCA vertices" indicates the number of vertices that are in V_{lca} as a percentage of the total number of vertices.

4.6.4 Results on synthetic graphs

Figure 4.10 shows the speedup obtained by Algorithm LCA-BiCC on synthetic graphs. These graphs were generated using GTGraph suite [12] with default parameters. In Figure 4.10, graph rand_pM_qM refers to a random graph generated with p million vertices and q million edges. We obtained an average speedup of 2.53x with respect to the best of BFS-BiCC and Color-BiCC [94].

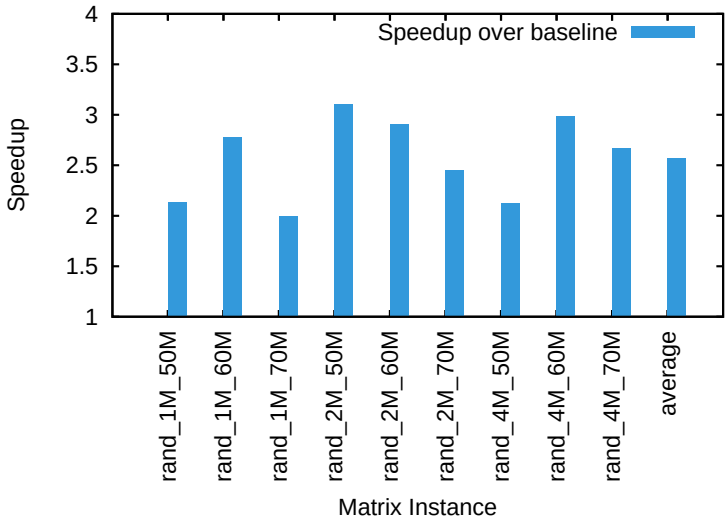


Figure 4.10: Comparing the overall performance improvement of our approach with respect to best of BFS-BiCC and Color-BiCC. The Y-axis shows the ratio of the time taken by the baseline to LCA-BiCC algorithm.

4.7 Further Improvements

In this section we present a simple yet efficient optimization for Algorithm BFS-BiCC [94] for identifying the biconnected components of graph G. Our improvements is based on two observations. Firstly, we make use of the observation from Section 7.2.1 that identifying the bridges in a graph is an efficient operation, especially in a parallel setting. Bridges also help in partitioning a graph into its 2-edge-connected components that can be processed independently. Secondly, given a 2-edge-connected graph, we use Lemma 3 to discard vertices that are certainly not articulation points and work with a small subset of potential articulation points. We call the vertices that are certainly not articulation points as *non-essential* vertices. Such a notation with respect to edges was also used by Cong and Bader [44].

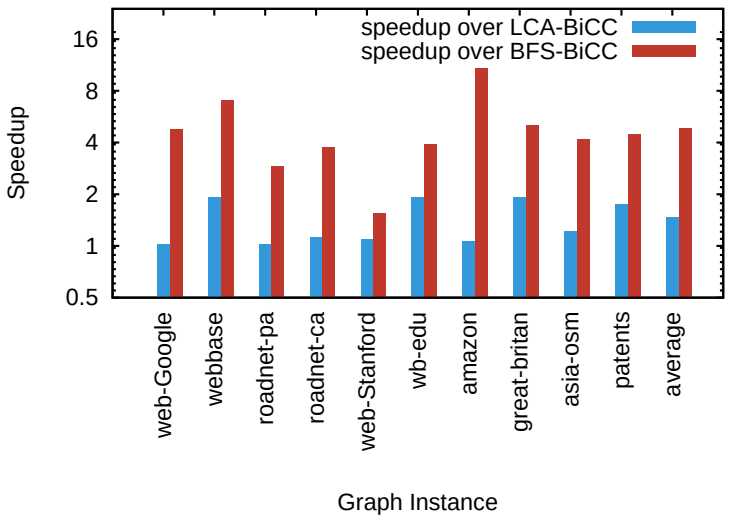


Figure 4.11: Speedup obtained by Algorithm LCA-BFS-BiCC over Algorithm LCA-BiCC and Algorithm BFS-BiCC ([94]), for the graphs from Table 4.1.

We use the above two observations to modify Algorithm BFS-BiCC from [94]. Unlike Algorithm BFS-BiCC, we do not examine each vertex for its possibility to be an articulation point. The BFS-BiCC approach is limited only to potential articulation points. Rest of the details are similar to that of Algorithm BFS-BiCC. As can be seen from Table 4.2, since the percentage of LCA vertices is small in most real-world graphs, we expect that our modifications to Algorithm BFS-BiCC would result in an improvement.

Our algorithm, called Algorithm LCA-BFS-BiCC, is shown as Algorithm 6. Some of the steps such as computing a BFS tree (Step I), finding the bridges of G (Step II) are identical to that of Algorithm LCA-BiCC. In Step III, we limit the call to Procedure BFS-L described by Madduri et al. [94, Algorithm 6] to only vertices in $V_{\text{lca}}(G_i)$ for each i. In Step III, once we identify the articulation

Algorithm 6 LCA-BFS-BiCC(G)

```
1: procedure LCA-BFS-BiCC(Graph G)
2:     T ← BFS(G)  /*Step I*/
3:     {G_1, G_2, ···, } = Bridges(G, T) /*Step II */
4:     for all G_i, i = 1, 2, ···, in parallel  do /*Step III*/
5:         for all v ∈ G_i do
6:             Articulation(v) = false; visited(v) = false
7:         end for
8:         for all  u ∈ V_lca(G_i) \ r_i  do /* r_i is the root of
9:                 the BFS tree of G_i */
10:            v ← P(u)
11:            if Articulation(v) = false  then
12:                l ← BFS-L(G, L, v, u, visited) /*BFS-L refers to
    [94, Algorithm 6]*/
13:                if l ≥ L(v) then
14:                    Articulation(v) ← true
15:                end if
16:            end if
17:        end for
18:        Check if r_i is an articulation point
19:        Identify the Biconnected Components of G_i using the Ar-
    ticulation Points of G_i /*Step IV*/
20:    end for
21: end procedure
```

points of each G_i, we use a step similar to that Step VI of Algorithm LCA-BiCC to identify the biconnected components of G.

4.7.1 Experimental Results

We use the experimental platform as described above in Section 4.6. We compare the results of LCA-BiCC algorithm with optimized BFS-BiCC approach on graphs given in Table 4.1.

In Figure 4.11, we show the improvement obtained by Algorithm LCA-BFS-BiCC compared to Algorithm LCA-BiCC. On average, Algorithm LCA-BFS-BiCC is about 1.46x faster compared to Algorithm LCA-BiCC. Figure 4.11 also shows the improvement obtained by Algorithm LCA-BFS-BiCC over the BFS-BiCC algorithm of [94]. Central to the improvement obtained by Algorithm LCA-BFS-BiCC is the idea that we have to perform BFS from only vertices in V_{lca}.

To understand the extent of improvement of Algorithm LCA-BFS-BiCC over Algorithm BFS-BiCC, we note the following. From the column labeled "% LCA Vertices" of Table 4.2, we see that in general, real-world graphs have a small percentage of vertex that are in V_{lca}. So, we expect significant performance gain for Algorithm LCA-BFS-BiCC. However, Figure 4.11 indicates the performance gain of Algorithm LCA-BFS-BiCC does not match the corresponding expected gain. For instance, if a graph has 10% of vertices in V_{lca}, one can expect a 10x improvement in the run time of Algorithm LCA-BFS-BiCC compared to that of Algorithm BFS-BiCC of [94]. This is not the case in general for the following reasons.

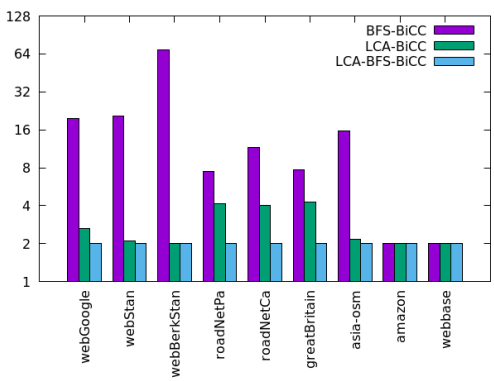
Algorithm BFS-BiCC introduces optimizations such as truncating the BFS-like traversals that are deemed unnecessary, invalidating the vertices of an already established biconnected component, and the like. Algorithm LCA-BFS-BiCC also benefits from these optimizations. However these optimizations mean that in Algorithm BFS-BiCC, even though most BFS traversals are terminated early on, there is still redundant work that is removed by using Algorithm LCA-BFS-BiCC. For experimental purposes, when the above mentioned optimizations from BFS-BiCC are removed, we do notice that

the speedup achieved by Algorithm LCA-BFS-BiCC compared to that of Algorithm BFS-BiCC is near the expected speedup.

Also in the BFS-BiCC approach, the choice of root vertex affects the runtime of the level-truncated BFS. Since the percentage of LCA-vertices (from Table 4.2) are quite low for real world graphs, it is observed during our empirical study that this choice would have minimum affect on the overall runtime of LCA-BFS-BiCC.

Results on GPUs Our algorithm GPU-LCA-BiCC is considerably faster than the BFS-BiCC version with speedups ranging from 5.18x to nearly 70x. The GPU implementation is at-least 2x faster than its CPU implementation. The removal of the 2^{nd} and the 3^{rd} pass and the efficient construction of the alias vertices results in faster GPU implementation.

BFS-BiCC does BFSs from every point in a BFS tree and checks whether the children node is accessible to node higher than the parent if the parent node is removed. If it is, then that node cannot be an articulation point. However Chaitanya[] proves that only $V_{lca}(G)$ is potential set of articulation points. Hence BFS-BiCC can be modified to include only the LCA points. We implemented this new version and tested it against our code. The final speedup across all three implementations is given below.



Relative Speedup of GPU-LCA-BiCC

It appears that LCA-BFS-BiCC is faster than BFS-BiCC only in certain cases. In most of the other cases, LCA-BFS-BiCC performs similar if not worse than BFS-BiCC. This speedup in certain cases can be attributed to two factors
First, in large sparse graphs, the number of non-tree edges are less and hence finding LCAs takes very less time. However as graphs become less sparse, finding LCAs itself becomes a complex task in terms of time. The advantage of working on a smaller subset of vertices is offset by actually finding the smaller subset of LCAs.
Second, it can be observed that even in cases of large sparse graphs, LCA-BFS-BiCC although always better, still gives higher speedups in only certain cases. The following table illustrates the point better.

Table 4.3: Diameter and Timings (ms)

Graph	Diameter	BFS-BiCC	LCA-BFS-BiCC
roadNet-pa	786	86.3	88.15
roadNet-tx	1054	168.69	151.81
netherlands-osm	2554	531.6	216.41
greatBritain	9340	2800	862
asia-osm	48126	380(s)	7.43(s)

One can see that BFS-BiCC slows quite considerably as the diameter increases. However no such slow-up occurs in GPU-BiCC or LCA-BiCC. BFS-BiCC as mentioned, does BFSs from every vertex till a higher up vertex is reached. In case of large diameter graphs, it often results in a single thread traversing upwards in a long chain of nodes. This causes other finished threads to wait and stalls the program. LCA-BFS-BiCC has no such traversal by a single thread upwards a long chain of nodes and is hence unaffected by diameter of the graph. Besides, finding LCA eliminates majority of vertices from consideration.

Despite these modifications, GPU-BiCC performs much better

than LCA-BiCC. GPU-BiCC was also tested for various starting points. The BFS was run from maximum degree vertex and some other random vertices. However no substantial change in timings was observed. GPU-BiCC performs consistently irrespective of starting point.

Chapter 5

Further Improvements in Graph k-Connectivity Testing on GPUs

Several recent sequential and parallel algorithms for k-connectivity rely on one or more breadth-first traversals of the input graph. It can be also noticed that the time spent by the algorithms on BFS operations is usually a significant portion of the overall runtime of the algorithm.

In this chapter, we study how one can, in the context of algorithms for graph connectivity, mitigate the practical inefficiency of BFS operations in parallel. Our technique suggests that such algorithms may not require a BFS of the input graph but actually can work with a sparse spanning subgraph of the input graph. The incorrectness introduced by not using a BFS spanning tree can then be offset by further post-processing steps on suitably defined small auxiliary graphs.

5.1 Introduction

In a remarkable result, Cheriyan and Thurimella [38] showed that the k-connectivity of an undirected graph can be tested by using a kn sized subgraph of the graph instead of using the entire graph. Formally, let T_i for $i \geq 1$ is the BFS spanning forest of $G \setminus \left(\cup_{j=1}^{i-1} T_j\right)$. Cheriyan and Thurimella show that the graph $\cup_{i=1}^{k} T_i$ is k-connected if and only if G is k-connected. The graph $H := \cup_{i=1}^{k} T_i$ is said to be a *certificate* for the k-connectivity of G. Similar results are also shown by Khuller and Scheiber [100].

The technique of Cheriyan and Thurimella [38] did improve the practical performance of parallel algorithms for testing the k-connectivity of an undirected graph. Evidence for this can be seen from the work of Bader and Cong [45], Chaitanya and Kothapalli [36], and that of Wadwekar and Kothapalli [153] for finding the biconnected components of a graph on symmetric multiprocessors, multi-core CPUs, and GPUs respectively. Much of this improvement can be attributed to the smaller size of the certificate in terms of the number of edges in the input graph.

However, in general, on large input graphs the time taken to obtain the certificate via parallel BFS operations can be a significant portion of the total run time. For instance, consider Algorithm N-GPU-BiCC from [153], which we rename as Algorithm Cert-GPU-BiCC in this chapter. This algorithm is so far the fastest known implementation for finding the biconnected components of a graph in parallel. Algorithm Cert-GPU-BICC performs two BFS traversals on the graph G to obtain a certificate of size at most $2n - 2$ edges for testing the biconnectivity of G. Figure 5.1 shows the time spent by Algorithm Cert-GPU-BiCC on BFS operations on a set of eight graphs. As shown in Figure 5.1, these two BFS operations consume on average 66% of the time spent by Algorithm Cert-GPU-BiCC. It indicates that to design faster parallel algorithms for graph k-connectivity, one must relook at the expensive BFS operations.

The inherent difficulty of efficiently performing a BFS traversal of

a graph led to several researchers identifying numerous algorithmic and data structure optimizations on various modern architectures. Some of these include the direction optimizing BFS by Beamer et al. [20], cache- and data structure optimizations by Chhugani et al. [39] fine-grained task management based approach on GPUs by Merrill et al. [114], and graph decomposition based methods by Buluç et al. [34]. Despite these advances, we notice in our study that BFS traversals still consume a significant portion of the run time of parallel graph connectivity algorithms.

The large time spent by BFS operations can be attributed to the fact that a BFS traversal requires assigning vertices to levels such that for $i \geq 0$, the shortest hop distance from the source of the BFS to any node in level i is i. Arriving at such an assignment in parallel requires expensive algorithmic/programming constructs such as synchronization, concurrent data structures, and work balancing among threads.

The above situation inspires us to design parallel graph k-connectivity algorithms that mitigate the inefficiencies of BFS operations. It must be noted that certificate based algorithms for graph k-connectivity are efficient only after obtaining the necessary certificate using k BFS traversals. Therefore, we suggest designing parallel algorithms that do not perform BFS operations on large graphs. One way of achieving this goal is to trade-off the cost of obtaining the certificate to its accuracy.

In this chapter, we show that by using novel strategies we can avoid performing BFS on the input graph G. Instead, we use a sparse spanning subgraph H' of the input graph. The subgraph H' thus identified is tested for its k-connectivity. It must be noted that as H' may not be an accurate certificate for the k-connectivity of G, the k-connectivity of H' may not provide an answer to the k-connectivity of G immediately. To make up for this inaccuracy, we include additional steps on an auxiliary graph F created out of G and the k-connectivity information obtained from H'. The auxiliary graph F is constructed such that G is k-connected if and only if F is k-connected. The sizes of H' and F are usually smaller compared to that of G resulting in a

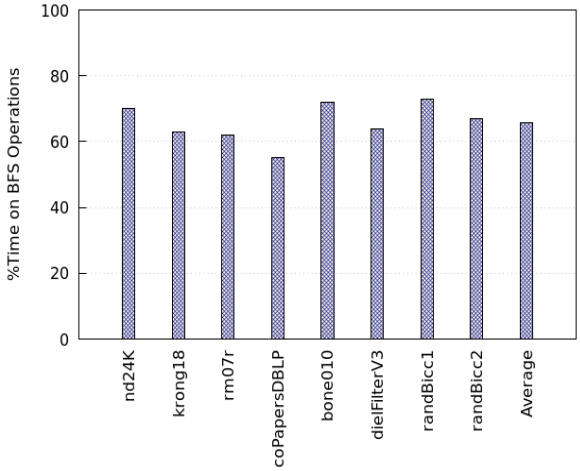


Figure 5.1: Figure shows the percentage time spent by Algorithm Cert-GPU-BiCC (cf. [153]) on BFS operations.

low overall run time.

We implement our approach for testing the 2, and 3-connectivity and obtaining the 1, 2, and 3-connected components of a graph on Nvidia GPUs. Our results are summarized in the following.

- For testing the connectivity, and obtaining the connected components of a graph our approach results in a speedup of 2.2x over [153] on a variety of real-world [43] and random graphs.
- For testing the 2-connectivity, and obtaining the biconnected components of a graph our approach results in a speedup of 2.2x over [153] on a variety of real-world [43] and random graphs.
- We provide the first known GPU based algorithms for testing 3-connectivity of a graph and finding the 3-connected components of a graph.
- For testing 3-connectivity, and obtaining the 3-connected components of a graph our approach results in a speedup of 2.1x

over a corresponding certificate-based approach implemented in this chapter.

Note that the certificate based approach of Cheriyan and Thurimella [38] involves three BFS traversals. So, one naturally expects a higher speedup for our approach on 3-connectivity. However that is not to be case as the graphs we use have a large number of 3-connected components resulting in more time spent in finding the 3-connectivity and the 3-connected components of F.

We believe that our technique has applications to other graph problems where one can algorithmically replace structures that are expensive to compute with simple to obtain and possibly inaccurate structures followed by a post-processing step. Our work therefore opens the possibility of reinterpreting important steps in parallel graph algorithms so as to make them more efficient in practice.

5.2 An Overview of Our Approach

Several recent studies on parallel graph algorithms have explored varied techniques to improve their practical efficiency on multi-core and accelerator based architectures. Many such studies use well-known graph computations such as traversals, spanning trees, edge/vertex decompositions as a subroutine. These algorithms can be summarized as follows. From the input graph G, one obtains a structural subgraph H such that computation on G can be translated or reduced to computations on H followed by additional post-processing steps as required. An example of this can be seen in the work of [58, 119] where a reduced graph that shrinks all vertices of degree two is used as the graph H.

The above mentioned approach of computing on a subgraph H often helps if H is of a smaller size than G. The benefits of the above stated approach however will be limited if identifying H is expensive possibly due to strict structural guarantees required on H. As can be noticed from Figure 5.1, a large portion of time spent in obtaining H indicates scope for revisiting the approach.

In this direction, we propose to reconsider the algorithmic implication of replacing H with a suitable, easy to create structure H' such that the computation can be done on H' instead of H. In case the result of the computation on H' does not provide a correct result for the required computation on G, additional steps may be required depending on the nature of the problem. However, in these additional post-processing steps, the size of the problem is often much smaller than the size of the original graph. Therefore, it is expected that the cost of post-processing is small.

The approach can be seen to have three stages. In Stage I, we obtain a subgraph H' of G. Stage II performs computation on H'. An optional Stage III introduces a post-processing step, if required. The technique as presented above allows for multiple possibilities at all stages. In Stage I, H' can be obtained by (i) uniformly sampling the input graph G, (ii) by relaxing the structural properties required of H, (iii) using importance sampling, and the like. In Stage II, the computation on H' is chosen based on the input problem. Depending on the choices exercised in Stage I and Stage II, we consider the question of whether the output of Stage II can lead to the required output on the original graph. If the output of Stage II is insufficient to arrive at the final answer, we consider Stage III as the post-processing stage. In Stage III also, the computation required depends on the nature of the problem and the utility of H'. Stage III, depending on the problem can use possibilities such as iterating, and augmenting the result, and constructing an auxiliary graph for suitable computation.

We note that as H' and H are expected to be of similar size, the time taken for computing on H and H' will not differ significantly. Hence, for our technique to be useful in practice, the cost of Stages I and III should be lesser than the cost of obtaining H from G.

Figure 5.2 illustrates the idea of our approach. In Figure 5.2, we also list some of the possibilities at each stage of the approach. The particular choice used in this chapter is shown in bold text in Figure 5.2.

In this chapter, we apply our approach to test the k-connectivity and find the k-connected components of an undirected graph G for

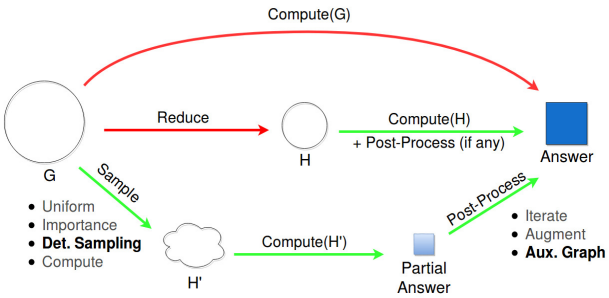


Figure 5.2: Figure that illustrates our technique in comparison to other approaches towards practical parallel graph algorithms. The top path (colored red) represents direct computation that is usually expensive. The middle path indicates preprocessing via strict structural subgraphs or constructs that are sometimes expensive to create. The bottom path (colored green) corresponds to the less expensive approach proposed in this chapter.

$k = 1$, $k = 2$, and $k = 3$. Cheriyan and Thurimella [38] show that a subgraph H can be constructed as a certificate for G via k BFS traversals. (The k-connectivity of H offers a quick way to test the k-connectivity of G.) As obtaining H via multiple BFS traversals of the input graph can consume a significant portion of the overall runtime (cf. Figure 5.1), we show that our approach can be helpful in arriving at faster parallel algorithms for graph k-connectivity.

5.3 Application to 1-Connectivity

Sutton et al. [146] present modifications to the standard PRAM algorithm of Shiloach and Vishkin [138] to make the algorithm more suitable on modern parallel architectures. Their modifications are aimed at addressing the irregular memory accesses of typical PRAM style graph algorithms. While the algorithm of Sutton et al. also

uses the basic idea of the algorithm from [138], the main difference is in avoiding iterating over the same edge several times and using atomic operations (such as compare_and_swap) to control concurrent operations.

The main subroutines in the algorithm of Sutton et al. [146] are Link and Compress. The subroutine Link is called *once* for every edge where it is ensured that for every edge uv of the graph, u and v remain in the same tree. The subroutine Compress is similar to the pointer-jumping routine that is popular in parallel algorithms [90, 164].

Algorithm 7 Procedures Link and Compress from [146].

```
1: procedure LINK((e = uv, π))
2:     π_1 = π(u), π_2 = π(v)
3:     while (π_1 ≠ π_2) do
4:         h = max{π_1, π_2}, ℓ = min{π_1, π_2}
5:         if Compare_and_Swap(π(h), h, l) then return
6:         end if
7:         π_1 = π(π(h)), π_2 = π(ℓ)
8:     end while
9: end procedure
10: procedure COMPRESS((v, π))
11:     while (π(π(v)) ≠ π(v)) do
12:         π(v) = π(π(v))
13:     end while
14: end procedure
```

Unlike the Shiloach-Vishkin algorithm, the algorithm by Sutton et al. [146] is sensitive to the order in which edges are processed through subroutines Link and Compress. In fact, one can create worst-case example graphs and an adversarial order of processing of edges so that the Subroutine Link requires $O(n)$ work for some edge uv and the Subroutine Compress requires a total of $O(n^2)$ work. This is in contrast to the upper bound of $O(m + n)$ on the work performed by the Shiloach-Vishkin algorithm [138].

In the case of 1-connectiivity, Sutton et al. [146] use a subgraph

sampling based approach where a subgraph H of G is formed with n vertices and at most $O(n)$ edges. This subgraph H can be identified via several sampling methods: choosing uniformly at random, choose edges having vertices of low degree as an end point, choosing vertices uniformly at random along with all the neighbors of chosen vertices, and the like.

Sutton et al. through their experiments show that sampling via vertices and their neighbors is most inefficient of the above three choices. Between sampling edges uniformly at random and sampling edges based on the degree of the end points of the edges, the latter is shown to perform better. This behavior is true even when one considers how the largest connected component is completely identified as a fraction of the processed edges.

5.4 Application to 2-Connectivity

Recall from graph theory that a graph G is said to be biconnected, or 2-connected, if every pair of vertices $v, w \in V(G)$ have at least two vertex disjoint paths between them. The maximal 2-connected subgraphs of G are called as the biconnected components of G. A vertex v of G is called an *articulation point* or a *cut point* if the removal of v from G disconnects G. An edge vw of G is called a *bridge* or a *cut edge* if removing vw from G disconnects G. Finding whether G is biconnected and the biconnected components of G has applications in networks [27,154], clustering [33], planarity testing [79] and data visualization [5].

On GPUs, the only known algorithm for this problem is presented recently in [153]. Algorithm GPU-BiCC from [153] argues that in a parallel setting, finding the bridges of a graph G is much easier compared to finding the articulation points. Based on this observation, the algorithm first identifies the bridges of G and separates G into its 2-edge-connected components. (The 2-edge-connected components of G are its maximal subgraphs such that every pair of vertices in each subgraph have at least two edge disjoint paths between them). To

identify the articulation points in each 2-edge-connected component G_i of G, Algorithm GPU-BiCC builds an auxiliary graph G'_i such that bridges of G'_i can be used to locate the articulation points of G_i, and hence those of G. This information is then used to subsequently identify the biconnected components of G. For further details, we refer the reader to [153]. Algorithm GPU-BiCC is 4x faster compared to other parallel approaches [36,142]. On dense graphs, Algorithm Cert-GPU-BiCC from [153] uses the certificate as defined by Cheriyan and Thurimella [38] to obtain a further 2x speedup on Algorithm GPU-BiCC.

5.4.1 Our Approach

As mentioned in the previous section, one can take H as the subgraph formed by taking the union of a BFS tree T of G and the BFS spanning tree of $G \setminus T$. This certificate H will have n vertices and at most $2(n-1)$ edges. However, as Figure 5.1 shows, obtaining H is an expensive step, taking an average of 66% of the total time of Algorithm Cert-GPU-BiCC. We therefore use our approach as outlined in Section 5.2 by replacing H with a suitable H'.

To this end, we start with H' as a kn sized spanning subgraph of G for an appropriately chosen constant k and proceed to find the biconnected components of H'. As H' may miss including certain edges critical to answer the biconnectivity of G, H' is not a certificate for biconnectivity of G. Nevertheless, the biconnected components of H' can be used to create an auxiliary graph F. Each vertex in F roughly corresponds to a biconnected component of H' and edges of F represent edges between these components. The edges of $G \setminus H'$ are used to add additional edges to F so that F acts as a valid certificate for the biconnectivity of G. As H' is of comparable size to H and the size of F is expected to be small, our approach can help reduce the time spent in BFS operations. More formal details of our approach are presented in the following.

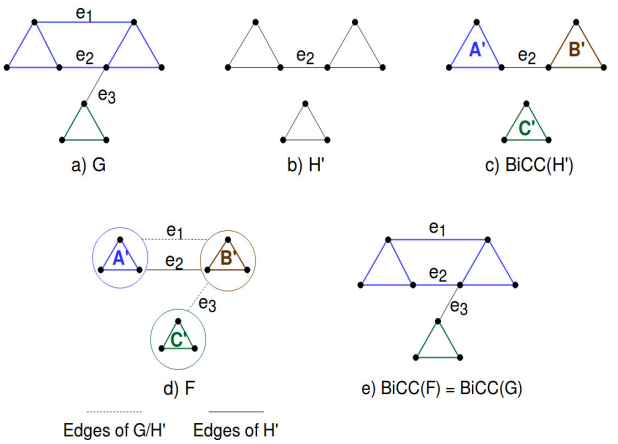


Figure 5.3: An example run of Algorithm Sample-GPU-BiCC on the graph in part (a) of the figure.

Our Algorithm

Algorithm Sample-GPU-BiCC for finding the biconnected components (BCCs) of a connected graph G is listed as Algorithm 8. An example of Algorithm 8 is shown in Figure 5.3. Each of the steps of the algorithm are explained below.

Algorithm 8 Algorithm Sample-GPU-BiCC

1: Input: A connected graph G
2: Output: The Biconnected Components (BCCs) of G.
3: Obtain a subgraph H' from G
4: Find the BCCs of H' using Algorithm GPU-BiCC.
5: Extract F using the BCCs of H' and edges of G
6: Find the BCCs of F using Algorithm GPU-BiCC.

- Step 1 – Obtain subgraph H' from G: Recall that H' is a kn sized subgraph of G. We identify H' by viewing the edges of G as an edge list and including every m/kn^{th} edge for a total

of kn edges. Note that no randomness is used as any kn edges suffice for our purpose.

- Step 2 – Find BCCs of H': Once H' is obtained, we find the BCCs of H' using Algorithm GPU-BiCC ([Algorithm 6] from 4). These BCCs are used to define the vertices of F.

- Step 3 – Extract F using BCCs of H' and the edges of G: In this step, we create an auxiliary graph F. The BCCs of H are treated as super-vertices that correspond to the vertices of F. Recall that a vertex can be part of several BCCs. In particular, articulation points belong to multiple BCCs. Hence we keep such vertices as a separate vertex in F. Two vertices in F are joined by an edge if there exists an edge $vw \in E(G)$ such that v and w are in different super-vertices of F.

  This results in F being a multi-graph. In such cases, since we need to know if there are at least two edges between nodes in F, we need to add at most two edges between any two vertices of F. For every pair of vertices v, w in F with two edges between them, we keep only one such edge between v and w, and add an auxiliary vertex v' and edges $vv', v'w$ to F. By doing so, F is now a simple graph.

  We note that as the edges of G are all used to define the edges of F, F acts as a certificate for the 2-connectivity of G. In other words, if vertices x and y have more than one vertex-disjoint path between them in G, then either x and y belong to the same super-vertex of F or the super-vertices of F that contain x and y have more than one vertex-disjoint path between them. The former happens when all the edges on at least one cycle containing x and y is in H'. The latter happens when no cycle containing x and y is in H' in which case, the edges of the cycle(s) that are not in H' induce edges in F that ensure that the super-vertices of F containing x and y have at least two vertex-disjoint paths.

- Step 4 – Find BCCs of F: In this step, we find the BCCs of F using Algorithm GPU-BiCC [153]. The biconnected components identified in this step can be used to identify the biconnected components of G.

5.4.2 Implementation Details

We implement Algorithm Sample-GPU-BiCC on a GPU. For BFS on a GPU, we use the implementation from [114] that uses a fine-grained task management strategy. According to our approach, these BFS operations are done on subgraphs H' and F thereby requiring lesser time. This is followed by identifying the Least Common Ancestor (LCA) of the end points of every non-tree edge. Here, we launch one thread for every non-tree edge. Generating F requires a lookup of the edge list of G along with the information of BCCs of H'. This is easily implemented on a GPU by assigning a thread to every edge of G. We therefore note that Algorithm Sample-GPU-BiCC is amenable to a GPU-based execution where a massive thread pool is supported. We arrange the threads into blocks with 1024 threads per block.

5.4.3 Experimental Results and Discussion

Experimental Platform

The experimental platofrm we use in our experiments in the K40c GPU, a summary of which is presented in Chapter 2.

Dataset

The graphs we use for our experiments are taken from real-world datasets [43], random graphs following the Erdős-Rényi model [24] generated using the GTGraph generator [12]. All the graphs we consider are undirected and unweighted. Directed graphs are made undirected by removing the direction on the edge. Graphs that are not connected are augmented with additional edges to make them connected. Key properties of the graphs are shown in Table 5.1.

Table 5.1: Graphs used in our experiments. In the table, the letter K (*resp. M*) stands for a thousand (million).

Graph	Description	Nodes	Edges
Real-World Graphs [43]			
nd24k	3D Mesh, ND set.	72K	14.3M
kron18	kronecker, DIMACS10	262K	10.5M
rm07r	3D viscous case	381K	37.4M
coPaperDBLP	coauthor citation network	540K	15.2M
bone010	3D trabecular bone	986K	36.3M
dielFilterV3	High-order vector finite element method in EM	1.1M	45.2M
Random Graphs			
rand-Bicc1	1 BCC	1M	75M
rand-Bicc2	10000 BCCs	1M	75M

Results

In this section we compare our implementation of Algorithm Sample-GPU-BiCC to that of Algorithm N-GPU-BiCC [153]. The overall improvement in performance on the graph listed in Table 5.1 is shown in Figure 5.4. Algorithm Sample-GPU-BiCC achieves a speedup ranging from 1.47x to 3.35x compared to Algorithm Cert-GPU-BICC [153]. The average speedup as shown in Figure 5.4 is 2.2x. All the above experiments were run with $k = 4$. The time spent by our approach on BFS operations is listed on the top of each bar. As can be noted, this time is on the average only 15% of the total time indicating that our approach is successful in mitigating the practical inefficiency of BFS operations in the context of parallel graph biconnectivity algorithms. The graph nd24K has a high speedup of 3.35x

as it is dense and is biconnected. Even a small sample of edges keeps almost all the nodes in a single BCC. For the graph coPaperDBLP, the lower than average speed-up can be attributed to its graph structure and the sampling strategy used. In this case, H' has several long chains of vertices of degree two. This increases the BFS time and the subsequent time for finding the BCCs of H'.

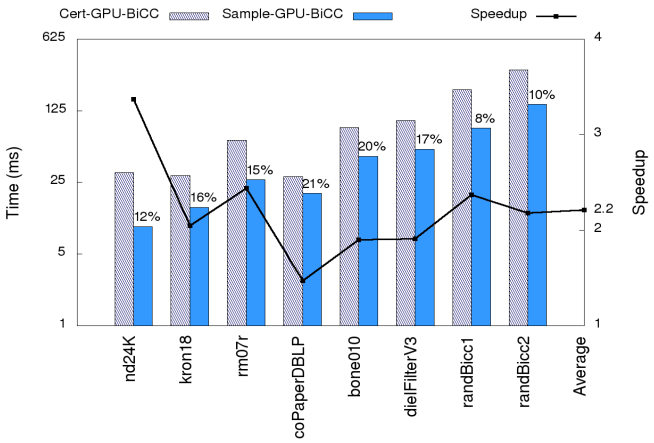


Figure 5.4: Figure showing the time taken by Algorithms Cert-GPU-BiCC and Sample-GPU-BiCC on the graphs listed in Table 5.1. Numbers on the right bars indicate the percentage of time spent by Algorithm Sample-GPU-BiCC in BFS operations. The Secondary Y-axis gives the speedup of Algorithm Sample-GPU-BiCC over Algorithm Cert-GPU-BiCC.

To study the impact of the choice of k on the performance of Algorithm Sample-GPU-BiCC, we plot time taken by our algorithm on two graphs from Table 5.1 as we vary k. The results of this experiment are shown in Figure 5.5 and 5.6 for graphs kron18 and coPaperDBLP respectively. When k is small, the size of H' is small. As a result, the time taken in Step 2 is small. However, if only few edges of G are included in H', the number of biconnected components found in Step 2 tends to be high. Therefore, the size of F grows, resulting in

Step 4 consuming more time. On the other hand if the value of k is high, the size of H' is high. As a result, the time taken in Step 2 is high. But, since more edges have been included in H', the size of F decreases thereby making Step 4 relatively faster. Steps 1 and 3 are not significantly impacted by the choice of k and hence omitted from Figures 5.5 and 5.6. In Figures 5.5 and 5.6, the vertical line at $k = 4$ indicates that the minimum for total time is achieved at $k = 4$.

Another factor to be noted is that the size of F, shown in Figures 5.5 and 5.6, indeed decreases as we increase k. This is in tune with our original motivation that doing a BFS on such a small graph will be significantly better than doing a BFS on G. The BFSs on smaller-sized H' and F combined are cheaper than a BFS on G. (Obtaining a certificate from G requires 2 BFSs on G, not one). The above discussion suggests that k should be chosen appropriately. We see from Figures 5.5 and 5.6 that a good value of k is around 4 while values of k between 3 to 5 offer a similar result in general.

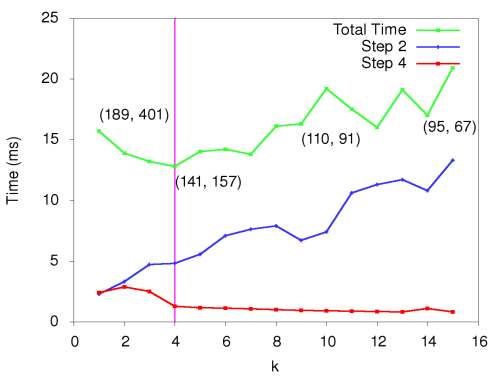


Figure 5.5: Figure represents the time taken by Algorithm Sample-GPU-BiCC on the graph kron as k is varied. Tuples on the line labeled "Total Time" show the number of vertices and edges of F in thousands at $k = 1, 4, 9$, and 14.

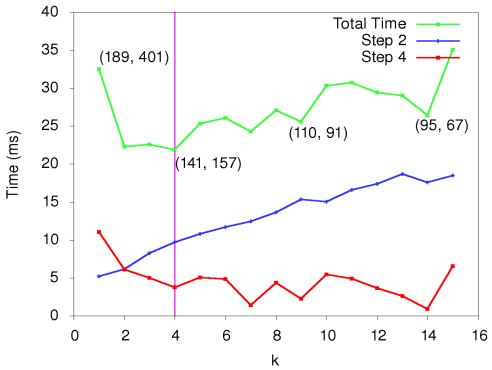


Figure 5.6: Figure represents the time taken by Algorithm Sample-GPU-BiCC on the graph coPaperDBLP as k is varied. Tuples on the line labeled "Total Time" show the number of vertices and edges of F in thousands at $k = 1, 4, 9$, and 14.

Discussion

In this section, we summarize a few important points concerning our approach.

- Obtaining H': We observe that H' can be generated in several other ways such as a uniformly-at-random process on the edges, a selection based on degree of the vertices, and other such strategies. We found that using randomness to obtain H' is not necessary to make our approach work. With a deterministic post-processing phase, we believe that one should focus more on trying to reduce the overall run time instead of getting a "good" H'. In all the approaches, the impact on the performance was almost similar. Hence we use deterministic sampling.

- Certificate based approaches: From the work of Bader and Cong [45] and also that of Cheriyan and Thurimella [38], it is apparent that using a certificate for testing the biconnectivity of a graph is practically efficient. In our approach, as the graphs

H' and F are very sparse, such a certificate is not required and Algorithm GPU-BiCC is enough.

5.5 Application to 3-Connectivity

5.5.1 Overview

Recall that a graph is said to be triconnected, if every pair of vertices $v, w \in V(G)$ have at least three vertex disjoint paths between them. The maximal 3-connected subgraphs of G are called as the triconnected components of G. A *separating pair* in a graph G is a pair of vertices v, w such that removing v and w from G disconnects G. Graph triconnectivity has applications in networks [73] and in determining isomorphism in planar graphs [86].

Hopcroft and Tarjan [87] presented the first sequential algorithm for finding the triconnected components of a graph. The algorithm from [87] is based on the depth-first traversal (DFS) of a graph. Given that DFS is a P-complete problem [90], this approach would not be parallelizable in the PRAM sense. Over the years, few PRAM style parallel algorithms have been presented for finding the triconnected components of a graph. Ramachandran and Vishkin [128] present a PRAM algorithm for testing triconnectivity that runs in $O(\log n)$ time using $O(n + m)$ work. Miller and Ramachandran [116] extend the work from [128] to also obtain the triconnected components of a graph using $O(\log^2 n)$ time and $O(n + m)$ work on a CRCW PRAM. Their algorithms make use of the ear decomposition of a graph and define an auxiliary graph for every ear of G. These auxiliary graphs are then used to check the 3-connectivity of G followed by finding the 3-connected components of G. To date, the algorithm of Miller and Ramachandran is the fastest PRAM algorithm for identifying the triconnected components of a graph in parallel. These algorithms [116, 128] can be recast to use the result of Cheriyan and Thurimella [38]. Vishkin and Edwards [59, 60] study parallel implementations of 2- and 3-connectivity algorithms on the XMT architecture [152] and compare how these XMT implementations scale with increasing

number of cores.

In this chapter, we implement the algorithm of Miller and Ramachandran [116] on a GPU and also extend our approach from Section 5.2 for the graph triconnectivity problem. We start by briefly describing the algorithm of Miller and Ramachandran [116].

5.5.2 The Algorithm of Miller and Ramachandran for Graph Triconnectivity

The algorithm of Miller and Ramachandran [116] is based on an open ear decomposition of a graph. An open ear decomposition of a graph $G(V, E)$ is a partition of E into ordered edge-disjoint paths $P_0, P_1, P_2, ...$ such that P_0 is a simple cycle and every other path P_i, $i \geq 1$, has its endpoints on previous paths (ears) and no internal vertices of P_i lie on P_j, for $j < i$. Since a vertex cannot be internal to two ears, an open ear decomposition provides scope for traversing the graph in parallel. In addition, Miller and Ramachandran [116] prove that the two vertices from any separating pair in a biconnected graph are non-adjacent vertices of some ear P_i.

As a result, Miller and Ramachandran start with an open ear decomposition of a biconnected graph. The algorithm then generates the bridges for every non-trivial ear in parallel. (An ear is non-trivial if it has at least three vertices.) For a given subgraph S, the bridges with respect to S is a partition $V(G \setminus S)$ such that two vertices are in the same class if and only if there is path connecting them without using any vertex of S. For the graph in Figure 5.7(a) the bridge graph of ear P_1 is shown in Figure 5.7(b). Each such bridge is then compressed into a single vertex as indicated by vertices B_1 through B_5 in Figure 5.7(c). This single vertex is connected to the original ear through the same attachments as the corresponding bridge. This step is shown in Figure 5.7(c). This is done for every bridge for every non-trivial ear in parallel. The bridge graph is simplified into an ear graph by merging bridges which share the same attachments on an ear as shown in Figure 5.7(d).

The ear graph for every ear is further simplified by merging inter-

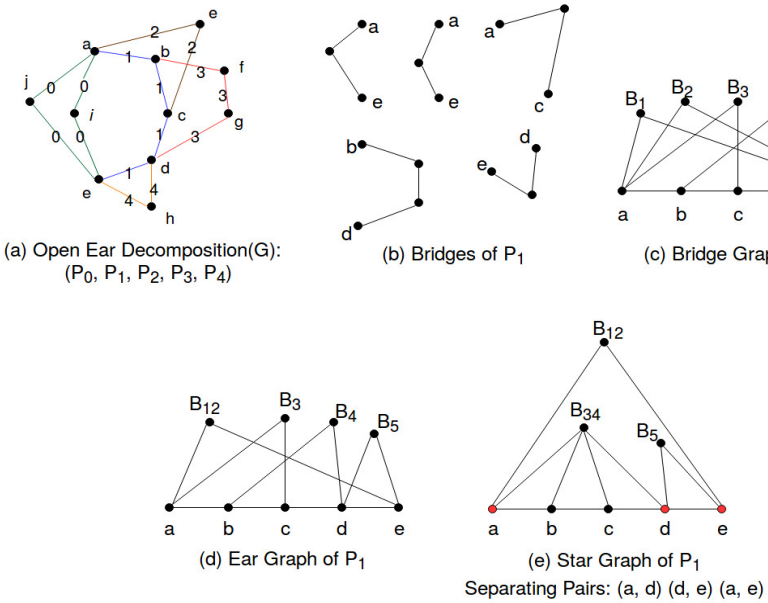


Figure 5.7: Figure showing the stages in the algorithm of Miller and Ramachandran [116]. The numbers on edges in part (a) of the figure show the ear that the edge belongs to.

lacing bridges into an non-overlapping graph, called a star graph. All separating pairs can be easily discovered through a star graph. Figure 5.7(e) shows the formation of a star graph from the corresponding ear graph and the subsequent separating pairs with respect to a single ear. This process is done across the graph for all ears in parallel. Once the separating pairs are identified, the triconnected components are generated by splitting the graph into Tutte splits [150] for every separating pair. The entire algorithm is shown in run in $O\log^2 n)$ time using $O(n+m)$ work in the CRCW PRAM model. We refer the reader to [116] for further details.

5.5.3 Triconnectivity on GPU

To the best of our knowledge, there is no known GPU based algorithm for the graph triconnectivity problem. In this section, we provide a GPU based implementation for the algorithm of Miller and Ramachandran [116]. A brief summary of our implementation is given below. Henceforth, we refer to our GPU implementation for triconnectivity as Algorithm GPU-TriCC listed as Algorithm 9.

Algorithm 9 Algorithm GPU-TriCC

1: Input: Biconnected graph G
2: Output: Triconnected components of G
3: Find an open ear decomposition of G
4: **for all** every nontrivial ear P_i **do**
5: Construct the bridge graph from bridges
6: Obtain the ear graph $G_i(P_i)$ from the bridge graph
7: Coalesce the interlaced ear graph into a star graph $G_i^*(P_i)$
8: Identify the separating pairs from $G_i^*(P_i)$
9: **end for**
10: Use Tutte splits to obtain the triconnected components

We employ Ramachandran's [127] popular ear decomposition algorithm for generating the open ear decomposition. Our ear decomposition requires two graph traversals and a sorting of edge list. Obtaining bridges and the subsequent bridge graph requires a connected components algorithm. We use Soman et al. [145] GPU implementation for the same. The ear graphs are generated through a divide and conquer approach as mentioned in [116]. Assuming r ears, the first step in the divide and conquer approach generates the ear graph for the first and the last $r/2$ ears. Connected components of the i^{th} stage are utilized at the $(i+1)^{th}$ stage as we narrow down to generating the ear graph for every individual ear. Every ear graph is then coalesced in parallel to generate the star graph. Coalescing involves resolving all overlapping attachments in the ear graph. Separating pairs can be easily identified once the star graph is generated as shown in Figure

5.7(e). The graph is then split into upper split and lower split graphs for every separating pair (a, b) on the ear P_i. The upper split and lower split graphs are a division of vertices belonging to ears $P_j, j < i$ and ears $P_k, k > i$. Miller and Ramachandran [116] prove that each of the split is biconnected and every other separating pair lies either in the upper split graph or in the lower split graph but not in both. Hence this procedure is applied recursively till no separating pair is present in either of the split graphs generated. Thus the triconnected components of G are identified.

Notice from the algorithm of [116] that the bulk of the work done can be associated with each ear subsequent to obtaining an open ear decomposition. As every graph G has $m - n + 1$ ears, the number of edges in G heavily impacts the performance. Hence a reduction in the size of the graph through use of certificates provides scope for a better performance. To this end, we make use of the idea of Cheriyan and Thurimella [38]. Accordingly, a certificate for triconnectivity of G is obtained as the union of T, $F_1 = BFSSpanningForest(G/T)$ and $F_2 = BFSSpanningForest(G/(T \cup F_1))$, where T is a BFS tree of G. The graph $H := T \cup F_1 \cup F_2$ is then provided as the input to Algorithm GPU-TriCC. This modification is named as Algorithm Cert-GPU-TriCC and is listed as Algorithm 10.

Algorithm 10 Algorithm Cert-GPU-TriCC.

1: Input: Biconnected graph G
2: Output: Triconnected Components of G
3: $T := BFS(G)$
4: $F_1 :=$ BFSSpanningForest(G/T)
5: $F_2 :=$ BFSSpanningForest$(G/(T \cup F_1))$
6: $H := T \cup F_1 \cup F_2$
7: Run GPU-TriCC on H

Results

On a collection of real-world graphs listed in Table 5.1 along with the random graphs of Table 5.1, we study the performance of Algorithms GPU-TriCC and Cert-GPU-TriCC. The random graphs are generated to have a particular number of TCCs as shown in Table 5.1. The GPU used for these experiments is an Nvidia K40c GPU (cf. Section 5.4.3). From Figure 5.8, we can observe that Algorithm Cert-GPU-TriCC is on an average 5x faster compared to Algorithm GPU-TriCC.

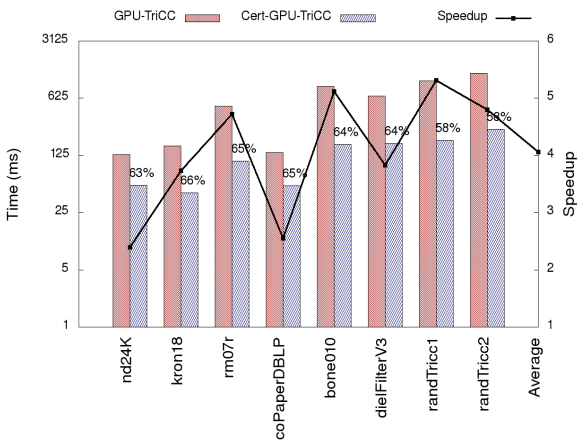


Figure 5.8: Figure showing the time taken by Algorithms GPU-TriCC and Cert-GPU-TriCC on the graphs listed in Table 5.2. Numbers on right bars indicates the percentage of time spent by Algorithm Cert-GPU-TriCC in BFS operations.

In Figure 5.8, we show the percentage of the time spent by Algorithm Cert-GPU-TriCC in obtaining the certificate H using three BFS traversals of G. As can be observed from Figure 5.8, Algorithm Cert-GPU-TriCC despite being 5x faster than Algorithm GPU-TriCC, spends nearly 63% of total time in obtaining H. The high cost of procuring the certificate leads us to try our approach from Section 5.2 for this problem.

5.5.4 Our Approach

As discussed and shown in the previous section, the three BFS traversals required to obtain a certificate H for testing the triconnectivity of a graph take up almost 63% of the total time. A suitable H' can reduce the initial cost of three BFSs. We begin with a kn sized random subgraph H'. However, as is the case in biconnectivity, H' can miss out out on some critical edges required for triconnectivity. It is not a valid certificate yet. We then find the TCCs of H'. The TCCs are treated as super-vertices. These super-vertices form the vertex set of an auxiliary graph F. The edges of the rest of G are then used to define the edges of F. In order to keep the size of F small, the graph F is refined to ensure that at most three edges are present between any two vertices of F. Finally, the TCCs of F are identified which correspond to the TCCs of G. The algorithm is explained in-depth in the following.

Our Algorithm

Algorithm Sample-GPU-TriCC for finding the TCCs of a connected graph G is listed as Algorithm 11. Each of the steps are explained below.

Algorithm 11 Algorithm Sample-GPU-Tricc

1: Input: Biconnected Graph G
2: Output: Triconnected components of G
3: Obtain a spanning subgraph H' from G
4: Find the TCCs of H'
5: Extract F using the TCCs of H' and edges of G
6: Find the TCCs of F

- Step 1 – Obtain a spanning subgraph H' from G: As in the case of Algorithm Sample-GPU-BicC, we identify H' by viewing the edges of G as an edge list and including every m/knth edge for a total of kn edges. Note that no randomness is used as any kn edges suffice for our purpose.

- Step 2 – Find the TCCs of H': We find the TCCs of H' using Algorithm GPU-TriCC. Since H' is a sampled subgraph, it may not be biconnected. However, Algorithm GPU-TriCC only requires the ears and not their numbering. Therefore, we modify the ear decomposition algorithm of Ramachandran [127] to find an open ear decomposition within individual biconnected components. Due to this modification, the ears are identified correctly, though they may not be correctly numbered.

- Step 3 – Constructing F using the TCCs of H' and edges of G: The TCCs of H' are compressed into super-vertices. Since a vertex in a separating pair can belong to multiple triconnected components, vertices in each separating pair are treated as independent super-vertices. These super-vertices form the vertices of F. The edges of F are identified in three steps. First, an edge is added between two nodes of F if there exists an edge $vw \in E(G)$ such that v and w are part of different TCCs of H'. In second step, F is filtered to ensure that no more than three edges are present between any two vertices of F. This is done to keep the size of F as small as possible. In the third step, we convert F to be a simple graph. To this end, for every two nodes in F with more than one edge between them, we split each such edge by introducing auxiliary vertices.

  Similar to the arguments provided in Section 5.4.1, we note that the graph F has the property that if vertices a, b, c of G have at least three vertex-disjoint paths between them in G, then either they belong to the same super-vertex of F, or the super-vertices of G containing these vertices have at least three vertex-disjoint paths between them in F. Therefore, F can be used to identify the triconnectivity and the triconnected components of G.

- Step 4 – Find the TCCs of F: We run Algorithm GPU-TriCC on F to generate the TCCs of F. These components can be used to identify the triconnected components of G.

5.5.5 Implementation Details

As can be noticed from [116], for the graph triconnectivity problem, some computations such as BFS and LCA traversals are common to the biconnectivity problem. In this case too, on the GPU, we therefore use the BFS implementation from [114]. Open ear decomposition is implemented through sorting and LCA traversals. Sorting is performed using Thrust library [6]. LCA traversals are done by assigning a thread to every non-tree edge. Generating the bridge graph for every ear involves finding the connected components of various appropriately defined subgraphs. For this purpose, we use the GPU based algorithm from Soman et al. [145]. Generating the star graph with respect to every ear and the subsequent identification of triconnected components can also be done on a GPU by expressing the computation as a sequence of multiple kernels.

5.5.6 Experimental Results, Analysis, and Discussion

The experimental platform we use for our experiments is described in Section 5.4.3. We scheduled the above algorithm on 1024 threads per blocks.

Dataset

We use two different datasets for our experiments. We use the real-world datasets [43] from Table 5.1. In addition, we use random graphs generated according to the Erdos-Renyi model [24] that have a fixed number of TCCs. All the graph we consider are undirected and unweighted. Key properties of the graphs are shown in Table 5.2.

Results

We compare the performance of Algorithm Sample-GPU-TriCC to that of Algorithm Cert-GPU-TriCC. As noted earlier, Algorithm Cert-GPU-TriCC is to the best of our knowledge, the fastest algorithm on GPUs for finding the triconnected components of a graph.

Table 5.2: Graphs used in our experiments for triconnectivity. In the table, K refers to a thousand and M refers to a million.

Graph	Nodes	Edges	Description
Real-World Graphs			
Same as in Table 5.1			
Random Graphs			
rand-Tricc1	500K	30M	1 TCC
rand-Tricc2	500K	30M	10000 TCCs

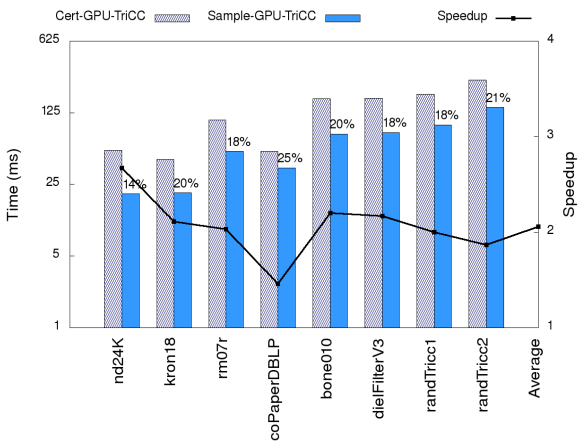


Figure 5.9: Figure showing the time taken by Algorithms Cert-GPU-TriCC and Sample-GPU-TriCC on the graphs listed in Table 5.2. Numbers on the right bar indicate the percentage of time spent by Algorithm Cert-GPU-TriCC in BFS operations. The secondary Y-axis gives the speedup of Algorithm Sample-GPU-TriCC over Algorithm Cert-GPU-TriCC.

Figure 5.9 shows the time taken by Algorithm Sample-GPU-TriCC on the graphs listed in Table 5.2. As can be observed, Algorithm Sample-GPU-TriCC achieves a speedup of 2.1x on average over Al-

gorithm Cert-GPU-TriCC. Moreover, the percentage of time spent in BFS operations by Algorithm Sample-GPU-Tricc is now on average 17%. The value of k is set at 4 in this experiment.

We now proceed to study the performance of Algorithm Sample-GPU-TriCC as k is varied. On graphs rm07r and rand-Tricc1, Figures 5.10 and 5.11 respectively show the results of this study. As k increases, the size of H' increases resulting in increase in the time taken by Step 2. On the other hand, the decrease in the size of F with increasing k reduces the time taken in Step 4. The choice of k is to be made considering this trade-off. From our experiments, we note that $k = 4$ is a good choice in the case of triconnectivity. In Figures 5.5 and 5.6, the vertical line at $k = 4$ indicates that the minimum for total time is achieved at $k = 4$. However, values of k between 4 and 6 offer a near equal minimum.

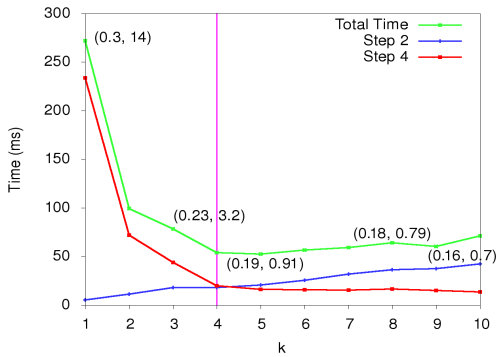


Figure 5.10: Figure represents the time taken by Algorithm Sample-GPU-TriCC on the graph rm07r as k is varied. Tuples on the line labeled "Total Time" show the number of nodes and edges of F in millions at various values of k.

Discussion

One can observe in Figure 5.10 and Figure 5.11 or even in Figure 5.6 and Figure 5.5 (in Section 5.4), that the time spent in Step 4 does

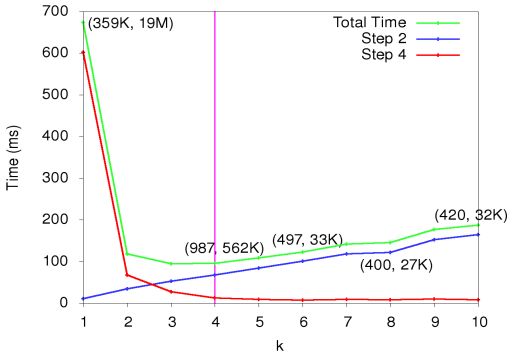


Figure 5.11: Figure represents the time taken by Algorithm Sample-GPU-TriCC on the graph rand-TriCC1 as k is varied. Tuples on the line labeled "Total Time" show the number of nodes and edges of F at various values of k.

not decrease significantly with increasing k. This is due to the fact that after some k, most of the biconnected/triconnected components of G are identified via H' alone.

In general, Algorithm Cert-GPU-TriCC involves more BFS operations than Algorithm Cert-GPU-BiCC. Thus, it seems that Algorithm Sample-GPU-TriCC should benefit more from our technique than Algorithm Sample-GPU-BiCC. However, as shown in Figure 5.4 and Figure 5.9, our technique results in a near similar speedup in both cases. This is due to the reason that for the graphs we considered in our dataset, and in general, we expect more triconnected components than biconnected components. So, the size of the auxiliary graph F generated using our technique is larger in the case of triconnectivity as compared to biconnectivity.

5.6 Conclusions

In this chapter, we studied how parallel graph connectivity algorithms can be improved by reducing the time spent in BFS operations. Our

results indicate that a significant gain in performance can be obtained by reinterpreting algorithms to perform BFS on graphs that are much smaller in size compared to the input graph. We believe that our approach can be useful in other settings too. As our results show promise, in future, we want to also understand how to theoretically analyze the speedup that can be obtained using our approach.

Chapter 6

Shortest Paths in Graphs

We now study ideas for shortest paths in graphs.

6.1 Our Approach for APSP

We start with graphs that are biconnected (Section 6.1.1) and extend our approach to graphs that are not biconnected in Section 6.1.2.

6.1.1 APSP for Biconnected Graphs

We now present our algorithm for APSP on biconnected graphs using the technique of ear decomposition. Algorithm 12 describes a brief pseudocode of our three phase algorithm followed by the details of each phase. The label {**cpu,gpu**} in Algorithm 12 is used to indicate that the corresponding task is computed in a heterogeneous manner on both the GPU and CPU. The labels {**cpu**} (*resp.* {**gpu**}) are used to indicate tasks that are executed solely on the CPU (GPU).

Preprocessing

Let G be a sparse and biconnected graph. It is known that a biconnected graph possesses an ear decomposition. An ear decomposition

Algorithm 12 APSP(G)

```
1: /* Phase I: Preprocessing */
2: {gpu}: G^r = Reduce(G)
3: /* Phase II: Processing */
4: {cpu,gpu}:
5: for each s ∈ V(G^r) do
6:     DIJKSTRA(G^r, s) /* Find shortest paths from s */
7: end for
8: /* Phase III: Post-processing */
9: {cpu,gpu}:
10: for each s ∈ G \ G^r in parallel do
11:     UPDATE_DISTANCE(s).
12: end for
```

of a graph $G = (V, E)$ is a partitioning of the edges of G into simple paths (ears) $P_0, P_1, \cdots$, as follows (see also [126]).

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

In such a decomposition, nodes of degree two, except possibly those on ear P_0, appear on exactly one ear. We show that such nodes of degree two can be removed from G. We call the resulting graph of G as the reduced graph G^r. One can formalize the notion of the reduced graph $G^r = (V^r, E^r, W^r)$ as follows. The nodes of G^r are the nodes of G that have a degree at least three. Two nodes v and w in G^r are neighbors if and only if v and w belong to a common ear P of G and have no nodes of degree three or more in between them on the ear P. The weight of an edge vw in G^r set as the sum of the weights of the edges $vx_1, x_1x_2, \cdots, x_iw$ in G such that nodes $x_1, x_2, \cdots, x_i$ are consecutive vertices on P with degree two in G and are in between v

and w on the ear P in G. For a node x_i, $i > 1$, of degree two on ear $P = (x_1 x_2 \cdots x_k)$ in G, we define functions *left* and *right* of x_i in G^r, denoted Left(x_i) and Right(x_i), as the nodes of degree at least three on P that are closest to x_i towards x_1 and x_k respectively. For instance, in the above example with $v, x_1, x_1, x_2, \cdots, x_i, w$ being on the same ear in that order with v and w having degree more than 2 and the x_is having degree of 2, Left(x) = v and Right(x) = w.

Notice that during the construction of the reduced graph, there could be multiple edges between nodes in the reduced graph. In this case, since we are interested in shortest paths, we retain the edge with the shortest weight and discard the remaining edges.

Phase II: Processing

In this phase, we find the shortest paths between all pairs of nodes in the reduced graph G^r. From each node v in G^r, we essentially run the algorithm of Dijkstra [48] that finds the shortest paths from v to all other nodes t in G^r. In short, we obtain all the shortest path values $S^r[s,t] \mid \forall\{s,t\} \in G^r$ We use the GPU implementation of Dijkstra's algorithm due to Harish et al. [80]. On CPU, we run multiple instances of Dijkstra's algorithm from different vertices of G^r. Each instance of Dijkstra's algorithm is run on an individual thread. The algorithm of Dijkstra is preferred over other shortest path algorithms for reasons including the ability to run each instance of Dijkstra's algorithm independently by a thread and the work involved in Dijkstra's algorithm depends linearly on the number of edges in the graph.

Post-processing

In this phase, we use the shortest paths in G^r to compute the shortest paths across all pairs of nodes in G. Consider the shortest paths originating from a node x in $G \setminus G^r$ with Left(x) = ℓ_x and Right(x) = r_x (refer to Figure 6.1). For paths from x that end at nodes y with Left(x) = Left(y) and Right(x) = Right(y), the shortest xy path is

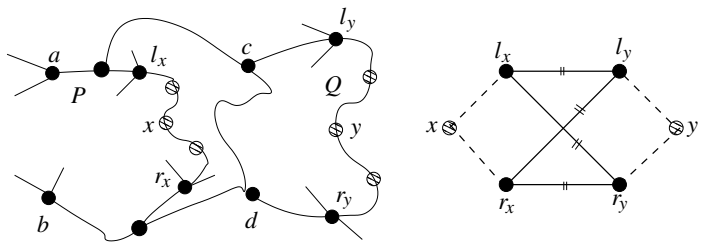


Figure 6.1: The shortest path between nodes x and y has to use one of ℓ_x or r_x to leave the ear $P = (a \cdots \ell_x \cdots x \cdots x \cdots r_x \cdots b)$ and one of ℓ_y or r_y to enter the ear $Q = (c \cdots \ell_y \cdots y \cdots r_y \cdots d)$. If the four pairwise shortest paths between ℓ_x, r_x, and ℓ_y, r_y are given, marked with double lines in the figure on the right, the shortest path from x and y can be obtained as the shortest among the four possible paths. In the figure on the left, nodes completely filled appear in the reduced graph and shaded nodes are removed in Stage II of preprocessing.

either the unique xy-path along P that does not use ℓ_x and r_x, or the path $x - \ell_x - r_x - y$. Paths from x that end at nodes y such that x and y have different Left and/or Right nodes, have to necessarily go via ℓ_x or r_x.

Let $S[s,t]$ store the weight of the shortest path between s and t in G. Clearly, for all $u, v \in V^r$, $S[u,v] = S^r[u,v]$ since the reduced graph preserves shortest-path distances between vertices of degree at least 3. To compute shortest path $S[x,v]$ between any $v \in V^r$ and any $x \in V \setminus V^r$, consider the ear P on which x lies and let $\text{Left}(x) = \ell_x$ and $\text{Right}(x) = r_x$ (v may coincide with ℓ_x or r_x). We can compute

$$S[x,v] = \min\left\{S^r[\ell_x, v] + wt(x,\ell_x), S^r[r_x, v] + wt(x, r_x)\right\}$$

Now we consider the most general case of computing $S[x,y]$ for nodes $x, y \in V \setminus V^r$. For this case, let ℓ_x, r_x and ℓ_y, r_y be the Left and Right nodes of x and y respectively (ℓ_x may coincide with ℓ_y or r_y, and similar reasoning applies to r_x). Using the same idea as above,

we can compute

$$S[x,y] = \min \left\{ \begin{array}{l} wt(x,\ell_x) + S^r[\ell_x,\ell_y] + wt(\ell_y,y), \\ wt(x,\ell_x) + S^r[\ell_x,r_y] + wt(r_y,y), \\ wt(x,r_x) + S^r[r_x,\ell_y] + wt(\ell_y,y), \\ wt(x,r_x) + S^r[r_x,r_y] + wt(r_y,y) \end{array} \right\}$$

The call to UPDATE_DISTANCE(s) in this phase essentially computes $S[s,t]$ for all $t \in G$ by using the appropriate formula described above.

6.1.2 Extension to General Graphs

As we are interested in large sparse graphs, it is very likely that our graphs are not 2-connected or even 2-edge-connected. Quite contrary, large sparse graphs arising out of real-world phenomena tend to have several 2-connected components of varying sizes. In such a scenario, such graphs do not have a ear decomposition as being 2-edge-connected is a necessary (and sufficient) condition for having an ear decomposition [126].

To make use of Algorithm 12, in a preprocessing step we start by partitioning G into its biconnected components $G_1, G_2, \cdots$ each of which is 2-connected. We now obtain an ear decomposition of $G_1, G_2, \cdots$, and obtain their respective reduced graphs $G_1^r, G_2^r, \cdots,$. Let $A_1^r, A_2^r, \cdots$, denote the set of *articulation points (APs)* in $G_1^r, G_2^r, \cdots$, respectively. We let $a = |\cup_i A_i^r|$. The quantity a denotes the number of articulation points in G.

In the processing step, we now find the shortest paths between pairs of nodes in each G_i^r individually, and in parallel. We store the computed results in a table A_i that stores the shortest distance between pairs of nodes in G_i.

Our post-processing is now spread across two stages. In Stage 1, for each $i = 1, 2, \cdots$ we extend the shortest paths between pairs of nodes in the ear graph G_i^r to shortest path between pairs of nodes in G_i. This is done as described in Section 6.1.1. These results are also stored in tables A_i for $i = 1, 2, \cdots$. To compute shortest paths

across pairs of nodes in different biconnected components we proceed as follows.

In Stage 2 of post-processing, we use the notion of the block-cut tree of a graph as described in [16]. The block-cut tree B of a graph G has nodes corresponding to the biconnected components of G. An *edge* exists between two nodes v and w in B if the corresponding biconnected components in G share an articulation point. We use the block-cut tree to find the shortest distance from each articulation point to every other articulation point in G. These results are stored in a table A of size $a \times a$. We use A to compute distance between nodes of different biconnected components, G_i and G_j.

For nodes $n_1 \in G_i$ and $n_2 \in G_j \mid i \neq j$, $d(n_1, n_2) = \min(d(n_1, a_1) + d(a_1, a_2) + d(a_2, n_2))$ where a_1 and a_2 are the AP's corresponding to G_i and G_j which are on the path from G_i and G_j.

6.1.3 Implementation Details

In our heterogeneous implementation of the processing and the post-processing step, we notice that work balancing is needed between the CPU and the GPU. Since a static approach for work balancing can fall short of the desired work balance, we use our custom work queue (from [88]). The workunits correspond to the processing (*resp.* post-processing) with respect to each biconnected component of the graph. For reasons of efficiency, the work units are sorted according to the size of the biconnected component and arranged in sorted order so that the GPU starts accessing the bigger workunits. If the graph is already biconnected and we are using Algorithm 12, then the workunits can correspond to the processing required with respect to a vertex. As is done in [88], the CPU and the GPU access workunits from the queue from either end points, and also in proportion to the number of threads supported on the CPU and the GPU.

Since the matrix A is needed by both the CPU and the GPU in the post-processing step, the matrix A is kept in the memory of both the CPU and the GPU. This forces us to limit our experiments to fit the available space on the GPU. One advantage of our method is that

the space used to store all the shortest path values is in $O(a^2 + \sum_i n_i^2)$ where n_i refers to the number of nodes in G_i. In most sparse graphs, the above quantity is usually much smaller than $O(n^2)$ that is required to store the shortest path values. See also Table 6.1 for evidence for this phenomenon.

6.1.4 Results and Analysis

In this section, we show experimental results of our algorithm and also compare the results with related approaches. We start by describing our experimental platform and the datasets used.

Our Experimental Platform

mention CPU, GPU details. Refer to Chapter 2.

Datasets

We experiment on two datasets: general graphs and planar graphs. General graphs for our experiment are taken from the dataset of sparse graphs from the University of Florida Sparse Matrix Collection [53]. These graph come from domains such as geometric, social networks, collaboration, and peer-to-peer networks. The planar graphs shown in Table 6.1 were generated using the OGDF framework [40] using methods that generate connected graphs. Some of the characteristics of the graphs considered are listed in Table 6.1. It can be observed that our dataset has a good diversity. The size of the graphs ranges from 10 K to 130 K, and the number of nodes of degree two range between 0% to 60%. Further, the size of the largest BCC as a percentage of edges also varies between 80% to 98%.

Results

We now compare the results of our algorithm labeled as "Our Approach" with two related approaches: the approach of Djidjev et al. [56] that works for planar graphs, and the approach of Banerjee et

Table 6.1: List of sparse graphs that we use in our experiments. In the column labeled "Largest BCC (%)", we show the number of edges in the largest BCC of the graph as a percentage of the number of edges in the graph. The column labeled "Nodes Removed (%)" shows the percentage of the nodes removed by our algorithm during the pre-processing step. The column "Our's memory" lists the total memory used by our algorithm as compared to the total memory that is used in APSP shown in the column labelled "Max Memory". Note that storage requirements are shown rounded to the nearest MegaByte (MB).

Graph	**|V**	**|E|**	**#BCCs**	**Largest BCC (%) (% $\|E\|$)**	**Nodes Removed (% $\|V\|$)**	**Our's Memory** (MB)	**M**
Graphs taken from [53]							
nopoly	10K	30K	1	100	0.018	443	4
OPF 3754	15K	86K	1	100	1.98	873	9
ca-AstroPh	18K	198K	647	98.43	15.85	970	1
as-22july06	22K	48K	13	99.9	77.60	851	2
c-50	22K	90K	1	100	52.04	651	1
cond mat 2003	31K	120K	2157	80.52	26.88	1826	3
delaunay n15	32K	98K	1	100	0	4096	4
Rajat26	51K	247K	5053	95.17	32.92	7176	9
Wordnet3	82K	132K	156	98.92	77.24	4663	2
soc-signs--epinions	131K	841K	609	99.7	67.86	12932	6
Graphs generated using the OGDF framework [40]							
Planar_1	19K	54K	46	99.55	12.42	1278	1
Planar_2	25K	64K	164	93.65	5.63	1627	1
Planar_3	30K	70K	298	96.53	19.72	2068	2
Planar_4	36K	94K	175	98.37	18.56	3890	4
Planar_5	41K	128K	223	95.63	16.34	4350	4

al. [16] that works for general graphs. We start by briefly describing these approaches.

Comparison with Djidjev et al. [56] for planar graphs The algorithm of Djidjev et al. [56] works as follows. As part of their approach, Djidjev et al. [56] starts by partitioning the input graph into k parts using the METIS decomposition [98]. The partitioning is used to define a boundary graph that contains nodes of the input graph that are the end points of edges that go across partitions. Once the shortest paths in each partition are obtained, the boundary graph is augmented with edges uv such that u and v are in the same partition and the weight of the edge uv is set to be the shortest distance between u and v as computed in the previous step. The shortest paths in the boundary graph are computed in a recursive fashion followed by the shortest paths in each partition. For further details we reader can refer to [56].

It is worthwhile to note that while the algorithm presented by Djidjev et al. [56] works for any general graph, the approach is efficient for particular classes of graphs, including planar graphs with the property that the number of vertices in the boundary graph is guaranteed to be small. For this reason, their experimental results are shown only for planar graphs.

The speedup achieved by our implementation on planar graphs compared to Djidjev et al. [56] approach is shown in the Figure 6.2. It contains the overall timings for our implementation along with Djidjev's for planar graphs on the Y1-axis on the right. The Y2-axis denotes the speedup achieved by our algorithm. An average speedup of 2.2x achieved is mentioned in the right most column of the Figure. As most planar graphs contain a good percentage of degree-2 vertices, we conclude that our approach for real world planar graphs is more beneficial compared to [56].

Comparison with Banerjee et al. [16] The algorithm provided by Banerjee et al. [16] works by decomposing the graph as follows. Given an input graph G, it constructs a block-cut tree for G. It then

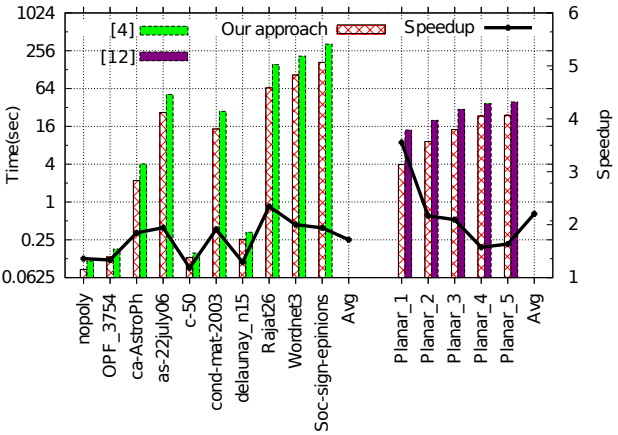


Figure 6.2: Figure displays the absolute time taken by our approach, labeled "Our Approach" compared to [16] for general graphs and [56] for planar graphs.

computes the shortest paths within each biconnected component and later extends the computation of shortest paths across the blocks. The algorithm also optimizes the run time by removing the iterative pendants vertices. That is, it initially removes vertices of degree-1 from the graph. It then checks if the degree of any vertices adjacent to the vertices removed in the first iteration, degenerates to 1. This method, though reduces the computation time compared to other existing algorithms for real world sparse graphs, it does not effectively benefit from the degree-2 vertices present in the graph. Also this model requires more storage compared to our approach. For further details interested reader can refer to [16]. To illustrate our algorithm's computational efficiency we compare our results with Banerjee et al. [16] for general graphs.

Figure 6.2 shows the relative improvement of our approach compared to Banerjee et al. [16] implementation for general graphs. The plot contains the overall timings for both the implementations on the Y1-axis on the left. The timings displayed on the Y1-axis are on

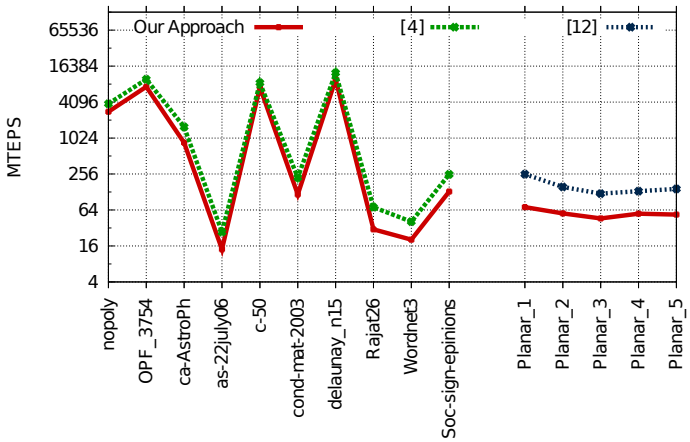


Figure 6.3: MTEPS achieved by our algorithm, labeled "Our Approach" and that of [16] for general graphs and [56] for planar graphs.

a logarithmic scale. The Y2-axis denotes the speedup achieved by our algorithm with respect to that Banerjee et al. [16]. The average speedup achieved is 1.7x.

Another way to study the scalability of parallel graph algorithms is to use the metric MTEPS standing for Million Traversed Edges Per Second. This metric is computed as the ratio of the product of the number of edges and number of vertices over the time taken in seconds. A higher MTEPS indicates a more scalable algorithm. Figure 6.3 provides the MTEPS achieved by the approaches of Djidjev et al. [56] and Banerjee et al. [16] on planar and non-planar graphs respectively in comparison to our approach. We finally note that we are limited in this comparison by the space available on the GPU although our approach needs lesser space compared to that of both [16, 56].

Chapter 7

Computing Metrics on Graphs

A class of graph computations that have become popular in recent years are based on metrics on graphs. Metrics on graphs assign a value to the nodes and/or edges of a graph based on certain criteria. Computing metrics on nodes of a graph is an essential step in understanding the properties of the graph. In this chapter, we concentrate on metrics such as diameter, pagerank and betweenness-centrality in sections 7.1 to 7.3 respectively.

7.1 Diameter

Summary of Crescenzi et al. TCS 2013 in parallel.

7.2 Pagerank

Pagerank as a metric is being used to measure the importance of nodes in not only web graphs but also in social networks, biological networks, road networks, and the like. The core of the computation of pagerank can be seen as an iterative approach that updates the pageranks of nodes until the values converge.

However, as real-world graphs such as road networks and the web have a large size, one needs to design efficient techniques to address the challenges of scale. In addition to parallelism that can be exploited, it is important to also look for specific properties of graphs and their impact on the algorithm.

In this section, we present four algorithmic techniques that optimize the pagerank computation on real-world graphs. The techniques are presented with the aim of exploiting the nature of the real-world graphs and eliminating redundancies in the pagerank computation. Our techniques also have the advantage that with little extra effort one can quickly identify which of the techniques will be suitable for a given input graph.

We implement our algorithm on an Intel i7 980x CPU running 12 threads using OpenMP Version 3.0. We study our techniques on four classes of real-world graphs: web graphs, social networks, citation and collaboration networks, and road networks. Our implementation achieves an average speedup of 32% compared to a baseline implementation.

7.2.1 Introduction

Pagerank computation introduced by Brin and Page [120] is considered to be a fundamental advancement in the development of search and related technologies. Informally, the pagerank computation assigns a rank to every webpage where the rank indicates the importance of a web page. Pagerank type metrics have since become popular and are used in evaluating the relative importance of nodes in other classes of networks including social networks, citation networks, biological networks, road networks, and the like [72]. It is therefore not surprising that there has been a lot of research interest in the past decade on pagerank computation [29, 57, 95, 102, 132, 163], to mention a few.

The pagerank computation proceeds in iterations and in each iteration the current pagerank of each node is updated by using the pagerank of its incoming neighbors. The computation stops when the

pagerank values at all nodes change only by a small value across successive iterations. In each iteration, along each directed edge $e = (u, v)$, a fraction of the pagerank value of u is transferred to the *new* pagerank value of v. This computation offers opportunities for optimization where certain edges of the graph can be identified as *redundant* for one or more iterations. These optimizations can be achieved by a deeper understanding of the structure of popular real-world graph classes such as web graphs, road networks, and citation networks.

There have been a number of works on parallel algorithms to compute the pagerank of nodes in a graph. However, some of the techniques are directed only towards web graphs e.g., [8, 57, 95] and may not translate to other real-world graphs. With pagerank being used for other purposes beyond its original intent [72], it is important to have an efficient algorithm for pagerank computation for several classes of real-world graphs.

It is to be noted that as real-world graphs increase in size, several scalability issues come to the fore in the process of algorithm design and implementation. Exploiting the parallelism in the computation provides only a limited succor. Hence, one needs to deploy efficient techniques to address the scalability issue arising out of the large sizes of the real-world graphs. Such a line of investigation has been pursued recently in finding the strongly connected components by Olukotun et al. [85], graph traversals and shortest paths [14,55], graph connectivity [64, 94], and the like on a variety of multicore and heterogeneous execution platforms.

In this section, we present techniques to identify and eliminate the redundancies in the pagerank computation. We name our techniques with the acronym STIC-D for SCC and Topological Ordering, Identical Nodes, Chain Nodes, and Dead Nodes. The first technique makes use of the observation that sparse directed graphs tend to have a large number of strongly connected components (SCCs). Further, we observe that the block graph of a graph plays a big role in how the pagerank values of nodes converge to their final values. We call this technique as *SCC and Topological Order*. Our second technique,

called *Identical Nodes* is based on identifying nodes with an identical incoming neighbourhood. It can be observed that such nodes have an identical pagerank value, hence the computation can be done for only one of the nodes in every class of identical nodes. Our third technique, called *Chain Nodes* is based on the fact that one can shortcut nodes along paths as their contribution to the computation of pagerank can be captured by a succinct formula. We also introduce a fourth technique based on approximation, called *Dead Nodes*, that involves retiring nodes that do not have a big change across iterations.

While each of the above techniques are simple, there are additional challenges we address to translate these techniques to efficient algorithms. Note that the pagerank computation itself has near-linear work in practice. For most real-world graphs of varying sizes, the number of iterations required for the pagerank computation to converge is usually less than 100. This means that the preprocessing time and any post-processing time required as part of the proposed optimizations have to be very light-weight so as not to offset the gains accrued from the proposed optimizations. Further, it is likely that a specific subset of the above techniques will be appropriate for a given graph. As we aim for techniques that are general-purpose, one also needs to have fast, simple, and effective mechanisms that can identify the applicable techniques for a given graph.

To validate our techniques, we consider four types of real-world graph classes: web graphs, social networks, citation and collaboration networks, and road networks, and show that our techniques can improve the computation of pagerank by a factor of 32% on average.

Motivation

One of the key motivations of our work is to understand the structural properties of real-world graphs and their impact on algorithms. Specific to the computation of pagerank, we seek properties that can allow us to reduce the number of computations that are invoked for each node of the graph. In this direction, we first note that real world graphs being sparse in nature tend to have several strongly connected

S. No.	Graph Name	No. nodes	No. SCCs	Source
1	soc-LiveJournal1	4,847,571	971,232	[3]
2	wiki-Talk	2,394,385	2,281,879	[3]
3	webbase-2001	118,142,155	41,126,852	[23]
4	Stanford_Berkeley	683,446	109,238	[4]
5	GAP-twitter	61,578,415	27,970,913	[3]

Table 7.1: The number of strongly connected components in five real-world graphs.

components (SCCs). Figure 7.1 shows evidence for the same for some real-world graphs. As can be noticed from Figure 7.1, real-world graphs have a large number of SCCs with a small size, and very few SCCs of a large size. This property indicates that a decomposition based on SCCs can be useful in parallel algorithms for pagerank computation.

Now, consider two SCCs H_1 and H_2 of a graph G such that nodes in H_2 have some incoming edges from nodes in H_1. It can be noticed that nodes in H_2 can start computing the pagerank only after nodes in H_1 have converged to their final pagerank value. This property establishes a topological order in which the SCCs of G can be processed during the pagerank computation. Further, once nodes in H_1 converge to their final pagerank values, their contribution to nodes in H_2 does not change across iterations of the pagerank computation for nodes in H_2. These observations form the basis of two of our techniques.

Other structural properties of real-world graphs that we make use of in this work include identifying nodes with identical incoming neighborhoods and also nodes that lie on long directed paths. We show in the subsequent sections that these structural properties offer good reduction in the amount of computation required to arrive at the pagerank values of nodes in a graph.

Related work

Computing metrics on graphs in parallel has been witnessing a renewed interest in recent years. In the context of pagerank, Broder et al. [28] characterized the structure of web graphs as having a bow-tie nature with one large strongly connected component that also has a large number of incoming and outgoing edges. Arasu et al. [8] use the characterization of Broder et al. [28] and represent the pagerank computation as solving a set of independent matrix equations on a block upper triangular matrix. Kamvar et al. [95] extend the characterization from Broder et al. [28] to partition the input graph into blocks where a block corresponds to a collection of nodes that are also physically related in the context of a web graph by being in the same domain. They proceed to compute a page rank within each block, called the local pagerank, a pagerank for the blocks called as BlockRank, and finally the global pageranks using the local pageranks and the BlockRanks. It is to be noted however that the final ranks computed by Kamvar et al. [95] are *approximate* ranks and can differ from the pagerank values as computed without using the block structure of the graph.

Similar approximation based approaches can be seen in the work of SiteRank by Wu and Aberer [163], the U-Model work of Broder et al. [29], the ServerRank work of Wang et al. [157], and the HostRank/DirRank work of Eiron et al. [61]. Kohlschutter et al. [102] extend the results of Kamvar et al. [95] to obtain exact pageranks. But one limitation of these works [61, 95, 102, 157, 163] is that their algorithms are tailored for web graphs and may not work well for general purpose graphs. As pagerank and related computations gain prominence in other domains, generic techniques are of interest.

Parallel computation of pagerank on other emerging architectures such as GPUs has been studied by Duong et al. [57]. The work of [57] does not introduce any algorithmic optimizations in the computation of pagerank. On the other hand, we believe that our algorithmic techniques will be applicable to other architectures too.

Some of the techniques that we propose are found to be relevant

in computing centrality metrics on graphs. For instance, Sariyuce et al. [134] use graph decomposition into biconnected components, removing identical nodes and nodes of degree one, to compute the betweenness-centrality measure of nodes in a graph.

Expressing the pagerank computation as a Markov chain and solving for the steady state transition probabilities of the underlying Markov chain is a mechanism used by many authors. Pandurangan et al. study such an approach in the distributed setting [132].

Organization of the Section

The rest of the section is organized as follows. In Section 7.2.2, we give an algorithmic background for pagerank. We present the baseline algorithm in Section 7.2.2. In Section 7.2.3, we present the algorithmic techniques. Section 7.2.3 covers the implementation details of the algorithm. Experimental results are presented in 7.2.4.

7.2.2 Preliminaries

Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices and $m = |E|$ edges. Let $IN(u)$ denote the set of nodes that have an incoming edge to u. Let outdeg(u) denote the number of edges from u to other nodes. The pagerank of a node u, denoted $pr(u)$, is then given as follows.

$$pr(u) = \sum_{v \in IN(u)} \text{contribution}(v \to u) + \frac{d}{n} \tag{7.1}$$

In Equation 7.1, d is called the damping constant and has its genesis in the manner in which pagerank is usually interpreted. The pagerank values represent a probability distribution where the pagerank of a node denotes the probability of a random walk to visit that node. The damping constant d can be interpreted as the probability that the random walk stays at the same node in the next step. The value of d is taken to be 0.15 usually. The quantity contribution$(v \to u)$ is defined as follows.

$$\text{contribution}(v \to u) = (1-d) \cdot \frac{pr(v)}{\text{outdeg}(v)} \tag{7.2}$$

One can also combine Equations 7.1,7.2 to arrive at the following simplified equation.

$$pr(u) = (1-d) * \sum_{v \in IN(u)} \frac{pr(v)}{\text{outdeg}(v)} + \frac{d}{n} \tag{7.3}$$

Using this formula, we get a pagerank distribution of nodes of a graph. However this formula is cyclic in nature as two nodes can make contributions to each other if they are part of a directed cycle. One way to resolve this dependency is to apply the formula from Equation 7.3 iteratively over all the nodes until the pagerank values converge. We say that the pagerank values of nodes have converged when there is very little change in their values across an iteration. This can be measured by a function such as the maximum difference of pagerank values or the total difference of pagerank values across an iteration.

Baseline Algorithm

Algorithm 13 translates Equation 7.3 into an iterative pagerank computation. We refer to Algorithm 13 as the baseline algorithm against which we compare our techniques.

We initialize the error as ∞ and the initial pagerank values of all nodes to $\frac{d}{n}$. For reasons of efficiency of parallel computation of pagerank, in Algorithm 13, for each node u, the new pagerank value of u is computed as the sum of contributions from the incoming neighbors of u. The variable `Threshold1` is a constant that reflects the accuracy of the pagerank needed by an application.

Lines 12–17 iterate over all the nodes in parallel. Each node updates its pagerank based on the contributions of its incoming edges. Lines 18–21 calculate the error function as the L_1 norm of the change in the pagerank values of nodes across one iteration. The variable `error` is used to decide whether to proceed for the next iteration or declare convergence.

Algorithm 13 Base Pagerank(G)

```
1: procedure MAIN(Graph G = (V, E), Array outdeg )
2:     pr=Compute((V, E), outdeg)
3:     return pr
4: end procedure
5:
6: procedure COMPUTE(Graph G,Array outdeg )
7:     error = ∞
8:
9:     for all u ∈ V do
10:        prev(u) = d/n
11:
12:    end for
13:    while error > threshold1 do
14:
15:        for all u ∈ V in parallel do
16:            pr(u) = d/n
17:            for all v ∈ V such that (v, u) ∈ E do
18:                pr(u) = pr(u) + prev(v)/outdeg(v) * (1 − d)
19:            end for
20:        end for
21:        for all u ∈ V do
22:            error = max(error, abs(prev[u] − pr[u]))
23:            prev(u) = pr(u)
24:        end for
25:    end while
26:    return pr
27: end procedure
```

7.2.3 Our Algorithmic Techniques

In this section, we describe our algorithmic techniques that help speedup the pagerank computation by eliminating redundancies. The acronym *STIC* stands for our techniques based on *SCCs and Topological Ordering, Identical Nodes, and Chain Nodes*. In brief, the SCCs and Topological Ordering breaks the pagerank computation into computations on smaller subgraphs that are strongly connected and processed in a particular order. The Identical Nodes optimization refers to eliminating the redundant computation for nodes with an identical incoming neighborhood. The Chain Nodes optimization shortcuts directed paths by a directed edge that connects the end points of the path. More details of these optimizations are presented in the following sections.

SCC and Topological Ordering

Recall from Equation 7.3 that in an iterative computation pagerank values of nodes depend cyclically on one another. It is also to be observed that if the pagerank value of all incoming neighbors of a node v converge, then the pagerank value of v will also converge by the next iteration. This observation can be extended to a decomposition of a directed graph into its strongly connected components. For a directed graph $G = (V, E)$, a maximal subset of vertices $U \subseteq V$ such that every pair of nodes in U have at least one directed path between them is said to be a strongly connected component (SCC) (cf [46]). One can also define the block graph H of G where H has one node for each SCC of G. For two nodes $u, v \in H$, there is an edge from u to v if and only if there exist vertices a and b in the corresponding SCCs C_u and C_v of G such that the edge $(a, b) \in E(G)$. The graph H is directed as defined and is also acyclic.

It can be noted that in a topological sort of the nodes of H, if a node $v \in V(H)$ comes after a node $u \in V(H)$, then nodes in the SCC C_v cannot contribute to the pagerank values of nodes in the SCC C_u. Viewed differently, the pagerank computation will benefit if we start processing nodes in C_v after the nodes in all SCCs that have an

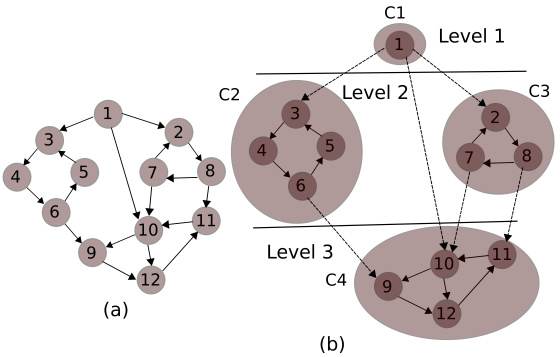


Figure 7.1: The SCCs of a graph and our processing order.

incoming edge to $v \in H$ converge.

Our SCC and Topological Ordering technique uses the above observations as follows. In a preprocessing step, we find the SCCs of the input graph G and perform a topological sort of the block graph H of G. This results in an ordering of the SCCs of G as $C_1, C_2, \cdots$ so that the pagerank values can be computed for C_1, followed by C_2, and so on.

Let us call edges that have end points in different SCCs as cross edges. Figure 7.1 illustrates the above ideas. In Figure 7.1, cross edges are shown as dashed lines. As Figure 7.1(b) shows, there are three levels in the topological order of the block graph of the graph in Figure 7.1(a). We can compute the pageranks of nodes in component C_1, followed by components C_2 and C_3 in Level 2 in parallel, and finally component C_4 in Level 3.

There is one additional redundant computation that can be identified as we compute pageranks according to a topological order of nodes in the block graph. Consider a node $a \in V(G)$ that appears in SCC C of G such that the node $u \in V(H)$ corresponding to C has a rank of r in a topological sort of H. Node a may have incoming neighbors in SCCs with a rank strictly less than r. In our scheme, the pagerank value of such nodes would have already been computed as we process the nodes in component C. Therefore, in the pagerank

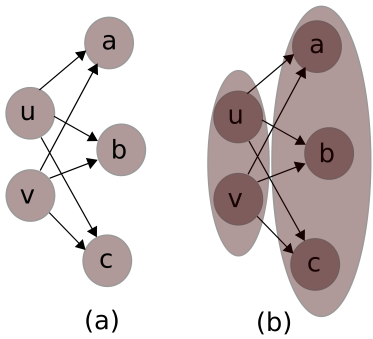


Figure 7.2: Figure (b) illustrates identical nodes in the graph in Figure (a).

computation of node a, the contribution of incoming neighbors of a in components with rank strictly less than r does not change across iterations. Hence, the computation corresponding to such cross edges need be performed only once. In other words, the pagerank of nodes in SCC C can be initialized to the sum of their contributions from incoming neighbors from SCCs with a strictly smaller rank than the rank of C.

Identical Nodes

In this optimization, we notice that the pagerank of a node is completely dependent on its incoming neighbors. So, it follows that two nodes that have identical incoming neighbors would also have the same pagerank value. Indeed, such is the case at the end of each iteration also. We therefore call two nodes as identical if they have the same set of incoming neighbors. The notion of identical nodes allows one to compute the pagerank of one *representative* node in every class of identical nodes. The pagerank value of the representative node for each class is referred to by all the nodes in that class when they need the pagerank value.

We illustrate the above in Figure 7.2 where nodes a, b and c have

u and v as common neighbors. Notice also any two nodes that are identical will always belong to the same SCC or will be in an SCC of size one. This helps in the implementation since it will be easy to all identical nodes will be processed along with the component.

Chain Nodes

This algorithmic optimization concerns nodes along directed paths. On a directed path $P = \langle u = u_0, u_1, u_2, \cdots, u_k = v \rangle$, notice that nodes u_i with $1 \leq i < k$ have exactly one incoming and one outgoing node. Similar to the pagerank computation described in Equation 7.3, it is now possible to compute the contribution of the pagerank of u to the pagerank of v in on step instead of propagating the contribution of u to v via the edges of P in k iterations. Lemma 6 shows the exact contribution of u to the pagerank of v.

Lemma 6 *Let P=$\{u = u_0, u_1 \cdots, u_k = v\}$ be a directed path. Then the contribution of u on v is given as follows.*

$$\begin{aligned} \text{contribution}(u \to v) &= \frac{pr[u]}{outdeg[u]} * (1-d)^k \\ &\quad + \frac{(1-d)*(1-(1-d)^{k-1})}{N} \end{aligned}$$

Proof: We prove the lemma by induction. Notice that for $k = 1$, the lemma essentially restates Equation 7.2. Assuming the induction hypothesis for a directed path of length k, we now show the induction step. By the induction hypothesis, on a directed path $u_0, u_1, \cdots, u_k$, by Equation (7.1), we can say that:

$$\begin{aligned} pr[u_k] &= \text{contribution}(u \to u_k) + \tfrac{d}{n} \\ pr[u_{k+1}] &= \text{contribution}(u \to u_{k+1}) + \tfrac{d}{n} \end{aligned} \tag{7.4}$$

Also by Equation (7.3) and the fact that the in-degree of u_{k+1} and out-degree of u_k is one, we get that $pr[u_{k+1}] = pr[u_k] * (1-d) + \frac{d}{n}$. From above two equations, we have that $\text{contribution}(u \to u_{k+1}) = pr[u_k] * (1-d)$.

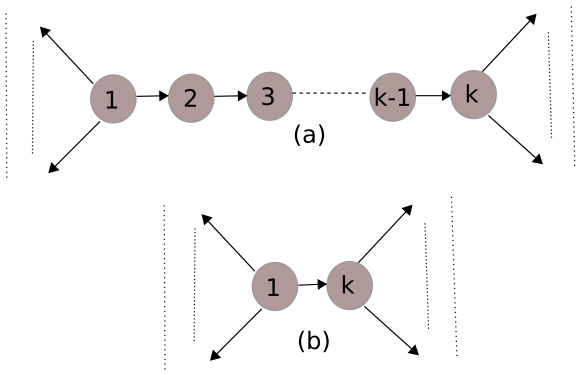


Figure 7.3: A directed chain of nodes can be compressed to a single edge as illustrated in the figure.

By Equation (7.4) and the previous equation, we can get:

$$\begin{aligned}
&\text{contribution}(u \to u_{k+1}) \\
&= \left(\text{contribution}(u \to u_k) + \tfrac{d}{n}\right) \cdot (1-d) \\
&= \left(\frac{pr[u] \cdot (1-d)^k}{outdeg[u]} + \frac{(1-d)\cdot(1-(1-d)^{k-1})+d}{n}\right) \cdot (1-d) \\
&= \frac{pr[u] \cdot (1-d)^{k+1}}{outdeg[u]} + \frac{(1-d)-(1-d)^k+d}{n} \cdot (1-d) \\
&= \frac{pr[u] \cdot (1-d)^{k+1}}{outdeg[u]} + \frac{(1-d)\cdot(1-(1-d)^k)}{n}
\end{aligned}$$

□

Figure 7.3 illustrates this optimization. As can be seen, on a directed path of k nodes, we can remove all but one edge and the end points of the path.

Using Lemma 6, it is possible to perform two related optimizations. On the path P, nodes u_i for $1 \leq i < n$ can be removed from the graph and hence the pagerank computation. Nodes u and v can be joined with a directed edge (u, v). During the pagerank computation, we use the result of Lemma 6 to set the contribution of u to the pagerank of v.

Marking Nodes Dead

The final optimization that we introduce is to mark certain nodes as Dead Nodes. This is done as follows. While calculating the pagerank values of nodes a graph using the iterative method, it is likely that the pagerank values of some nodes do not change considerably across iterations. We consider the possibility that for such nodes, their additional contribution to their outgoing neighbors will be minimal. Therefore, we mark such nodes as *Dead Nodes* and do not include them in the subsequent iterations. For reasons of efficiency, for each node v that is not marked as a dead node, a check is performed every t_{num} iterations to identify whether the pagerank value of v changes beyond a pre-specified threshold (`threshold2` in Algorithm 16). If the change in the pagerank value of v is within the threshold, then v is marked as a dead node. Further, the computed pageranks using this optimization differ by less than $\pm$`threshold1`, indicating that the computed pagerank values are very close to their actual values. The value of `threshold1` can be adjusted by the application.

Putting Together Everything

Using the observations from Section 7.2.3–7.2.3, we now visualize the computation of pagerank as a three step process involving preprocessing, the actual computation of pagerank, and post-processing. Post-processing is required for the optimization corresponding to Sections 7.2.3 and 7.2.3. Algorithm 14 shows the pseudocode of our approach for pagerank.

Algorithm 14 is arranged as three steps with Algorithm 15, Algorithm 16 and Algorithm 17 forming the preprocessing, computation and post-processing steps respectively. To simplify our presentation, we consider the graph G as a weighted directed graph where each node has a weight. When using the Chain Node optimization, this weight allows us to set the contribution of the starting node of the path to the end node of the path as derived in Lemma 6.

In Line 2 of Algorithm 15, the function FindEquinodes returns an array `rep` which stores for each node the representative node of the

Algorithm 14 STIC-D(G)

procedure MAIN(GRAPH $G = (V, E)$)
  $\{\{C_{11}, C_{12} \cdot\cdot\}, \{C_{21}, C_{22} \cdot\cdot\} \cdots\}$ = SCC+TOPO(G)
  for $i = 0$ *to* $levels$ **do**
    for all $C_{ij} \in \{C_{i1}, C_{i2} \cdots\}$ in parallel **do**
      $(D_{ij}, initial, rep)$=Preprocess(C_{ij}, G)
      pr=Compute($D_{ij}, initial, rep$)
      Post-Process()
    end for
  end for
  return pr
end procedure

Algorithm 15 STIC-D(G)

procedure PREPROCESS($Graph\ C_i = (V_i, E_i)$,$Graph\ G = (V, E)$)
  rep = FindEquiNodes(C_i)
  if $f_1(rep, |V_i|, |E_i|) < thres1$ **then**
    $rep[u] = u, \forall u \in V$
  end if
  (D_i) = Compress(C_i)
  if $f_2(|V(D_i)|, |V_i|, |E_i|) < thres2$ **then**
    $D_i = C_i$
  end if
  $initial$ = Calinitial(D_i, G)
  return $(D_i, initial, rep)$
end procedure

Algorithm 16 STIC-D(G)

1: **procedure** COMPUTE($Graph\ G = (V, E)$,$Array\ initial$,$Array\ rep$)
2: $error = \infty$, $iterations = 0, adead = true$
3: $prev[u] = \frac{1}{n}, dead[u] = \frac{1}{n}, \forall u \in V$
4: **while** $error > threshold1$ **do**
5: **for all** $u \in V_{nc}\ s.t\ rep[u] = u, dead[u] > 0$ in parallel **do**
6: $pr[u] = initial[u]$
7: **for all** $v \in V$ such that $(v, u) \in E$ **do**
8: **if** $v \in V_c\ and\ rep[v] = v$ **then**
9: $pr[v] = initial[v] + \frac{prev[rep[red[v]]]}{outdeg[red[v]]} * weight[v] + \frac{1-d-weight[v]}{n}$
10: **end if**
11: $pr[u] = pr[u] + \frac{prev[rep[v]]}{outdeg[v]}$
12: **end for**
13: **end for**
14: $iterations = iterations + 1$
15: **if** $iterations\%tnum = 0\ \&\ adead = true$ **then**
16: Markdead($pr, dead, threshold2, adead$)
17: **end if**
18: **for all** $u \in V_{nc}$ with $rep[u] = u, dead[u] > 0$ **do**
19: $error = max(error, abs(prev[u] - pr[u]))$
20: $prev[u] = pr[u]$
21: **end for**
22: **end while**
23: return pr
24: **end procedure**

identical node set of which it is a part of. The function f_1 in Line 3 of Algorithm 15 calculates the percentage of nodes identified as identical nodes. If the percentage of identical nodes is below `thres1`, this optimization is not included.

In Line 6 of Algorithm 15, the function Compress removes all nodes of in-degree and out-degree one and their incident edges and fills the `red` array which stores for the chain nodes the alias node from which they should calculate the pagerank in the post-processing phase. The decision whether to include this optimization is done based on the function f_2 (Line 7) and the threshold `thres2`.

The function Calinitial (Line 10) sets the initial pagerank values of nodes in an SCC by considering the pagerank of endpoints of incoming cross edges whose pagerank converged already.

In Algorithm 16, V_c and V_{nc} refers to chain and non-chain nodes respectively. Lines 15, 16 of Algorithm 16 check for dead nodes by calling function MarkDead every `tnum` iterations. The function Markdead performs the check by comparing the change in pagerank in the previous `tnum` iterations against `threshold2`. The array dead stores a negative value if the node is dead or stores the pagerank that is at most `tnum` iterations old. The function Markdead also extrapolates the total number of dead nodes from the number of dead nodes seen till now and sets the value of `adead` as false if it less than 8% of n.

Implementation Details

In this section, we describe the implementation details of our algorithm.

We have distributed work between threads while iterating on components on same level in parallel by setting a threshold value of 10^5. So, if a component has more than 10^5 edges it would be worked on by all the threads. The work distribution is same for both the baseline algorithm and our algorithm.

All our preprocessing steps are using standard sequential algorithms. We use Kosaraju's algorithm (cf. [46]) for finding the strongly connected components, and BFS based algorithm for a topological or-

dering of the block graph. In FindEquinodes, we only consider nodes with in-degree 1 and 2 as these nodes account for a majority of the nodes that are identical for the purposes of the pagerank computation.

The accuracy of the baseline implementation and that of Algorithm 14 is controlled by the value of the variable `threshold1`. In our implementation, we set the value of `threshold1` to be 10^{-10}. In Algorithm 15, the values of the various thresholds are set as follows. The value of `thres1` and `thres2` are set at 7% and 15% of the size of the graph respectively by following our empirical observations. In Algorithm 16, the variable `tnum` was set to be 22 and `threshold2` is set $\frac{\texttt{threshold1} * \texttt{tnum}}{2 * n}$.

Algorithm 17 STIC-D(G)

1: **procedure** PostProcess($Graph\ G = (V, E)$,$Array\ initial$,$Array\ rep$)
2: **for all** $u \in V_c$ such that $rep[u] = u$ in parallel **do**
3: $pr[u] = initial[u] + \frac{prev[rep[red[u]]]}{outdeg[red[u]]} * weight[u] + \frac{1-d-weight[u]}{n}$
4: **end for**
5: **for all** $u \in V$ such that $rep[u] \neq u$ in parallel **do**
6: $pr[u] = pr[rep[u]]$
7: **end for**
8: **end procedure**

7.2.4 Experimental Results

In this section we describe our experimental platform, the dataset we use, and analyze the results of our algorithm.

Platform

We use an Intel i7 980x processor with 8 GB memory as our experimental platform. The 980x is based on the Intel Westmere micro-

architecture and has six cores with each core running at 3.4 GHz. With active SMT (hyper-threading), the i7 980x can support twelve logical threads. The memory hierarchy of the i7 980x has a three level cache system with an L1 cache of size 64 KB per core, an L2 cache of size 256 KB per core, and a shared L3 cache of size 12 MB.

Dataset

We experiment on four classes of real-world graph datasets namely web graphs, social networks, collaboration networks, and road networks. The graphs are part of standard datasets [3,4], and important characteristics of the graphs are listed in Table 7.2. In Table 7.2, the column SCC's indicates the number of strongly connected components in the graph, and the column Levels indicates the number of levels in the block graph of the input graph.

Results

We first present the overall speedup results and a study of the time spent across the phases of Algorithm 14. We then proceed to analyze each of the four graph classes.

Overall Speedup We calculate the speedup of Algorithm 14 as the ratio of the time taken by Algorithm 13 to that of Algorithm 14. Both the algorithms are run in a multi-threaded manner with 12 threads on the machine described in Section 7.2.4. Each experiment is run multiple times and the average speedup is used in reporting.

The results of the overall speedup are shown in Figure 7.4. In Figure 7.4, the phrase "baseline" refers to time taken by Algorithm 13. On the X-axis of Figure 7.4, the graphs are arranged according to their class as listed in Table 7.2. On an average, we get a speedup of 77% on web graphs, 28% on social graphs, 13% on collaboration networks, and 8% on road networks.

Graph name	Source	$\|V\|$	$\|E\|$	SCCs	Levels
Web Graphs					
web-Stanford	[89]	281903	2312497	29914	141
web-BerkStan	[89]	685230	7600595	109406	114
indochina-2004	[122, 123]	7414866	194109311	1749052	524
web-Google	[89]	916428	5105039	412479	34
web-Notredame	[7]	325729	1497134	203609	18
Social Networks					
soc-Slashdot0811	[89]	77360	905468	6724	4
soc-Slashdot0902	[89]	82168	948464	10559	3
soc-Epinions1	[130]	75888	508837	42185	10
soc-LiveJournal1	[11, 89]	4847571	68993773	971232	24
Collaboration Networks					
co-AuthorsDBLP	[66]	299067	977676	299067	7
co-Authorsciteseer	[66]	227320	814134	30322	7
coPapersCiteSeer	[66]	434102	16036720	6372	7
coPapersDBLP	[66]	540486	15245729	10244	6
Road Networks					
italy_osm	[4]	6686493	7013978	1470097	2692
great-britain_osm	[4]	7733822	8156517	2444901	725
germany_osm	[4]	11548845	12369181	2466406	534
asia_osm	[4]	11950757	12711603	3511783	10943

Table 7.2: List of graphs that we use in our experiments.

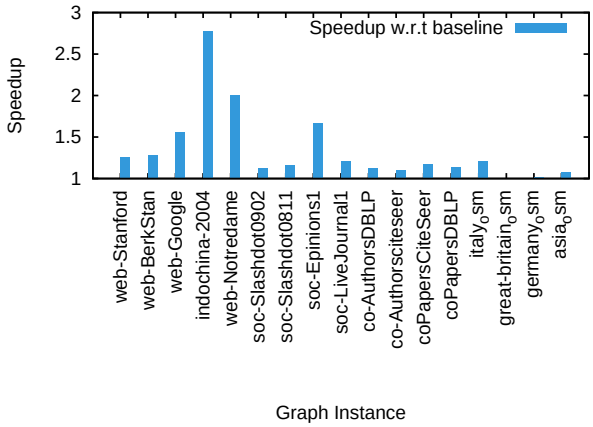


Figure 7.4: Figure shows the speedup of algorithm on graphs listed in Table 7.2.

Profile across Phases To understand the differences in the speedup that our algorithm achieves on various graphs, we first start by studying the relative time spent by our algorithm in the preprocessing and the actual computation phases. The post-processing time is not shown as it is noted to be negligible and under 1% of the total time. The reason behind this is that all our techniques are able to simultaneously compute the pagerank of the nodes removed in preprocessing.

The preprocessing time is on an average 25% on web graphs, 11% on social networks, 17% on collaboration networks and 30% on road networks. About half of this spent on finding the SCCs and a topological order among SCCs. To identify identical nodes, the time spent is usually under 2%, except for web graphs where it takes nearly 12%. Identifying identical nodes in web graphs takes more time as there is usually a large percentage of identical nodes. We observe that in road networks, both the SCC and Topological Ordering and the preprocessing time in the Chain nodes optimization step is 14%. This is because these graphs have a lot of levels as shown in Table 7.2.

In general, the high processing time of our algorithm can be at-

tributed to the fact that all the preprocessing algorithms are running as sequential algorithms. The effectiveness of our technique can be seen by comparing the time taken by our actual computation with the computation time taken by Algorithm 13. This ratio is on an average 2.4 on web graphs, 1.45 on social networks, 1.3 on collaboration networks, and 1.43 on road networks.

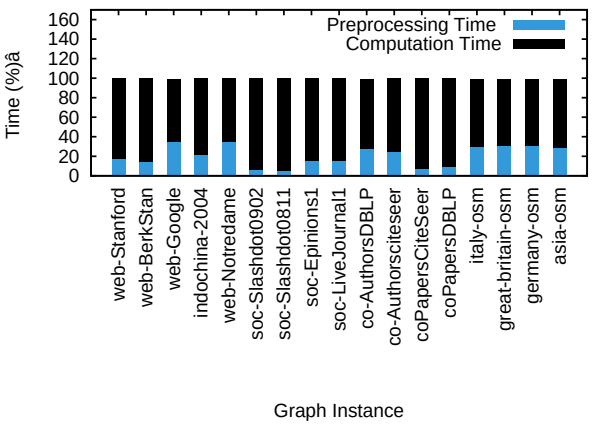


Figure 7.5: Profile of time taken across various steps of Algorithm 14.

Analysis of Graph Classes In this section, we discuss in detail the effect of our techniques on the graph classes from our dataset.

Web Graphs We experimented with five web graphs. We note that these graphs have a large number (15% on average) of cross edges, i.e., edges that have end points in two SCCs. These graphs also have a large number of SCCs indicating that our SCC and Topological Ordering optimization should actually improve the performance of our algorithm by a good factor. Further, we notice that most of the web graphs do not have long directed paths. Therefore, we do not apply the chain optimization step on web graphs. On the other

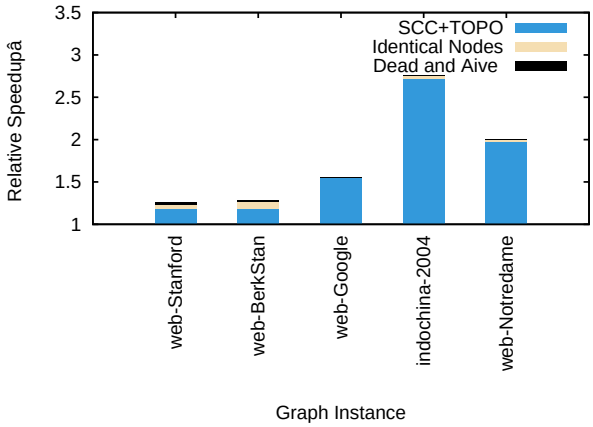


Figure 7.6: Relative Speedup of optimizations on Web graphs.

hand, these graphs contain a significant portion (18% on average) of identical nodes. So, we apply this optimization.

The impact of each of the optimizations applied on web graphs is shown in Figure 7.6. The dead and live optimization works only for the graphs web-Stanford, web-BerkStan and Indochina-2004. On an average, we get a 3% speedup with this optimization on these graphs.

Social Networks In the class of social networks, we consider four graphs. These graphs on average have about 10% of cross edges but the SCCs themselves are arranged in fewer levels. These graphs also do not have long directed paths, so this optimization is not considered for social networks. Some of these graphs may not also have a good number of identical nodes, so this optimization can be applied only in specific cases. Figure 7.7 shows the impact of the optimizations applied on social networks.

Collaboration Networks We now consider collaboration networks. These graphs are an example of graphs that do not benefit from our optimizations for several reasons. These graphs have very few cross

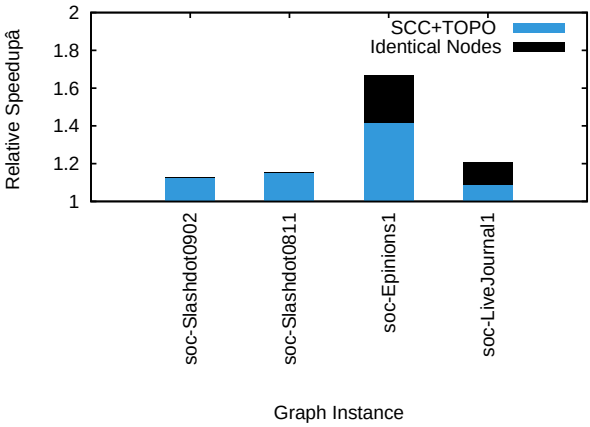


Figure 7.7: Relative Speedup of optimizations on social networks.

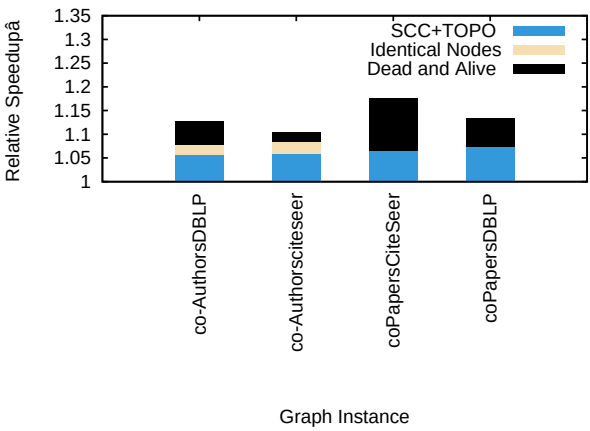


Figure 7.8: Relative Speedup of optimizations on Collaboration Networks.

edges, very few levels, and do not have long directed paths. These graphs however benefit from keeping nodes dead. The overall speedup on these graphs is quite small as several of our optimizations are

not applicable in this class of graphs. Figure 7.8 shows the relative speedup of techniques on collaboration networks.

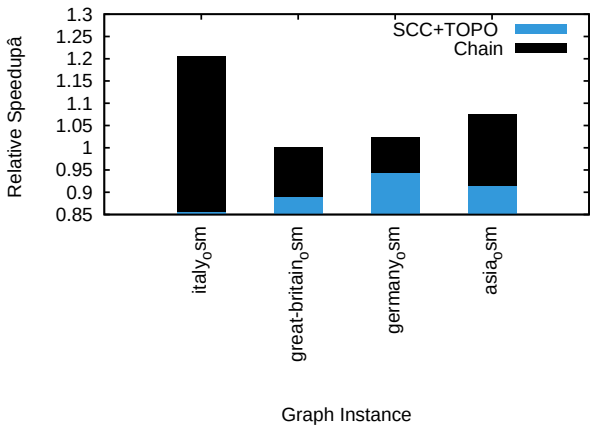


Figure 7.9: Relative Speedup of optimizations on road networks.

Road Networks Finally, we move to analyzing road networks. These networks by virtue of their application domain, have a large number of levels and also a large number of SCCs. This means that our SCC and Topological Ordering optimization incurs a huge preprocessing overhead. In fact, just applying this optimization on these graphs actually results in a 10% average slowdown compared to the baseline algorithm.

These graphs also have a small number of identical nodes. So, the advantage gained by identifying the identical nodes will be offset by the preprocessing time required. However, these graphs have a large number of long directed paths. So, the chain node optimization technique works well on these graphs. Figure 7.9 shows the relative speedup of techniques on collaboration network graphs.

7.3 Betweenness-Centrality

The betweenness-centrality measure of a node indicates the relative importance of each node v of G by considering the number of shortest path passing through v. Betweenness-centrality as a measure finds applications in several areas of graph analytics such as those in social networks [107] and biological graphs [103]. One property that we focus in this section is that real-world graphs, being sparse in nature, tend to have a reasonably good number of nodes of degree at most two. Removing such nodes can be helpful in problems such as computing the betweenness-centrality of a given graph. However, care is required to ensure that the centrality values of nodes removed from the graph can be computed efficiently using the corresponding values at nodes of degree greater than two. To this end, we use the ear decomposition of a graph that helps us to systematically identify and bookkeep details of nodes of degree at most two. An ear decomposition of a graph is a partitioning of a graph into simple paths that overlap only at end points. See Figure 3.1 for an example.

We start with biconnected graphs and show that one can use an ear decomposition of a biconnected graph to improve the practical efficiency of computing the betweenness-centrality values of nodes in the graph. In particular, we use the ear decomposition of a graph to remove nodes of degree two, perform the bulk of the computation with respect to the remaining graph, and use a post-processing step where we compute the betweenness-centrality of nodes that were removed and also update the betweenness-centrality values of the nodes remaining.

Having a post-processing step in our algorithm means that unlike existing approaches [111, 112, 133], we need to retain the results from the processing phase to the post-processing phase. Doing so would require a large amount of space that far exceeds the space available on current generation GPUs. To address this problem, we interleave execution of the processing and the post-processing steps along with identifying redundant information that need not be stored, and an orchestration of nodes in the processing step. Using these techniques

our implementation outperforms existing approaches by a factor of 1.51x speedup on a collection real-world graphs from the UFL dataset [4].

Finally, using ideas from Sariyuce et al. [134] and Wang et al. [155] we show how to extend our approach to graphs that are not biconnected. Our approach achieves a speedup of 1.9x on a collection of real world graphs from [4]. We note that any improvements to GPU-based algorithms for breadth first search (BFS) can improve our results too.

7.3.1 Related Work

Decomposition of graphs into subgraphs is a technique in parallel graph algorithms that is gaining significant research attention in recent years. Metis [98] decompose a graph into a given number k of subgraphs such that the number of edges that cross a partition is minimized. This decomposition is being used parallel graph algorithms lately as a subroutine [56]. However, for path-based problems such as shortest paths and betweenness-centrality, decomposition via Metis may not be ideal due to the presence of cycles that go across the partitions induced by a Metis decomposition. These cycles mean that computing shortest paths between nodes in two different partitions is non-trivial.

Parallel algorithms in the PRAM style for obtaining an ear decomposition are presented by Ramachandran [126] along with applications to problems such as planarity testing and triconnectivity. Bader et al. [13] show the results of implementing algorithm [126] on NPACI Sun E10K machines.

Computing the betweenness-centrality values of nodes in a graph has seen lot of interest in recent years in parallel computing research. Most papers [111,112,133,134] use the algorithmic approach of Brandes [26]. Sariyuce et al. [134] use a biconnected component decomposition of a graph, compute the betweenness-centrality of a node local to its biconnected component, followed by a post-processing step for computing the betweenness-centrality values with respect to the en-

tire graph. In a sequential computing model, they show that such a technique results in a speedup of 3.8x compared to Brandes [26]. Wang et al. [155] provided optimizations similar to [134] and achieve considerable speedup over existing multicore implementations including that of the Ligra framework [141].

Optimizing BFS on a GPU focusing on applications such as betweenness-centrality in unweighted graphs is studied by Sariyuce et al. [133], and by Bader and Mc. Laughlin [111, 112, 156]. Bader and Mc. Laughlin [111] improved the BFS implementation of Jia et al. [92] and Shi and Zhang [136]. Bader and McLaughlin perform a BFS with each node of the graph as the source of the BFS. Each such BFS is run in parallel on a SMX (SM) of the GPU. The information obtained from the BFS from v is used to compute the betweenness-centrality of v. Improvements to GPU BFS shown by Bader and Mc. Laughlin [112] lead to direct improvements computing betweenness-centrality values over their results from [111].

7.4 Computing Betweenness-Centrality

In a graph $G = (V, E)$ the betweenness-centrality (BC) of a node $v \in V$ is a measure of the number of shortest paths that pass through v. Equation 7.5 (see also [26]) captures the above formally where σ_{st} denotes number of shortest paths between s and t, and $\sigma_{st}(v)$ is number of shortest paths between s and t that pass through v.

$$bc(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{7.5}$$

Before we present our algorithmic approach for computing the betweenness-centrality values of nodes in a given graph, we briefly review the algorithm of Brandes [26] that has been the algorithm of choice [111, 112, 133, 134] for computing betweenness-centrality in parallel.

7.4.1 Brandes Algorithm

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

7.4.2 The Approach of Bader et al. [111, 112]

The main idea of the works of Bader and Mc. Laughlin [111, 112] is to target GPU specific optimizations to perform multiple BFS operations, one from each node of the graph as a source node. Bader and Mc. Laughlin [111] use SMX level parallelism and batch the n BFS operations on the SMXs. Other techniques introduced in [111] include memory usage optimization, reduction in atomic operations, and load balancing based on the structure of the graph.

Bader and Mc. Laughlin [112] introduce further optimizations such as warp level parallelism and warp-level load balancing using dynamic scheduling. These optimizations result in an improved BFS performance and a direct improvement over [111] for computing the betweenness-centrality values of nodes in unweighted graphs.

7.4.3 Our Approach

Our algorithmic approach to compute the betweenness-centrality values of nodes in a sparse graph uses the following outline. We start by considering graphs that are biconnected. Such a graph will have an ear decomposition as shown by [126, Lemma 2.1]. In a preprocessing step, we obtain an ear decomposition of the graph. Using the ear decomposition to perform the necessary book-keeping, we remove nodes of degree two from the graph. The (partial) betweenness-centrality values of the remaining nodes according to the original graph are computed by using standard algorithmic techniques. In a post-processing step we compute the betweenness-centrality values for nodes removed during the preprocessing step and also update the values for the nodes that remain after preprocessing. Finally, we show how to extend our approach to graphs that are not biconnected.

An illustration of our approach for biconnected graphs is shown in Figure 7.10. In the following, we present a pseudocode of our algorithm as Algorithm 18 and provide details of the steps in our algorithm in Sections 7.4.3–7.4.3.

Algorithm 18 Algorithm BetweennessCentrality(G)

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

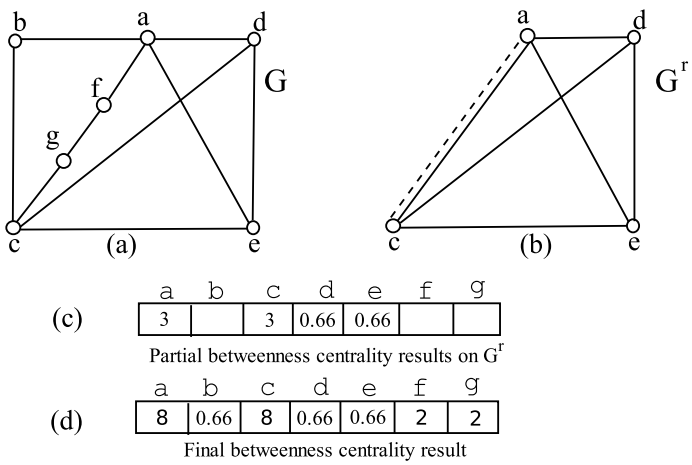


Figure 7.10: Figure (a) shows the input graph G with an ear decomposition where the number on the edges indicates the ear they belong to. Figure (b) shows the reduced graph G^r. A parallel edge in the reduced graph between nodes a and c is shown as a dotted line for illustration purposes. Figure (c) shows the *partial* betweenness-centrality values of nodes in G^r computed in the processing phase of our algorithm. Figure (d) shows the final betweenness-centrality values for *all* nodes obtained at the end of the post-processing phase.

Preprocessing

Let $G = (V, E)$ be a biconnected graph. The REDUCE(G) routine starts by obtaining an ear decomposition of G using Algorithm 2. In such a decomposition, nodes of degree two, except possibly those on ear P_0, appear on exactly one ear. The resulting reduced graph $G^r = (V^r, E^r)$ is defined as follows. The nodes of G^r are the nodes of G that have a degree at least three. Two nodes v and w in G^r are neighbors if and only if v and w belong to a common ear P of G and have no nodes of degree three or more between them on the ear P. Figure 7.10(a)–(d) shows an example. For a node x of degree two on ear $P = (a_1 a_2 \cdots a_k)$ in G, we define functions *left()* and *right()* of x in G^r, denoted Left(x) and Right(x), as the nodes of degree at least three on P that are closest to x towards a_1 and a_k respectively. For instance, in the example in Figure 7.10(b), Left(f) = a and Right(f) = c. (The terms Left and Right are only mnemonic in nature.)

Notice that during the construction of the reduced graph, there could be multiple edges between nodes in the reduced graph. In this case, since we are interested in shortest paths, we retain the edge with the shortest length and discard the remaining edges. An example shown in Figure 7.10(b) for purposes of illustration.

Processing

In the processing phase, we compute the betweenness-centrality values of nodes in G^r. We use the BFS routine from [112] in our processing step and perform a BFS in G from each node in G^r as the source node. Note that the BFS has to be done in the graph G and not the graph G^r so as to correctly capture the impact of the multiple edges in G^r on the betweenness-centrality values of nodes in G^r. Along with each BFS, we perform the forward propagation stage and the accumulation stage of the algorithm of Brandes. In the FWDSTAGE routine, for each source node v, arrays S_v, D_v and σ_v are recorded as the result of a BFS with v as the source node as is done in the forward propagation phase. As G is unweighted, P_v is not computed or stored explicitly and is simulated using the D_v array.

In the ACCUMULATE_PARTIALBC routine, we use the arrays S_v, D_v, and σ_v for each $v \in G^r$ computed in the forward stage to compute betweenness-centrality values of nodes in G^r. However, these computed betweenness-centrality values of nodes in G^r can change in the post-processing step as the accumulation stage of nodes in $G \setminus G^r$ is performed. Therefore, we call these values as the *partial* betweenness-centrality values as shown in Figure 7.10(c).

Post-processing

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

The routine SIM_FWDSTAGE works as follows. Let x be a node in $G \setminus G^r$ with $\text{Left}(x) = \ell_x$ and $\text{Right}(x) = r_x$. We simulate the actions of executing the forward stage of Brandes algorithm [26] for node x as follows. We need to obtain arrays S_x, D_x and σ_x as part of the forward stage. For obtaining the array S_x, we start by merging sequences S_{ℓ_x} and S_{r_x} as follows. Let v and w denote the first node in S_{ℓ_x} and S_{r_x} respectively. We now compare $D_{\ell_x}(x) + D_{\ell_x}(v)$ and $D_{r_x}(x) + D_{r_x}(w)$. If the former is smaller, then we add v to S_x. Otherwise, we add w to S_x. Nodes v and w are incremented to be the next node in S_{ℓ_x} and S_{r_x} depending on which of v or w is added to S_x in this step. (Alike the procedure Merge in Merge Sort [48]). In a similar fashion, we can also obtain arrays D_x, and σ_x from the respective arrays of nodes ℓ_x and r_x.

Once these arrays are obtained for node x, the accumulation stage (i.e SIM_ACCUMULATION routine) of Brandes algorithm can be simulated as described in Section 7.4.1. During this stage, the partial betweenness-centrality values of nodes in G^r will be updated as needed. At the end of the post-processing phase, we therefore have the final betweenness-centrality values of all nodes in G as shown in Figure 7.10(d).

7.4.4 Implementation Details

In this section, we mention some of the important implementation details of our algorithm. Since we use the implementation from [112] in Phase II of Algorithm 18, we stand to benefit from all the GPU specific optimizations that are included in the implementation of [112].

The preprocessing step in our algorithm necessitates a post-processing step unlike other algorithms [111,112,156]. To run the post-processing step, as described in our algorithm, we need $O(n)$ Bytes of information per node of G^r amounting to $O(n \cdot n^r)$ Bytes where $n^r = |V(G^r)|$. For even moderate value of n, this amount of space far exceeds the amount of space available on current generation GPUs.

To alleviate this problem, we run the processing and the post-processing steps in an interleaved manner. Doing so naively will not result in any improvement in the space utilization. However, we introduce two novel techniques in our implementation that help us in the following way. Firstly, in Section 7.4.4, we identify information computed in the processing phase that is not needed in the post-processing phase. Secondly, in Section 7.4.4, we orchestrate the nodes in G^r as to when the processing step corresponding to a node is performed and how long the information thus generated has to be kept in the memory. This allows us to reuse the limited space effectively.

Classifying Nodes in G^r

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

Figure 7.11(b) illustrates the idea of free and active nodes. In

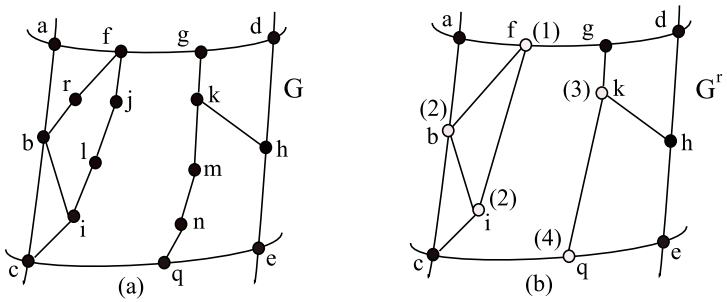


Figure 7.11: Figure (a) shows the input graph G. Figure (b) shows the reduced graph G^r. In Figure (b), nodes filled in black color indicate *free* nodes and the other nodes are *active* nodes. The numbers on the active nodes in Figure (b) indicate the BFS level number with f and k as the source nodes.

Table 7.2, column 7 shows the number of free and active vertices in the largest biconnected component of the corresponding graph. For the graphs listed in Table 7.2, the average number of active vertices is under 35% indicating that the classification into active and free vertices is a useful technique to reduce the storage required by our approach.

Orchestrating Nodes in the Processing Phase

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

To this end, let F denote the subgraph of G^r induced by V^a. We now find the connected components of F using standard parallel algorithms such as those presented in [144]. We also order the connected components of F in some order, say $F_1, F_2, \cdots$. Consider a connected component H of F and define $\text{Dep}(H) := \{x | x \in G \setminus G^r$,

Left$(x) \in V(H)$ or Right$(x) \in V(H)\}$. Once nodes in Dep(H) finish their post-processing, the information with respect to nodes in H is no longer required to be in memory and the associated space can be reused.

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

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

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

Columns 8 and 9 of Table 7.2 show the number of vertices in H and the maximum number of vertices in any one level of a BFS in H for the corresponding graph. As can be seen, the maximum number of vertices in any one level of a BFS in H is quite small compared to the number of vertices in the largest biconnected component. Recall that the storage required by our algorithm is at most the number of vertices in the largest biconnected component of the graph multiplied by twice the maximum number of vertices in any level of a BFS on

H. This indicates that the two techniques presented in this section make our algorithm scalable for large graphs also even on a single GPU with limited memory.

In our implementation, in the post-processing phase, recall that for a node $x \in G \setminus G^r$, we compute the arrays S_x using the arrays S_{ℓ_x} and S_{r_x} where $\ell_x = \text{Left}(x)$ and $r_x = \text{Right}(x)$. To do so, we use one SMX for each node x. Threads within an SMX compute the array S_x as explained in Section 7.4.3. A similar approach is followed for computing the arrays D_x, and σ_x.

7.4.5 Results

Platform

We use the GPU described in Section 7.2.4. For comparison studies with respect to libraries running on multicore CPUs, we use an Intel(R) Xeon(R) E5-2650 CPU with 128 GB RAM and a memory bandwidth of 68 GB/s for our experiments. The E5-2650 CPU is a dual processor where each processor has 10 cores and each core can process two threads using hyper threading. Each core operates at 2.34 GHz which can be boosted to 3 GHz using turbo boost technology. The E5-2650 CPU has 64 KB L1 cache per core, 256 KB L2 cache per core and a shared 25 MB L3 cache.

Datasets

We experiment with graphs from the dataset of sparse graphs from the University of Florida dataset [4]. Since we require the graph to be biconnected, we run algorithms on the largest biconnected component of the graphs listed Table 7.3. Since graphs in the dataset from Table 7.3 have a large biconnected component that spans more than 80% of the edges, as indicated by numbers shown in column labeled **Largest BCC**, the size of the graph that we run our algorithm is not significantly compromised.

Graph name	$\|V\|$	$\|E\|$	**Largest BCC**				
			$\|V\|$	$\|E\|$	**%Deg=2**	**Active**	**|V(H)**
roadNet-CA	2.0 M	2.7 M	1.57 M	2.34 M	24.0	424 K	41
roadNet-TX	1.4 M	1.9 M	1.05 M	1.57 M	25.0	276 K	57
soc-Epinions1	76 K	508 K	36 K	365 K	27	8.9 K	5.5 K
patents_main	241 K	560 K	151 K	474 K	26.1	44 K	3.8 K
coAuthorsDBLP	299 K	977 K	198 K	818 K	15.4	36.6 K	4.7 K
soc-Slashdot0902	82 K	474 K	51 K	47 K	23.0	11 K	6.4 K
caidaRouterLevel	192 K	609 K	132K	541 K	27.3	32.7 K	183
scircuit	171 K	479 K	135 K	335 K	13.5	16 K	58
soc-sign-epinions	131 K	841 K	58 K	642 K	27.7	12.6 K	8.8 K
p2p-Gnutella31	62 K	147 K	33 K	119 K	27.7	10.4 K	5.4 K

Table 7.3: List of graphs that we use in our experiments. In this table, the number of nodes and the number of edges are rounded to the nearest thousand (K) or the nearest million (M). The last column indicates the percentage of nodes that are eliminated from the largest BCC during the preprocessing step.

Results

We compare the results of our algorithm, labeled as "OUR" in the rest of this section, with a wide range of algorithms on GPUs and multi-core CPUs. The algorithms are listed in the following.

GPU Based Algorithms

- **Bader and Mc. Laughlin [112]:** This work is currently one of the fastest for computing betweenness-centrality on a GPU. We use the software from the authors of [112] in our comparison. This result is labeled "BM15" in the rest of the section.

- **Gunrock Library [156]:** Gunrock is a GPU based library for graph algorithms, which contains a routine for computing betweenness-centrality. We use the software from [156] and label this result as "GUNROCK" in the rest of this section.

Multi-core CPU Based Algorithms

- **APGRE [155]:** Wang et al. [155] extend the work of Sariyuce et al. [134] to multi-core CPUs. We implement the algorithm of [155] on the CPU described in Section 7.4.5 with a thread per core and label this result as "APGRE".

- **Ligra [141]:** The Ligra library [141] consists of routines for graph algorithms and runs on multicore CPUs. Results of the Ligra library are labeled "LIGRA" in our plots.

On the graphs from Table 7.3, the overall time taken by the above algorithms on the largest biconnected component is plotted in Figure 7.12(a) for multi-core CPUs and Figure 7.12(b) for GPUs[1]. The Y-axis of Figure 7.12(a) is on a logarithmic scale. The secondary Y axis

[1]Note that in [112], absolute time taken as mentioned are normalized to 8192 iterations. Similarly, the timings shown in [156] are normalized to one iteration.

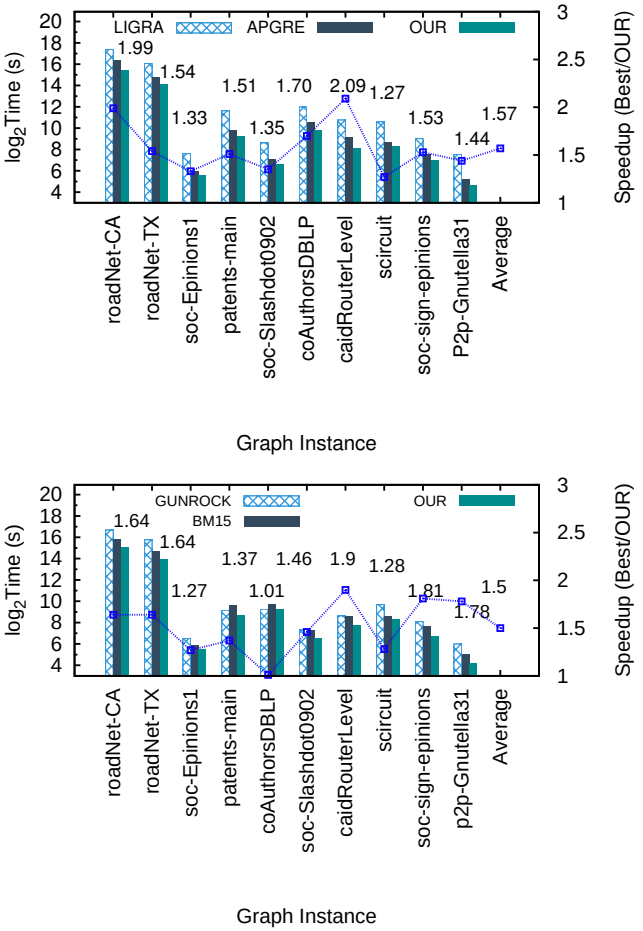


Figure 7.12: Comparing the overall performance improvement of Algorithm 18 on the largest BCC of the graphs listed in Table 7.3. Figure (a) compares our algorithm with respect to [155] and [141] on multicore CPUs. Figure (b) compares our algorithhm with respect to BM15 [112] and GUNROCK [156] on a GPU. The last instance on the X-axis of the figures shows the average speedup of our algorithm over the best of the other algorithms used in the comaprison.

of Figure 7.12(a) shows the speedup of our algorithm over the **best** of above mentioned baseline algorithms.

The absolute run time of the various algorithms considered is shown in Table 7.4 under the column labeled "Largest BCC".

Graph Name	**Multi-core CPU**			**GPU**		
	OUR	**APGRE**	**LIGRA**	**OUR**	**BM15**	**GUNR**
roadNet-CA	42946	85462	171448.70	33969	55030	104237
roadNet-TX	18013	27652	68978.50	15303	25097	80520
soc-Epinions1	46.15	61.37	196.20	43.25	56.11	100.76
patents_main	587	885	3181.90	416.5	755.54	578.91
coAuthorsDBLP	871	1484	4047.68	602.81	814.99	661.65
soc-Slashdot0902	97.57	132.09	395.05	90.72	153.03	154.56
caidaRouterLevel	271.63	566.5	1742.83	207.03	381.29	428.6
scircuit	310	393	1552.13	304.11	391.01	1646
soc-sign-epinions	125.12	190.93	520.11	103.65	187.14	273.14
p2p-Gnutella31	24.6	35.49	187.09	18.42	32.87	63.34

Table 7.4: This table shows absolute time in seconds of OUR algorithm, labeled OUR, BM15, GUNROCK and APGRE on the largest BCC of the graphs listed in Table 7.3. Times above a thousand seconds are rounded to the nearest integer.

The throughput of an algorithm for computing the betweenness-centrality on a graph G of n nodes and m edges is measured as $\frac{n \cdot m}{t}$ Traversed Edges Per Second (TEPS) where t is the time taken in seconds by the algorithm. The quantity MTEPS refers to Million TEPS. The throughput achieved by the algorithms under study is shown in Figure 7.13(a) for multi-core CPUs and Figure 7.13(b) for GPU based implementations. As can be seen from Figure 7.13, our algorithm achieves a higher MTEPS compared to the existing algorithms both on multi-core CPUs and GPUs.

Performance on Synthetic Datasets

To understand how the number of nodes eliminated in the preprocessing step of Algorithm 18 can impact the speedup achieved, we construct a synthetic graph of n nodes with average degree d as follows. A cycle graph on n nodes ensures that the graph will have only one biconnected component. On this cycle graph, we mark a $t\%$ of nodes as nodes that will have a degree of two. The degree of the rest of the unmarked nodes is increased by adding edges to pairs of nodes chosen uniformly at random. We ensure that each unmarked node has degree at least three using standard techniques from randomized algorithm [117]. An example of the graphs generated in this fashion is shown in Figure 7.14.

We study the results of our approach on synthetic graphs of size ranging from 100 K nodes to 300 K nodes with an average degree of 20 to 30. We vary the percentage of nodes that can be removed from the reduction step from 10% to 50%. We study the speedup of Algorithm 18 with respect to that of Bader and Mc. Laughlin [112]. As shown in Figure 7.15, for a fixed n, as the percentage of nodes of degree two increases, the speedup achieved by our algorithm also increases.

7.4.6 Extending our Approach to General Graphs

So far, we have assumed that our input graph is biconnected. In general, however, most real-world graphs are not biconnected. In this section, we briefly show how to extend our techniques to non-biconnected graphs. We start by reviewing the BADIOS framework of Sariyuce et al. [134] and the APGRE framework of Wang et al. [155] that we use in our solution.

The BADIOS and the APGRE Framework

The main idea of the BADIOS framework [134], called as APGRE framework in [155], is to decompose a graph G into its biconnected

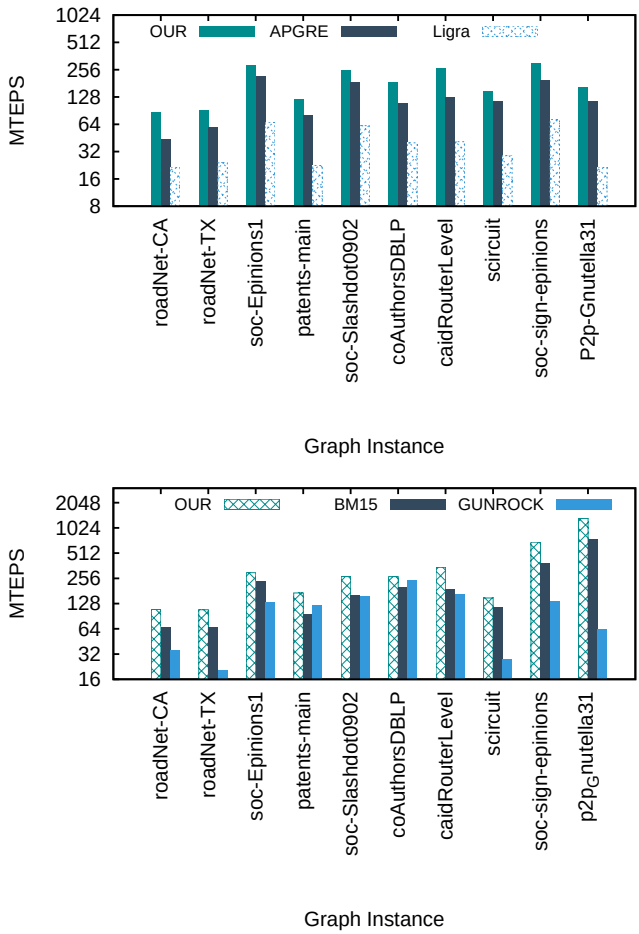


Figure 7.13: Comparing the MTEPS achieved by Algorithm 18 on the largest BCC of the graphs listed in Table 7.3 with respect to that of APGRE [155] and LIGRA [141] on multi-core CPUs (Figure (a), and with respect to BM15 [112], and GUNROCK [156] on a GPU.

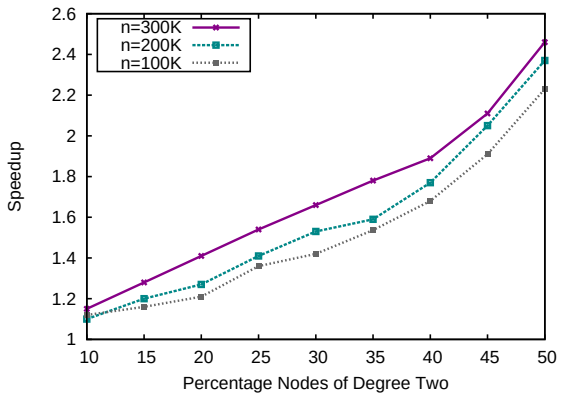


Figure 7.14: A synthetic graph with a specific number of nodes, marked shaded in the picture, of degree two.

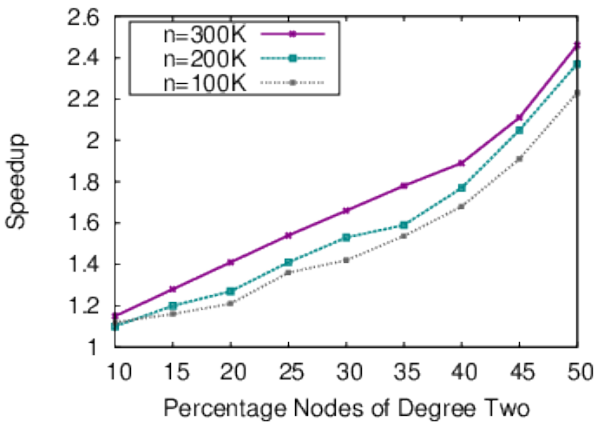


Figure 7.15: The relative performance obtained by Algorithm 18 over [112] on synthetic graphs.

components (BCCs) and use Brandes algorithm [26] on the individual biconnected components.

In the algorithm of Sariyuce et al. [134], each BCC has a copy of the articulation point, called as an *alias* node, which connects it to other neighbouring BCC's in the original graph G. A reachability metric is defined for alias nodes as follows. Consider the ith BCC $G_i = (V_i, E_i)$ of G let $v' \in V_i$ be an alias node. Consider any node $u \in V_i$ that is different from v'. The reachability metric for v', denoted reach(v'), is set to the number of nodes $x \in V \setminus V_i$ such that the path between u and x passes through v'. (Note that the choice of u is immaterial in the above.) Reach values for every alias node is computed by a leaf-to-root traversal of the block tree T as described by Puzis et al. [125]. The reachability metric is useful in extending the betweenness-centrality values computed on the BCCs of G to the entire graph.

Our Algorithm for General Graphs

To use the framework of BADIOS [134] or APGRE [155], we start by decomposing the input graph G into its biconnected components, $G_1, G_2, \cdots,$. Since each of these components, G_i, $i \geq 1$, are biconnected, they possess an ear decomposition. So, each G_i can be taken as input to Algorithm 18 to compute the betweenness-centrality values of nodes in G_i. At this point, once we have the reachability values for alias nodes as is done in [134, 155], it will be possible to extend the betweenness-centrality values of nodes with respect to each G_i to the entire G. A brief pseudocode is presented in Algorithm 19.

As can be seen, Algorithm 19 has a two stage preprocessing and a two stage post-processing. The first stage of preprocessing decomposes G into its biconnected components, and the the second stage applies ear decomposition on each component. In the processing step, betweenness-centrality values with respect to each biconnected component is computed similar to the processing phase of Algorithm 18. Finally, we have two post-processing steps: first that is similar to the post-processing phase of Algorithm 18, and the second one similar to

Algorithm 19 Algorithm Betweenness Centrality(G)

```
1: /* Phase I : Preprocessing */
2: G_i = BCC((G)) /* Stage 1 */
3: G_i^r = REDUCE(G_i) /*Stage 2 */
4: /* Phase II : Processing */
5: for each i do in parallel do
6:     for each v in G^r do in parallel do
7:         (S_v, D_v, σ_v) = FWDSTAGE(v, G)
8:         ACCUMULATE_PARTIALBC(S_v, D_v, σ_v)
9:     end for
10: end for
11: /* Phase III : Post Processing */
12: /* Stage 1 */
13: for each i do in parallel do
14:     for each v ∈ G_i \ G_i^r do in parallel do
15:         l_x ← Left(v) , r_x ← Right(v)
16:         (S_v, D_v, σ_v) = SIM_FWDSTAGE(v, l_x, r_x)
17:         SIM_ACCUMULATION(v, l_x, r_x, S_v, D_v, σ_v)
18:         Update the BC values to the nodes in G^r
19:     end for
20: end for
21: /* Stage 2 */
22: for each v ∈ G_i do in parallel do
23:     if v = art_vertex then
24:         bc[v] = bc[v] + reachval[v] · (n_i − 1)
25:         Update the BC values to the nodes in G_i
26:     end if
27: end for
```

that of the corresponding step in [134, 155].

Results

We reconsider the graphs listed in Table 7.3 and apply our approach to compute betweenness-centrality. We use the experimental platform mentioned in Section 7.2.4. For performance comparison, we consider the algorithms listed in Section 7.4.5.

Figure 7.16(a) shows the time taken by our algorithm and the other algorithms on multi-core CPUS for the graphs listed in Table 7.3. Figure 7.16(a) shows the time taken by our algorithm and the other algorithms on GPUS for the graphs listed in Table 7.3. The numbers in Figure 7.16 show the speedup achieved by our algorithm compared to the **best** of the other two algorithms. The throughput of our algorithm as MTEPS is shown in Figure 7.17(a) and (b) along with the throughput achieved by the other algorithms on multi-core CPUs and GPUs respectively.

The absolute runtime is shown in Table 7.5 under the column labled "Entire Graph". The speedup achieved in the case of the entire graph is higher than the speedup achieved on the largest biconnected component of the corresponding graph as can be seen from Figures 7.16(a) and 7.12(a). The reason for this is that when using the algorithms from [112, 156], every BFS has to run on the entire graph. In our algorithm, and also that of [155], each BFS runs only local to a biconnected component.

7.5 Conclusions

In this chapter, we studied three different problems: Fisrt, we proposed the STIC-D framework to optimize the time taken to compute pagerank in graphs. The techniques in the framework proposed are based on exploiting the structures found in real-world graphs and are useful in *reducing* the pagerank computation time.

Finally, we used the ear decomposition of a graph and its application to finding the betweenness-centrality of nodes in a graph.

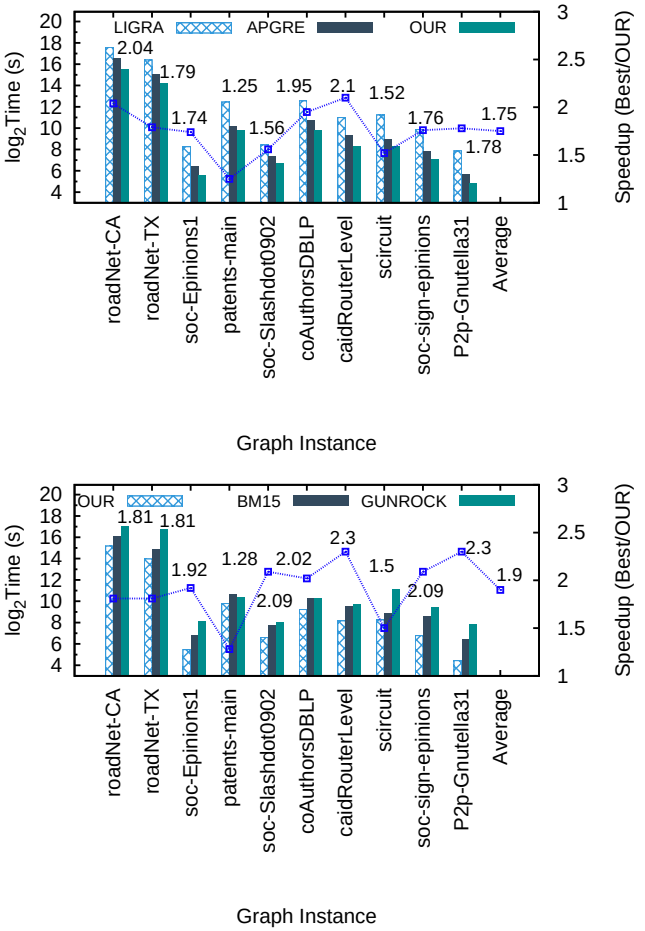


Figure 7.16: Comparing the overall performance improvement of Algorithm 19 on the entire graphs listed in Table 7.3. Figure (a) compares our algorithm with respect to [155] and [141] on multicore CPUs. Figure (b) compares our algorithhm with respect to BM15 [112] and GUNROCK [156] on a GPU. The last instance on the X-axis of the figures shows the average speedup of our algorithm over the best of the other algorithms used in the comaprison.

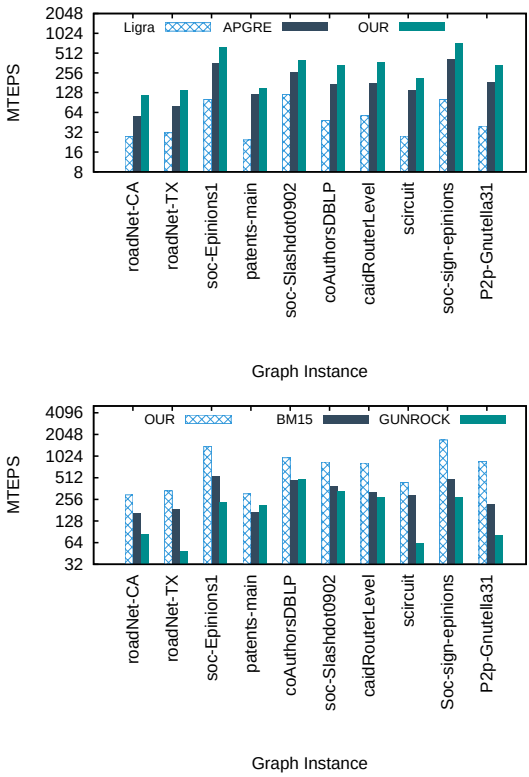


Figure 7.17: Comparing the MTEPS achieved by Algorithm 19 on the graphs listed in Table 7.3 with respect to that of APGRE [155] and LIGRA [141] on multi-core CPUs (Figure (a), and with respect to BM15 [112], and GUNROCK [156] on a GPU.

Graph name	Multi-core CPU			GPU		
	OUR	**APGRE**	**LIGRA**	**OUR**	**BM15**	**GUNRO**
roadNet-CA	47655.886	97085	198442.29	37112	67247	132243
roadNet-TX	19208.212	34447	85925.29	16035	28965	111214
soc-Epinions1	48.731	85	306.08	44.17	114.36	266.56
patents_main	905.738	1133	5564.65	888.6	1602	1267
coAuthorsDBLP	889.741	1734	6041.15	615.81	1237	1215
soc-Slashdot0902	103.658	161.21	345.11	98.24	211.37	251.1
caidaRouterLevel	313.188	658.56	2050.60	286.94	725.84	854.54
scircuit	315.372	480.86	2411.06	307.24	460.11	2173
soc-sign-epinions	130.694	229.55	938.60	109.97	381.63	686.42
p2p-Gnutella31	27.925	49.76	233.65	21.62	84.11	228.42

Table 7.5: This table shows absolute time of OUR algorithm, labeled OUR, BM15, GUNROCK and APGRE on the entire graph. Times above a thousand seconds are rounded to the nearest integer.

Our results indicate that for problems such as betweenness-centrality, using an ear decomposition is effective and practical.

We believe that our technique is of independent interest and can be applied to other graph problems.

Chapter 8

Enumerative Algorithms on Graphs

There has been a huge interest in graph enumeration problems such as listing cliuqes, listing triangles,...

Chapter 9

Conclusions

We now conclude.

Bibliography

[1] Cuda c programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`. Accessed: 2016-03-19.

[2] Intel math kernel library. `https://software.intel.com/en-us/intel-mkl`. Accessed: 2016-03-19.

[3] stanford network analysis platform dataset. `http://www.cise.u.edu/research/sparse/matrices/{snap}`.

[4] the university of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[5] Sas(r) optgraph, `http : / / support`, note = sas.com/documentation/cdl/en/procgralg/68145/PDF/default/procgralg.pdf. 1992.

[6] Thrust c. 1992. ++ library, `https://developer.nvidia.com/thrust`.

[7] réka albert, hawoong jeong, and albert-lászló barabási. internet: diameter of the world-wide web. *nature*, 401(6749):130–131, 1999.

[8] arvind arasu, jasmine novak, andrew tomkins, and john tomlin. pagerank computation and the structure of the web: experiments and algorithms. In *proceedings of the eleventh international world wide web conference, poster track*, pages 107–117, 2002.

[9] Sepehr Assadi. Simple round compression for parallel vertex cover. *CoRR*, abs/1709.04599, 2017.

[10] Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for $(\Delta + 1)$ vertex coloring. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 767–786, 2019.

[11] lars backstrom, dan huttenlocher, jon kleinberg, and xiangyang lan. group formation in large social networks: membership, growth, and evolution. In *proceedings of the 12th acm sigkdd international conference on knowledge discovery and data mining*, pages 44–54. acm, 2006.

[12] D. A. Bader and K. Madduri. GTgrpah: A suite of synthetic graph generators.

[13] David Bader and A.K. Illendula. An experimental comparison of parallel algorithms for ear decomposition of graphs using two leading paradigms. Technical report, 2000.

[14] D. S. Banerjee, S. Sharma, and K. Kothapalli. Work efficient parallel algorithms for large graph exploration. In *HiPC*, 2013.

[15] Dip Sankar Banerjee and Kishore Kothapalli. Hybrid Algorithms for List Ranking and Graph Connected Components. In *Proc. HiPC*, 2011.

[16] Dip Sankar Banerjee, Ashutosh Kumar, Meher Chaitanya, Shashank Sharma, and Kishore Kothapalli. Work efficient parallel algorithms for large graph exploration on emerging heterogeneous architectures. *Journal of Parallel and Distributed Computing*, 76:81–93, 2015.

[17] Dip Sankar Banerjee, Shashank Sharma, and Kishore Kothapalli. Work efficient parallel algorithms for large graph exploration. In *20th Annual International Conference on High Performance Computing, HiPC 2013, Bengaluru (Bangalore),*

Karnataka, India, December 18-21, 2013, pages 433–442. IEEE Computer Society, 2013.

[18] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Computing strongly connected components in parallel on CUDA. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 544–555. IEEE, 2011.

[19] baruch awerbuch and tripurari singh. new connectivity and msf algorithms for ultracomputer and pram. In *international conference on parallel processing, icpp'83, columbus, ohio, usa, august 1983*, pages 175–179. ieee computer society, 1983.

[20] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. pages 1–12, 2012. In Proc. ACM SC, pp. 12::10.

[21] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[22] Guy E. Blelloch and Bruce M. Maggs. Parallel algorithms. In *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*. Chapman & Hall/CRC, 2010.

[23] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW)*, 2004.

[24] B Bollobás. Random graphs,. 2011. Cambridge University Press.

[25] B. Bollobás. *Random graphs*. Cambridge University Press, 2001.

[26] U. Brandes. A faster algorithm for betweenness centrality. *J. Math. Sociology*, 25(2), 2001.

[27] U. Brandes, D. Wagner, M Junger, and P Mutzel. Visone – analysis and visualization of social networks. pages 321–340, 2003. in Graph Drawing Software, Springer-Verlag, .

[28] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, 2000.

[29] Andrei Z. Broder, Ronny Lempel, Farzin Maghoul, and Jan O. Pedersen. Efficient pagerank approximation via graph aggregation. In *Proceedings of the 13th international conference on World Wide Web*, pages 484–485, 2004.

[30] Ian Buck, Tim Foley, Daniel Reiter Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[31] Aydin Buluc and John R Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 503–510. IEEE, 2008.

[32] Aydin Buluç, John R. Gilbert, and Ceren Budak. Solving path problems on the GPU. *Parallel Comput.*, 36(5-6):241–253, 2010.

[33] R. Butafogo and B. Schneiderman. Identifying aggregates in hypertext structures,. pages 63–74, 1991. Proc. 3rd ACM Conference on Hypertext.

[34] A. Buluç, S. Beamer, K. Madduri, K. Asanovic, D. Patterson, and D. Bader. Distributed-memory breadth-first search on massive graphs. in parallel graph algorithms,. 2015. Boca Raton, FL, USA:CRC Press.

[35] E. Caceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and bsp. In *In 24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, pages 390–400, 1997.

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

[37] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446, 2004.

[38] J. Cheriyan and R. Thurimella. Algorithms for parallel k-vertex connectivity and sparse certificates. pages 391–401, 1991. in Proc. ACM Symp. Th. Comp. (STOC).

[39] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. pages 1–14, 2012. In Proc. Intl Conf. on High Perf. Comp., Net., Storage and Anal. (SC). pp. 14::10.

[40] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (ogdf). *Handbook of Graph Drawing and Visualization*, pages 543–569, 2011.

[41] JeeWhan Choi, Daniel Bedard, Robert J. Fowler, and Richard W. Vuduc. A roofline model of energy. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, pages 661–672. IEEE Computer Society, 2013.

[42] Fan Chung, Paul Horn, and Linyuan Lu. Diameter of random spanning trees in a given graph. *Journal of Graph Theory*, 69(3):223–240, 2012.

[43] SuiteSparse Matrix Collection. `https://sparse.tamu.edu/`.

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

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

[46] cormen, leiserson, rivest, and stein. *introduction to algorithms*. prentice hall.

[47] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms. The MIT PRess, Third Edition, 2009.

[48] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to algorithms, 2001.

[49] Pilu Crescenzi, Roberto Grossi, Michel Habib, Leonardo Lanzi, and Andrea Marino. On computing the diameter of real-world undirected graphs. *Theor. Comput. Sci.*, 514:84–95, 2013.

[50] Artur Czumaj, Slobodan Mitrovic, Jakub Łącki, Krzysztof Onak, Aleksander Mądry, and Piotr Sankowski. Round compression for parallel matching algorithms. *Proceedings of the Annual ACM Symposium on Theory of Computing*, (1):471–484, 2018.

[51] d. m eckstein. bfs and biconnectivity. Technical Report 79-11, Dept. of Computer Science, Iowa State University of Science and Technology, 1979.

[52] david e. culler, richard m. karp, david a. patterson, abhijit sahay, eunice e. santos, klaus e. schauser, ramesh subramonian, and thorsten von eicken. logp: a practical model of parallel computation. *commun. acm*, 39(11):78–85, 1996.

[53] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[54] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[55] dip sankar banerjee, ashutosh kumar, meher chaitanya, shashank sharma, and kishore kothapalli. work efficient parallel algorithms for large graph exploration on emerging heterogeneous architectures. *jpdc*, 76:81–93, 2015.

[56] Hristo Djidjev, Guillaume Chapuis, Rumen Andonov, Sunil Thulasidasan, and Dominique Lavenier. All-pairs shortest path algorithms for planar graph for gpu-accelerated clusters. *Journal of Parallel and Distributed Computing*, 85:91–103, 2015.

[57] nhat tan duong, quang anh pham nguyen, anh tu nguyen, and huu-duc nguyen. parallel pagerank computation using gpus. In *proceedings of the third symposium on information and communication technology*, soict '12, pages 223–230, 2012.

[58] D. Dutta, M. Chaitanya, K. Kothapalli, and D. Bera. Applications of ear decomposition to efficient heterogeneous algorithms for shortest path/cycle problems. pages 864–873, 2017. in Proc. IPDPS Workshops, pp.

[59] J A. Edwards and U. Vishkin. Better speedups using simpler parallel programming for graph connectivity and biconnectivity. pages 103–114, 2012. in Proc. of Intl. Work. Prog. Mod. and App. for Multicores and Manycores (PMAM), pp..

[60] J. A. Edwards and U. Vishkin. Speedups for parallel graph triconnectivity. pages 190–192, 2012. in Proc. ACM Symp. Par. Alg. and Arch. (SPAA), pp..

[61] Nadav Eiron, Kevin S. McCurley, and John A. Tomlin. Ranking the web frontier. In *Proceedings of the 13th international conference on World Wide Web*, pages 309–318, 2004.

[62] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. GPUQP: query co-processing using graphics processors. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1061–1063, 2007.

[63] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 285–297, 1999).

[64] g. cong and d. a. bader. an experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps). In *proc. ieee ipdps*, 2005.

[65] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*, page 3, 2005.

[66] robert geisberger, peter sanders, and dominik schultes. better approximation of betweenness centrality. In *alenex*, pages 90–100. siam, 2008.

[67] Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for MIS, matching, and vertex cover. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 129–138, 2018.

[68] Mohsen Ghaffari, Ce Jin, and Daan Nilis. A massively parallel algorithm for minimum weight vertex cover. In Christian Schei-

deler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 259–268. ACM, 2020.

[69] Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. Technical report, 2019.

[70] Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1636–1653, jul 2019.

[71] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms. *SIAM J. Comp.*, 28(2):733–769, 1999.

[72] David F. Gleich. PageRank beyond the web. *arXiv*, cs.SI:1407.5107, 2014. Accepted for publication in SIAM Review.

[73] D. K. Goldenberg, P. Bihler, M. Cao, J. Fang, B. Anderson, A. S. Morse, and Y. Yang. Localization in sparse networks using sweeps. 2006. In Proc. Intl. Conf. Mobile comp. and Net., pp. 110 – 121, .

[74] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 325–336. ACM, 2006.

[75] Naga K. Govindaraju, Ilknur Kabul, Ming C. Lin, and Dinesh Manocha. Fast continuous collision detection among deformable models using graphics processors. *Comput. Graph.*, 31(1):5–14, 2007.

[76] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha. Efficient collision culling among deformable objects using graphics processors. *Presence Teleoperators Virtual Environ.*, 15(1):62–76, 2006.

[77] John Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, SPAA '94, pages 16–25. ACM, 1994.

[78] John Greiner. A comparison of parallel algorithms for connected components. In Lawrence Snyder and Charles E. Leiserson, editors, *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '94, Cape May, New Jersey, USA, June 27-29, 1994*, pages 16–25. ACM, 1994.

[79] B. Haeupler, K. R. Jampani, and A. Lubiw. Testing simultaneous planarity when the common graph is 2-connected. 2010. in Algorithms and Computation. Proc. of ISAAC, .

[80] Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU using CUDA. In *Proc. of HiPC*, 2007.

[81] Nicholas J. A. Harvey, Christopher Liaw, and Paul Liu. Greedy and local ratio algorithms in the mapreduce model. In Christian Scheideler and Jeremy T. Fineman, editors, *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 43–52. ACM, 2018.

[82] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. pages 461–464, 1979. Commun. ACM 22(8), pp..

[83] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, 1979.

[84] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In Lawrence Rauchwerger and Vivek Sarkar, editors, *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, pages 78–88. IEEE Computer Society, 2011.

[85] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. In *Proc. of SC'13*, 2013.

[86] J. E. Hopcroft and R. E. Tarjan. Isomorphism of planar graphs. pages 131–152, 1972. In: Miller R.E., Complexity of Computer Computations. The IBM Research Symposia Series. Springer, pp..

[87] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. pages 135–158, 1973. SIAM J. Computing, 2(3), pp..

[88] SivaRamaKrishna Bharadwaj Indarapu, Manoj Kumar Maramreddy, and Kishore Kothapalli. Architecture- and workload-aware heterogeneous algorithms for sparse matrix vector multiplication. In *ACM COMPUTE*, pages 3:1–3:9, 2014.

[89] j. leskovec, k. lang, a. dasgupta, and m. mahoney. community structure in large networks: natural cluster sizes and the absence of large well-defined clusters.

[90] J. JaJa. An introduction to parallel algorithms,. 2004. Addison-Wesley, .

[91] Josef Jaja. *An Introduction To Parallel Algorithms*. Addison-Wesley, 2004.

[92] Yuntao Jia, Victor Lu, Jared Hoberock, MichaelGarl, and John C.Hart. Chapter 2 - edge v. node parallelism for graph centrality

metrics. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, pages 15 – 28. Morgan Kaufmann, Boston, 2012.

[93] jure leskovec, jon m. kleinberg, and christos faloutsos. graph evolution: densification and shrinking diameters. *acm trans. knowl. discov. data*, 1(1):2, 2007.

[94] k. george. slota and kamesh. madduri. simple parallel biconnectivity algorithms for multicore platforms. In *hipc*, pages 1–10, 2014.

[95] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Exploiting the block structure of the web for computing pagerank. Technical Report 2003-17, Stanford University, 2003.

[96] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. pages 288–306, 1991. in Journal of Computer and System Sciences Volume 42, pp..

[97] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proc. ACM SPAA*, pages 938–948, 2010.

[98] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Intl. Conf. Par. Proc.*, pages 113–122, 1995.

[99] A. Kazmierczak and S. Radhakrishnan. An optimal distributed ear decomposition algorithm with applications to biconnectivity and outerplanarity testing. *IEEE Transactions on Parallel and Distributed Systems*, 11(1):110–118, 2000.

[100] S. Khuller and B. Schieber. Efficient parallel algorithms for testing connectivity and finding disjoint $s - t$ paths in graphs. pages 288–293, 1989. in Proc, of IEEE Symp. Found. Comp. Sci. (FOCS), pp..

[101] H. Klauck, D. Nanongkai, G. Pandurangan, and P. Robinson. Distributed computation of large-scale graph problems. In *Proc. ACM SODA*, pages 391–410, 2015.

[102] Christian Kohlschutter, Paul-Alexandru Chirita, and Wolfgang Nejdl. Efficient parallel computation of pagerank. In *Proc. of the 28th European Conference on IR Research*, pages 241–252, 2006.

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

[104] Piyush Kumar. *Cache Oblivious Algorithms*, pages 193–212. Springer Berlin Heidelberg, 2003.

[105] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. GPU sample sort. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, pages 1–10. IEEE, 2010.

[106] LLVM. Llvm's analysis and transform passes. http://llvm.org/docs/Passes.html, 2003.

[107] J. Lou, S. Lin, K. Chen, and C. Lei. What can the temporal social behavior tell us? an estimation of vertex-betweenness using dynamic social information. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 56–63, 2010.

[108] L. Lovasz. Computing ears and branchings in parallel. In *Proc 26th IEEE Ann. Symp. on Foundations of Comp. Sci.*, pages 464–467, 1985.

[109] Lijuan Luo, Martin D. F. Wong, and Wen-mei W. Hwu. An effective GPU implementation of breadth-first search. In Sachin S. Sapatnekar, editor, *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 52–55. ACM, 2010.

[110] Michael D. McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh.* A K Peters, 2004.

[111] A. McLaughlin and D. A. Bader. Scalable and high performance betweenness centrality on the gpu. In *ACM SC*, pages 572–583, 2014.

[112] Adam McLaughlin and David A. Bader. Fast execution of simultaneous breadth-first searches on sparse graphs. In *21st IEEE International Conference on Parallel and Distributed Systems, ICPADS 2015, Melbourne, Australia, December 14-17, 2015*, pages 9–18. IEEE Computer Society, 2015.

[113] Kurt Mehlhorn and Stefan Naher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[114] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. 2015. in ACM Transactions on Parallel Computing, Volume 1 Issue 2, .

[115] Y. Moan, B. Schieber, and U. Vishkin. Parallel ear decomposition search (eds) and st-numbering in graphs. *Theoretical Computer Science*, 47(3):277–296, 1986.

[116] R. Motwani and P.Raghavan. Randomized algoritihms. 1992.

[117] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms.* Cambridge University Press, New York, NY, USA, 1995.

[118] NVidia. cuBLAS api referene guide. https://docs.nvidia.com/cuda/cublas/index.html, 2020.

[119] C. Pachorkar, M. Chaitanya, K. Kothapalli, and D. Bera. Efficient parallel ear decomposition of graphs with application to betweenness-centrality. pages 301–310, 2016. in Proc. of Intl. Conf. High Perf. Comp., pp..

[120] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[121] G. Pandurangan, P. Robinson, and M. Scquizzato. Fast distributed algorithms for connectivity and mst in large graphs. pages 429–438, 2016. in Proc. ACM Symp. Par. Alg. and Arch. (SPAA), pp..

[122] paolo boldi, marco rosa, massimo santini, and sebastiano vigna. layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *proceedings of the 20th international conference on world wide web*. acm press, 2011.

[123] paolo boldi and sebastiano vigna. the webgraph framework i: compression techniques. In *proc. of the thirteenth international world wide web conference (www 2004)*, pages 595–601, manhattan, usa, 2004. acm press.

[124] Suresh Purini and Lakshya Jain. Finding good optimization sequences covering program space. *ACM Trans. Archit. Code Optim.*, 9(4):56:1–56:23, 2013.

[125] R. Puzis, P. Zilberman, Y. Elovici, S. Dolev, and U. Brandes. Heuristics for speeding up betweenness centrality computation. In *SocialCom*, pages 302–311, 2012.

[126] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 275–340. Morgan-Kaufmann, 1993.

[127] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. pages 275–340, 1993. in Synthesis of Parallel Algorithms, J. H. Reif, Ed. Morgan-Kaufmann, pp.

[128] V. Ramachandran and U. Vishkin. Efficient parallel triconnectivity in logarithmic time,. pages 33–42, 1988. In: Reif J.H. (eds) VLSI Algorithms and Architectures, Lecture Notes in Computer Science, vol 319, pp..

[129] M. S. Rehman, K. Kothapalli, and P. J. Narayanan. Fast and Scalable List Ranking on the GPU. In *Proc. of ACM ICS*, 2009.

[130] matthew richardson, rakesh agrawal, and pedro domingos. trust management for the semantic web. In *the semantic web-iswc 2003*, pages 351–368. springer, 2003.

[131] R. Rivest, A. Shamir, and L Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120 – 126, 1978.

[132] Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. Fast distributed pagerank computation. In *Proc. of ICDCN*, pages 11–26, 2013.

[133] A. Erdem. Sarıyuce, E .Saule, K. Kaya, and U. V. Catalyurek. Betweenness centrality on gpus and heterogeneous architectures. pages 76–85, 2013.

[134] A. Erdem. Sarıyuce, E .Saule, K. Kaya, and U. V. Catalyurek. Shattering and compressing networks for betweenness centrality. In *SIAM Data Mining*, 2013.

[135] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Proc. ACM Symp. Graphics Hardware*, pages 97– 106, 2007.

[136] Zhiao Shi and Bing Zhang. Fast network centrality analysis using gpus. *BMC Bioinformatics*, (12), 2011.

[137] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. pages 57–67, 1982. J. Algorithms 3(1), pp.

[138] Yossi Shiloach and Uzi Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, pages 57–67, 1982.

[139] Yossi Shiloach and Uzi Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[140] J. Shun, L. Dhulipala, and G. E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. pages 143–153, 2014. In Proc. ACM Symp. Par. Alg. and Arch. (SPAA), pp.

[141] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 135–146. ACM, 2013.

[142] G. M. Slota and K. Madduri. Simple parallel biconnectivity algorithms for multicore platforms. pages 1–10, 2014. in Proc. of Intl. Conf. High Perf. Comp. (HiPC), pp.

[143] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 550–559. IEEE Computer Society, 2014.

[144] J. Soman, K. Kothapalli, and P. J. Narayanan. Some gpu algorithms for graph connected components and spanning tree. pages 325–339, 2010. Parallel Processing Letters 20(4): pp.

[145] Jyothish Soman, Kishore Kothapalli, and PJ Narayanan. Some GPU Algorithms for Graph Connected Components and Spanning Tree. In *Parallel Processing Letters*, volume 20, pages 325–339, 2010.

[146] M. Sutton, T. Ben-Nun, and A. Barak. Optimizing parallel graph connectivity computation via subgraph sampling. pages 12–21, 2018. in Proc., of IEEE Intl. Par. Dist. Proc. Symp, pp.

[147] R. E. Tarjan. Depth first search and linear graph algorithms. pages 146–160, 1972. SIAM Journal on Computing, Vol. 1, pp.

[148] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM*, 1972.

[149] Robert Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM*, 1985.

[150] W. T. Tutte. Connectivity in graphs. 1966. University of Toronto Press,.

[151] L. G. Valiant. A Bridging Model for Parallel Computation. *Comm. ACM*, 33(8):103 – 111, 1990.

[152] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism. pages 140–151, 1992. In Proc. 10th ACM Symp. Par. Alg. and Arch. (SPAA), pp.

[153] M. Wadwekar and K. Kothapalli. A fast GPU algorithm for biconnected components. pages 1–6, 2017. in Proc. of Intl. Conf. Cont. Comp., pp.

[154] F. Wang, M. T. Thai, and D. Z. Du. On the construction of 2-connected virtual backbone in wireless networks. pages 1230–1237, 2009. in IEEE Transactions on Wireless Communications, pp.

[155] Lei Wang, Fan Yang, Liangji Zhuang, Huimin Cui, Fang Lv, and Xiaobing Feng. Articulation points guided redundancy elimination for betweenness centrality. In Rafael Asenjo and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*

2016, Barcelona, Spain, March 12-16, 2016, pages 7:1–7:13. ACM, 2016.

[156] Y. Wang, Y. Pan, A. D. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens. Gunrock: GPU graph analytics,. pages 3–49, 2017. in Proc. of ACM Trans. Parallel Computing, 4(1),.

[157] Yuan Wang and David J. Dewitt. computing pagerank in a distributed internet search engine system. In *proceedings of the thirtieth international conference on very large data bases*, pages 420–431, 2004.

[158] D. West. *Introduction to Graph Theory.* Prentice Hall, 1996.

[159] D. B. West. *Intorudction to Graph Theory.* Prentice Hall, 2001.

[160] H. Whitney. Non-separable and planar graphs. *Transactions of the American Mathematical Society*, 34:339–362, 1932.

[161] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *In Proc. of ACM SC*, 2007.

[162] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[163] Jie Wu and Karl Aberer. Using siterank for decentralized computation of web document ranking. In *Proc. of the Third Workshop on Adaptive Hypermedia and Adaptive Web-Based Systems*, pages 265–274, 2004.

[164] J. C. Wyllie. *The complexity of parallel computations.* PhD thesis, Cornell University, Ithaca, NY, 1979.

[165] Grigory Yaroslavtsev and Adithya Vadapalli. Massively parallel algorithms and hardness for single-linkage clustering under L_p distances. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 5596–5605. PMLR, 2018.