Provably Efficient and Scalable Shared-Memory Graph Processing

Laxman Dhulipala

CMU-CS-20-128
August 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy E. Blelloch, Chair
Umut Acar
Phillip B. Gibbons
Vahab Mirrokni (Google Research)
Julian Shun (MIT)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported in part by the National Science Foundation under grants numbers CCF-1408940, CCF-1533858, CCF-1629444, and CCF-1845763, and by the Intel Science and Technology Center for Cloud Computing. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF, or other funding parties.

Keywords: parallel algorithms, shared-memory algorithms, parallel graph processing, parallel batch-dynamic algorithms, graph-streaming systems

Abstract

Graph processing is a fundamental tool in many computational disciplines due to the widespread availability of graph data. However, processing large graphs quickly and cost-effectively is a major challenge, and existing approaches capable of processing large graphs have high computational cost, only solve a limited set of problems, and possess poor theoretical guarantees. Similarly, existing graph processing approaches for dynamic or streaming graphs employ ad-hoc algorithms with unknown theoretical costs and suboptimal performance. This thesis argues that shared-memory algorithms can serve as the foundation for a graph processing toolkit for static and evolving graphs that is cost-effective, has strong theoretical guarantees, and achieves state-of-the-art performance.

The first part of this thesis studies static graph processing. We design a rich interface for parallel graph processing which extends the Ligra interface with provably-efficient primitives. Using the interface, we design provably-efficient shared-memory implementations of over 20 fundamental graph problems. Our implementations, which we have made publicly available as part of the Graph Based Benchmark Suite, solve these problems on the largest publicly available graph, the WebDataCommons hyperlink graph, with over 200 billion edges, in a matter of seconds to minutes using a commodity multicore machine. We also adapt our algorithms for graphs stored in non-volatile memory, thereby extending our results to graphs larger than main memory. Compared to existing graph processing results, our results apply to a much broader set of problems, use orders of magnitude fewer resources, and in many cases run an order of magnitude faster.

The second part of this thesis studies graph processing in the batch-dynamic model, which generalizes the classic dynamic algorithms model by allowing algorithms to ingest batches of updates. We design work-efficient parallel batch-dynamic algorithms with polylogarithmic depth for the dynamic trees and dynamic connectivity problems. We show that shared-memory parallel batch-dynamic algorithms can achieve strong speedups and outperform the fastest sequential dynamic algorithm baselines.

The final part of this thesis studies streaming graph processing. We design a compressed purely-functional tree data structure, called a *C*-tree, which admits efficient parallel batch updates and enables a dynamic graph representation based on nested trees. Using this representation, we design a serializable graph-streaming system called Aspen that can concurrently apply updates and queries. Compared to existing work, Aspen achieves orders of magnitude higher update rates, while using less memory. We show that Aspen can concurrently update and analyze the WebDataCommons hyperlink graph on a single commodity multicore machine.

For my grandfather

Acknowledgements

I would like to thank my advisor, Guy Blelloch, for his support, encouragement, and mentoring which were instrumental to the completion of this thesis. It has been a great privilege to work with Guy. Guy's office door was always open, and he was always welcoming when I stopped by for an unscheduled visit to discuss some problems or ideas. Thank you for taking a chance to work with me when I was a undergraduate, and for the guidance and unerring advice you gave in the years that came afterwards. Working with you was one of the best parts of graduate school—thank you for everything Guy.

I am very grateful to all of the members of my thesis committee. Julian, thank you for many years of mentorship, collaboration, and friendship. You were a role model for me as a young undergraduate (and still are!), and were a big part of why I was motivated to come back to graduate school. You have been involved in much of the research presented in this thesis, and I am truly grateful for all of your help, insights, and advice with this work. Umut, thank you for all of your help, advice, and for being a positive influence in my life both as an undergraduate and as a graduate student. I have many fond memories of working with you, from TA'ing 15-210 with you, Kanat, and Margaret, and our memorable drive to attend PPoPP, to all of the candid and helpful advice you gave me throughout graduate school. Phil is one of the most good natured and down-to-earth people I have worked with. I'm so happy that I got to know you better through working with you and the rest of the NVM group. In research, I'm always struck by your optimism and positive attitude. Thank you for all of your mentorship and advice. I have truly enjoyed working with Vahab and his group over the past two years. Vahab is optimistic and patient, has a great sense of humor, and also has deep intuition for promising ideas. I am very grateful to you for encouraging and believing in our work on shared memory algorithms, for all of your advice and mentorship.

I would like to thank all of the mentors who hosted me during visits and summer internships away from CMU. Zoya, thank you for your kindness and for being a great host to work with; working with you helped me gain confidence about doing research outside of areas I felt comfortable with. Kuba, you're a great researcher and role model, and I've really enjoyed working with you over the past two years. Thank you for all of your advice, help, and hard work, and for always being optimistic about the work we did together. I would also like to thank Saman Amarsinghe, who I interacted with during GraphIt meetings during a summer at MIT. Working with you and your group was a great experience, and I left that summer with many warm memories about being at MIT.

Thanks to Charles Leiserson, Jeremy Fineman, T.B. Schardl, and Tim Kaler, and all of the participants of the Parlay meetings over the years who were welcoming and accepting

even when I was young and inexperienced. Thanks to the organizers of the Algorithms for Big Data workshop—Timo Bingmann, Tim Conrad, Ulrich Meyer, and Matthias Mnich—for inviting me to visit Dagstuhl. Thanks to Janardhan Kulkarni and Richard Peng for inviting me to visit MSR, and for some memorable discussions. Thank you to Deborah Cavlovich, Catherine Copetas, Jennifer Landefeld, and Benjamin Cook for all of your help with navigating the Ph.D. experience!

I would like to extend my sincere thanks to all of the collaborators I worked with during this time. Thank you to Ajay Brahmakshatriya, Alon Shalita, Brian Karrer, Changwan Hong, Charles McGuffey, Chi Wang, Daniel Anderson, David Durfee, Guissepe Ottaviano, Hao Wei, Hongbo Kang, Hossein Esfandieri, Igor Kabiljo, Janardhan Kulkarni, Jessica Shi, Jiezhong Qiu, Katharina Flügel, Michal Friedman, Naama Ben-David, Quanquan Liu, Richard Peng, Sam Westrick, Sergey Pupyrev, Shangdi Wang, Shoaib Kamil, Soheil Behnezhad, Tom Tseng, Warren Schudy, Xiaorui Sun, Xinyi Chen, Yan Gu, Yihan Sun, Yiqiu Wang, and Yunming Zhang for their insights, good humor, hard work, and enthusiasm, which made doing research with you so much fun.

I am very thankful to all of the friends I have met both at CMU and outside of it during these years. To Fait Poms and Nic Resch, thanks for being inspiring and eclectic roommates. I grew a lot spending time with you, and I am grateful to both of you for many interesting conversations. To Hao Wei, thank you for being a model roommate and friend; it was always a pleasure to come home and be able to talk about everything from music to solving puzzles with you. To Ellis Hershkowitz, thanks for being a positive influence in my life. I miss our walks through Schenley that swung from excitement about new research ideas, to commiserating about the rougher aspects of graduate school. To Lin Ma and Charlie McGuffey, thanks for being excellent officemates. Lin, thanks for all of the great conversations, Szechuan food we had together, and for encouraging me to swim more. Thanks to you, Charlie, and Ellis for watering the office plants while I was away. Thank you to everyone who participated in ParallelRG, and gave talks and shared their knowledge with me! Thanks to Cheul Young Park, Nishanth Maddineni, Tsutomu Okano, and all of our friends from Stever for their continued friendship all this time later. Thanks to Arjun Iyer, Sumukh Anand, and Vimal Balu for their long friendship, quick wit, and wisdom. There are many other friends that I met at CMU that have shared their ideas and passions with me—Alex Wang, Anson Kahng, Brandon Kase, Cyrus Omar, David Wajc, David Witmer, Goran Zuzic, Jordan Williams, Naama Ben-David, Roie Levin, Ryan Stout, Sam Westrick, Shannon Joyner, Shayak Sen, Todd Nowacki, Vijay Bhattiprolu, and so many others—thank you for being positive influences in my life.

This thesis could not have been written without the unconditional support and love of my family. I am who I am today only because of you. This thesis is dedicated in loving memory of my grandfather, Dr. D. Lakshmana Rao.

Contents

List of Figures

List of Tables

1 *Introduction*

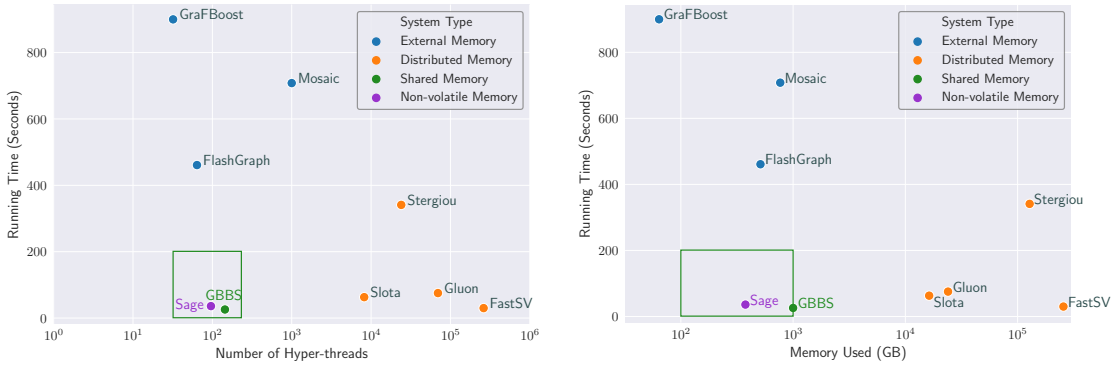
The goal of gaining a deeper understanding of computing through a suitable mix of both theoretical understanding and well-designed experiments should be a goal we all work toward together.

Michael Mitzenmacher
Theory Without Experiments: Have We Gone Too Far?

In recent years, a rapid growth in the availability of graph datasets has led to significant interest in algorithms and systems that can efficiently represent and process graph data. As a representative example of a real-world graph with many applications, consider the WebDataCommons hyperlink graph, the largest publicly available graph today, which contains 3.5 billion vertices representing Web pages, and 128 billion directed edges representing hyperlinks from one page to another [241]. Understanding and analyzing the structure and properties of this graph is of fundamental interest in many different applications and communities, including search engines, computer storage systems, and even web anthropology. However, the sheer size of this graph poses a significant computational challenge, and to date, very few graph processing tools can analyze a graph of this scale. It is likely that graphs in the near future will continue to grow in size[1], and thus designing a powerful graph processing toolkit that enables users to quickly process large graphs like the WebDataCommons graph is a fundamental task.

Interest in real-time data analytics for graphs has led to interest in processing evolving graphs in a batch setting, specifically for streaming and dynamic graph processing. The goal of streaming graph-processing is to represent an evolving graph as it is updated, and to make consistent snapshots of this graph available to graph-processing queries that arrive concurrently with the updates. A dynamic graph algorithm ingests a sequence of updates and queries to the graph and recompute analytics on the graph in response to the changes to the graph. The *batch* setting, where updates and queries can arrive in arbitrary sized unordered batches, is important in both of these tasks since the rate at which graphs such as the Twitter social network change can be on the order of hundreds of thousands of updates per second. For such rapid update rates, instead of updating the graph or recomputing an analytic (for example, connectivity, or a clustering of the graph) after every update, it is usually sufficient in practice to periodically apply the update every few seconds, or minutes, depending on the latency requirements of the use-case. Unfortunately existing solutions in the literature for both dynamic and graph-streaming systems suffer

[1]Note that there are already several private graph datasets at companies such as Google and Yahoo! which contain several hundred billion vertices and 6–7 trillion edges [336, 209].



Hyper-threads versus running time

Memory (Gb) versus running time

Figure 1.1: Plot of the amount of hardware used (number of hyper-threads, and memory in gigabytes) versus running time in seconds for existing systems that can solve connected components on the WebDataCommons hyperlink graph. The green boxes correspond to an ideal system which uses relatively little hardware and also achieves low running time. Note that Mosaic [225] only reports running times for the 2014 WebDataCommons graph, with roughly half the edges. GraFBoost [191] unfortunately does not report running times for connected components in their paper, but we report their running time for Single Source Betwenness Centrality, which is the fastest time they report on this graph.

from poor latency, struggle to scale to very large graphs such as the WebDataCommons graph, and have unknown theoretical costs [130, 100, 139, 156, 226].

Even considering the well-studied situation in static graph processing, existing static graph processing tools only solve a limited number of problems, and often produce approximations or incomplete results on very large graphs due to difficulty of exactly computing certain graph analytics. For example, a recent paper studied how to compute the coreness values[2] of the WebDataCommons graph [330]. Despite an impressive engineering effort, and using a supercomputer with 256 machines, 8192 hyper-threads, and 16.3 terabytes of memory, they were unable to compute the exact coreness values, and instead were able to obtain an approximation of the values to the nearest power of two in 6 minutes using this configuration. The situation is similar for other problems, such as strongly connected components (SCCs), where no existing parallel graph processing system can compute the SCCs of the WebDataCommons graph. In general, very few algorithms have been applied to this graph, and those that have often take hours to run [385, 225, 191], with the fastest times requiring between 1–6 minutes using a supercomputer [333, 330].

Figure 1.1 illustrates the situation for existing external memory and distributed memory

[2] A *k-core* of a graph is the maximal subgraph such that every vertex in the subgraph has degree at least k. The *coreness* value of a vertex is the maximum k such that it is contained in a k-core.

results that compute the connected components of the WebDataCommons hyperlink graph. The two plots display the running time of the system, plotted against the number of hyper-threads and the amount of DRAM, respectively. These plots show that existing distributed-memory results can be fast, but have a high computational cost, frequently using thousands of hyper-threads and terabytes of memory to achieve these results [333, 111, 336, 382]. Existing external-memory results have lower hardware cost in terms of hyper-threads and memory used, but are significantly slower compared to distributed-memory results [385, 225, 191]. The cost of running these analytics in the cloud today is dictated both by the total amount of hardware resources (cores and memory) used, and the duration of time the resources are used, and thus both distributed-memory and external-memory result in high cost. Furthermore, these solutions often have poor or unknown theoretical bounds making it hard to predict how they will perform on other types of graphs.

To summarize, today if one is a user interested in processing very large graphs, or dynamic and streaming graphs, there are no reliable solutions that are scalable, possess strong theoretical bounds, and are cost-efficient. Motivated to improve upon these limitations and the narrow applicability of existing graph processing tools, in this thesis, we study the following question:

Can we design a graph-processing toolkit that enables users to easily and cost-effectively solve a broad set of problems on massive static and evolving graphs?

Our Approach in this Thesis

In this thesis, we argue that a *shared-memory approach* to graph processing enables the construction of a graph processing toolkit that is affordable, practical, scalable, and provably-efficient. Specifically, this thesis considers shared-memory multicore machines, which are machines that contain multiple CPUs (processing cores) sharing a single shared address space. Such machines are widely available today, and shared-memory machines with dozens to hundreds of cores and terabytes of memory can be rented for tens of dollars on the hour on web services like Amazon Web Services and Google Cloud Platform. Indeed, today is is hard to find a mainstream processor, even those designed for phones, that do not have multiple processing cores. With Moore's law—the prediction that the number of transistors on computer chips will double every two years—now ending, chip manufacturers are embracing increasing core and shared-memory parallelism as a path towards continued efficiency gains [213].

There are a number of advantages of a shared-memory approach to graph processing. First, despite the increasing sizes of real-world graphs today, even the WebDataCommons hyperlink graph—the largest publicly available graph with 3.5 billion vertices, and 128 billion edges—can be represented in the main-memory of a relatively modest shared-

memory machine. Thus, by designing efficient graph algorithms for a shared-memory setting, we can potentially solve a wide range of fundamental problems on the largest graphs available today using in-memory processing techniques, which can be an order of magnitude faster than using distributed graph processing techniques [238]. Furthermore, by focusing on a shared-memory setting, we can take advantage of decades of research on designing provably-efficient, and ideally work-efficient parallel graph algorithms (e.g., [316, 196, 224, 17, 346, 247, 284, 188, 104, 277, 246, 140, 57, 242]). In this thesis, we analyze parallel algorithms based on their ***work*** and ***depth***, which are the total number of operations performed by the algorithm, and the longest chain of sequential dependencies in the algorithm, respectively.[3] In parallel algorithm design, the gold standard is to design a ***work-efficient*** parallel algorithms, which is an algorithm that performs asymptotically the same amount of work as the fastest sequential algorithm for the task.

There are several reasons that algorithms with good theoretical guarantees are desirable. For one, they are robust as even adversarial inputs will not cause them to perform extremely poorly. Furthermore, they can be designed on pen-and-paper by exploiting properties of the problem instead of tailoring solutions to the particular dataset at hand. Theoretical guarantees also make it likely that the algorithm will continue to perform well even if the underlying data changes. Finally, careful implementations of algorithms that are nearly work-efficient can perform much less work in practice than work-inefficient algorithms. This thesis shows that this reduction in work often translates to faster running times on the same number of cores. We note that most running times that have been reported in the literature on the Hyperlink Web graph use parallel algorithms that are not theoretically-efficient.

The remainder of the chapter further explores these topics:

- Section 1.1 describes this thesis' contributions to shared-memory graph processing, including new implementation techniques, parallel algorithms, a benchmark suite of graph algorithms, and an extension of our approach to a non-volatile memory setting.

- Section 1.2 introduces the batch-dynamic model of computation, and describes our results on efficient parallel graph algorithms in this model for the forest connectivity, and general connectivity problems.

- Section 1.3 introduces the graph-streaming model, and introduces a purely-functional compressed tree data structure called a *C*-tree which admits efficient batch updates and is well-suited for this setting. It then introduces Aspen, a low-latency graph-streaming system build using *C*-trees.

[3] Chapter 2 provides a formal definition of work and depth in the parallel model of computation used in this thesis.

1.1 *Shared-Memory Graph Processing*

The first part of the thesis addresses how to cost-effectively solve a variety of fundamental graph problems and analytics on very large graphs. This part of the thesis comprises three chapters: Chapter 4, which presents a parallel bucketing data structure, and presents state-of-the-art results for a set of *ordered* graph algorithms, Chapter 5, which presents a benchmark suite of 20 fundamental graph problems, and presents theoretically-efficient and practical graph algorithms solving each of these problems, and Chapter 6, which considers the problem of efficient parallel graph analytics on larger-than-memory graphs stored on NVRAMs, and presents efficient techniques and algorithms that achieve state-of-the-art results in this setting.

Graph Processing using Ligra and Ligra+

This thesis builds on the *Ligra* graph processing framework [319]. The Ligra framework provides users with a graph data structure, a data structure for representing subsets of vertices called a vertexSubset, and two primitives called EDGEMAP and VERTEXMAP. The VERTEXMAP primitive lets users apply a user-defined function F to each vertex in an input vertexSubset in parallel. At a high level, the EDGEMAP primitive lets users apply a user-defined function F (returning a boolean) to each edge incident to a vertexSubset and create a new vertexSubset containing all neighbors for which F returns true.[4] These simple primitives enable solving a variety of fundamental graph analytics, such as breadth-first search, Bellman Ford, betweenness centrality, connected components, and PageRank.

In the years following the development of Ligra, many other graph processing systems were built, which by-and-large solve the same problems as Ligra, and either directly adopt Ligra's implementation approach or refinements of it. Consider, for example, Polymer [380], Ligra+ [322], Gemini [389], Grazelle [159], and GraphIt [383]. The novelty in each of these works is to improve the efficiency of the Ligra interface by enabling NUMA-awareness [380], enabling Ligra to process larger graphs using compression techniques [322], enabling distributed-processing [389], providing more sophisticated scheduler and vectorzation optimizations [159], and using compilation-based techniques to make it exceptionally easy to implement parallel graph programs [383].

Limitations of Existing Work

Unfortunately, despite over a hundred papers written during the six years of subsequent research on shared-memory graph processing, none of these works has significantly broadened the scope of graph problems considered, or applied their algorithms to very large graphs such as the Hyperlink Web graph. Furthermore, much of the existing work on

[4] Detailed definitions of VERTEXMAP, EDGEMAP and the Ligra API is given in Chapter 2.

Graph Dataset	Num. Vertices	Num. Edges	Uncompressed	Compressed	Savings
ClueWeb [81]	978,408,098	74,744,358,623	285GB	100GB	2.8x
Hyperlink2014 [241]	1,724,573,718	124,141,874,032	474GB	186GB	2.5x
Hyperlink2012 [241]	3,563,602,789	225,840,663,232	867GB	354GB	2.4x

Table 1.1: Large graph inputs studied in this thesis, including number of vertices, edges, memory required to store the graph in an uncompressed CSR format, memory required to store the graph in the parallel byte-compressed CSR format used in this thesis, and the savings obtained over the uncompressed format by the compressed format.

shared-memory graph processing does not provide strong theoretical guarantees on the parallel costs of the algorithms, making it hard to understand the algorithm's performance in practice, and also to predict whether the algorithm will achieve fast runtimes on a new set of graph inputs.

For example, the problem of identifying the largest k-core containing each vertex—a classic and fundamental data-mining task—is not supported by any existing graph processing framework, and prior to this thesis, the only parallel algorithms available were ParK [110], and a parallel local algorithm by Sariyuce et al. [299]. Both algorithms are work-inefficient, are not written within any graph framework, and thus require adoption of a different set of graph processing tools to use. A similar situation holds for many other important graph problems, including Δ-stepping, which is a more work-efficient version of the Bellman-Ford algorithm for graphs with positive edge weights [243], parallel approximate set-cover [67, 68], strongly connected components [176, 332], biconnectivity [331], and other problems. Finally, all of the aforementioned works also process only very small graphs, with a few billion edges at the most.

The Need to Process Large Graphs

The sizes of graphs available to us today is rapidly increasing, along with the need to quickly and efficiently analyze these graphs. For example, the nervous system of the worm *C. Elegans* was painstakingly assembled over a period of 8 years during the 1980s, and contained just 302 vertices representing neurons connected by 3498 edges [369]. Building this graph gave neuroscientists a new tool that could be used to generate hypotheses about neuron functionality and to understand behavioral responses when ablating specific neurons. Work is currently underway to sequence the connectome of small mammals (containing roughly 20 million neurons and 200 billion edges), and the long term goal of these efforts is to construct the connectome of a human brain (containing roughly 86 billion neurons and 250 trillion edges) [218]. A crucial aspect of using these brain networks in scientific research will be software and algorithms that can efficiently analyze and extract meaningful information from them.

Similar increases in the sizes of graphs have occurred in many other areas, such as

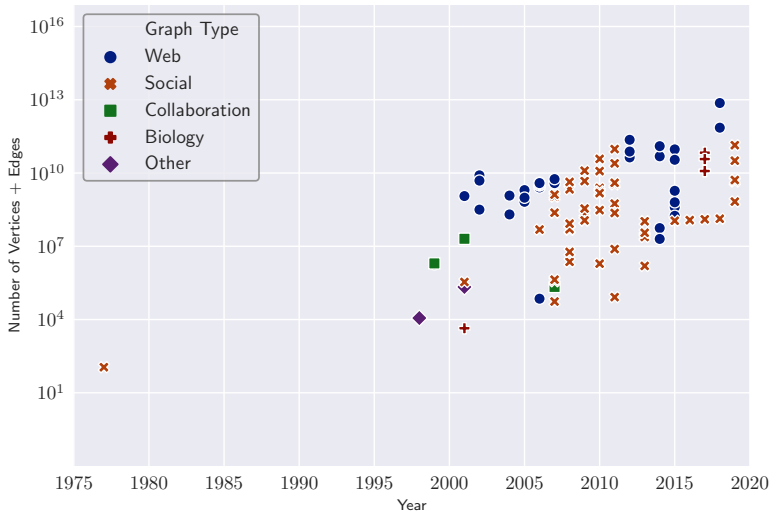


Figure 1.2: The total number of vertices and edges in different types of real-world graphs (in logarithmic scale) versus the year that the graph was reported in a publication. The data shows a consistent trend in increasing graph sizes since the mid 1990s.

social networks, routing networks, protein-interaction networks, E-commerce graphs, and graphs of the World Wide Web. Figure 1.2 plots the total number of vertices and edges in various graphs versus the year that the graph was reported in a publication. The figure shows a consistent trend in increasing graph sizes from the mid 1990s onwards. The reasons for this increase have come broadly from advancements in storage technology (which make it possible to gather and store large amounts of data), increased usage of the Internet, and data-driven approaches to scientific progress in many of these fields.

How this Thesis Extends Ligra

One of the contributions of this thesis is to show that suitable extensions to the Ligra graph processing interface enable solving a diverse set of fundamental graph analytics that significantly expands the set of problems solved by existing graph processing systems. Furthermore, by equipping these interfaces with strong bounds on their parallel costs, we can obtain strong provable guarantees on the costs of the resulting algorithms. Finally, by enabling the extended interface to work in conjunction with compression techniques from Ligra+, we can easily run any algorithm implemented using the interface on very large compressed graphs, without any algorithmic modifications. Table 1.1 illustrates the space savings obtained from using Ligra+ to store the three largest graph inputs studied in this thesis; enabling compression provides a 2.56x reduction in space, on average.

Algorithm	Work	Depth	Parameters
k-core	$O(\|E\| + \|V\|)$	$O(\rho \lg \|V\|)$	ρ: peeling complexity, see Chapter 2.
wBFS	$O(r_{src} + \|E\|)$	$O(r_{src} \lg \|V\|)$	r_{src}: eccentricity from the source vertex *src*, see Chapter 2.
Δ-stepping	$O(w_\Delta)$	$O(d_\Delta \lg \|V\|)$	w_Δ, d_Δ: work and number of rounds of the original Δ-stepping algorithm.
Approximate Set Cover	$O(M)$	$O(\lg^3 M)$	M: sum of the sizes of the sets.

Table 1.2: Cost bounds for the ordered graph algorithms studied in this thesis. The work bounds are in expectation and the depth bounds are with high probability.

Ordered Graph Algorithms

Chapter 4 presents a shared-memory graph processing framework called *Julienne* which extends the Ligra framework with an interface for *parallel bucketing*. Each of the problems that studied in this part of the thesis require maintaining a dynamic mapping between vertices and a set of ordered bucket over the course of the algorithm's execution. These problems include non-negative integral-weight single-source shortest paths, approximate set-cover, and k-core, amongst several others. Each of the efficient algorithms for these problems works as follows at a high level. In each round, the algorithm extracts the vertices contained in the lowest (or highest) bucket and performs some computation on these vertices. The algorithm then updates the buckets containing either the extracted vertices or their neighbors. Finally, the algorithm terminates once there are no further vertices remaining in the bucket structure.

To make this abstract algorithm template more concrete, consider the weighted breadth-first search (wBFS) algorithm, which solves the single-source shortest path problem (SSSP) with nonnegative, integral edge weights in parallel [125]. Like BFS, wBFS processes vertices level by level, where level i contains all vertices at distance exactly i from *src*, the source vertex. The i'th round relaxes the neighbors of vertices in level i and updates any distances that change. Unlike a BFS, where the unvisited neighbors of the current level are in the next level, the neighbors of a level in wBFS can be spread across multiple levels. Because of this, wBFS maintains the levels in an ordered set of buckets. On round i, if a vertex v can decrease the distance to a neighbor u it places u in bucket $i + d(v, u)$. Finding the vertices in a given level can then easily be done using the bucket structure. We can show that the work of this algorithm is $O(r_{src} + |E|)$ and the depth is $O(r_{src} \lg |V|)$ where r_{src} is the eccentricity from *src*.[5] However, without bucketing, the algorithm has to scan all vertices

[5]The eccentricity of a vertex is upper-bounded by the diameter of the graph; Chapter 2 provides precise definitions.

Shortest Path Problems	Breadth-First Search, Integral-Weight SSSP, General-Weight SSSP, Single-Source Betweenness Centrality, Single-Source Widest Path, k-Spanner
Connnectivity Problems	Low-Diameter Decomposition, Connected Components, Biconnected Components, Strongly Connected Components, Spanning Forest, Minimum Spanning Forest
Covering Problems	Maximal Independent Set, Maximal Matching, Graph Coloring, Approximate Set Cover
Substructure Problems	k-Core, Approximate Densest Subgraph, Triangle Counting
Eigenvector Problems	PageRank

Table 1.3: 20 important graph problems considered in the Graph Based Benchmark Suite (GBBS), covering a broad range of techniques and application areas.

in each round to compute the current level, which makes it perform $O(r_{src}|V| + |E|)$ work and the same depth, which is not work-efficient.

In this thesis, we study four bucketing-based graph algorithms—k-core[6], Δ-stepping, weighted breadth-first search (wBFS), and approximate set-cover.

To provide simple and theoretically-efficient implementations of these algorithms, we design and implement a work-efficient interface for bucketing in the Ligra shared-memory graph processing framework [319]. Our extended framework, which we call ***Julienne***, enables us to write short (under 100 lines of code) implementations of the algorithms that are efficient and achieve good parallel speedup (up to 43x on 72 cores with two-way hyper-threading). Table 1.2 displays the work and depth bounds for the algorithms developed using Julienne.

The Graph Based Benchmark Suite

Building on our study of ordered graph algorithms in Chapter 4, in Chapter 5 we study how to design practical and theoretically-efficient shared-memory parallel graph algorithms for a broad set of fundamental graph problems, including some ordered graph problems. Table 1.3 lists the problems that we consider in this part of the thesis. We consider problems spanning a wide variety of domains, including different types of shortest path problems, connectivity problems such as undirected connectivity, minimum spanning forest, biconnectivity, and strong connectivity, symmetry breaking problems like maximal independent set, maximal matching, and graph coloring, amongst many other problems.

[6]The definitions of k-core and coreness (see Chapter 2) have been used interchangeably in the literature, however they are not the same problem, as pointed out in [297]. In this thesis we use k-core to refer to the coreness problem. Note that computing a particular k-core from the coreness numbers requires finding the largest induced subgraph among vertices with coreness at least k, which can be done efficiently in parallel.

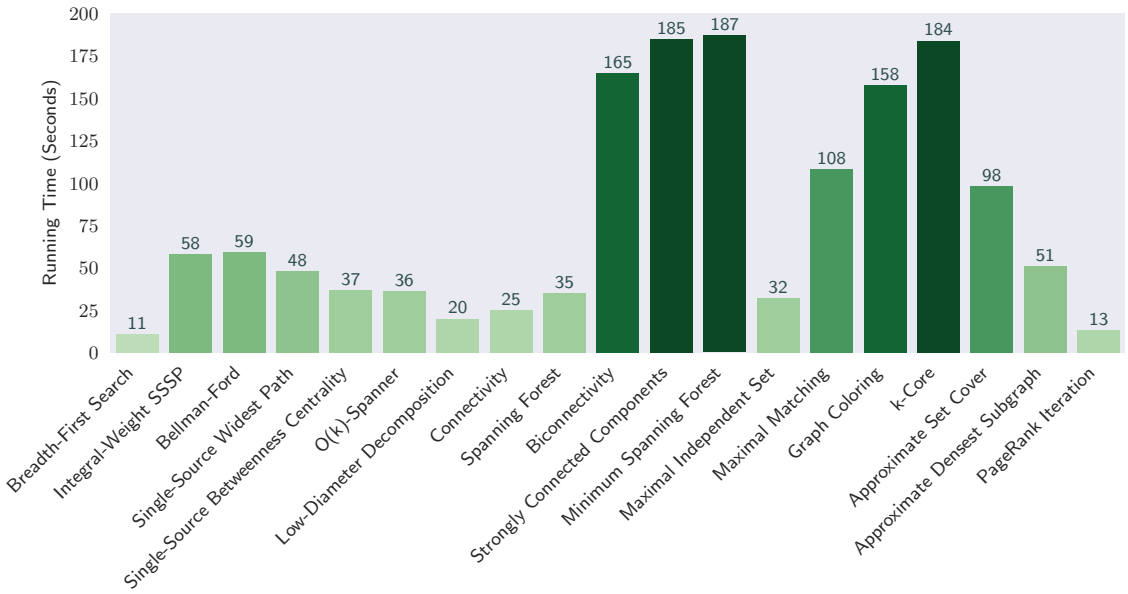


Figure 1.3: Running times of our shared-memory graph algorithm implementations on the Hyperlink Web graph, on a 72-core, 2-way hyper-threaded machine equipped with 1TB of memory. Times for 19 out of the 20 problems we consider are shown. We elide the running time for Triangle Counting, which runs in 1168 seconds on this graph (the algorithm enumerates all 9.6 trillion triangles in this graph).

We implement efficient parallel graph algorithms for all of these problems using only simple fork-join primitives, similar to those used in Cilk. We have made the contributions from this part of the thesis have been made publicly available as the Graph Based Benchmark Suite (GBBS), a problem-based benchmark suite for graph problems.[7]

We present an experimental evaluation of all of our implementations on a relatively modest multicore machine (72 cores, equipped with 1 TB of memory). We are able to apply our algorithms to the largest publicly-available graph, the Hyperlink Web graph which we discussed earlier in this introduction. Figure 1.3 shows the running times for 19 of the problems that we study on the Hyperlink Web graph (triangle counting, which runs in about 20 minutes, is excluded to make visualizing the results easier). For the remaining 19 problems, our parallel implementations all run in under 3.5 minutes on this graph, which contains over 200 billion edges.

In almost all cases, the numbers we report are faster than any previous performance numbers for any machines, even much larger supercomputers. In many cases, our results are the first time these problems have been solved on this graph or a graph of this size in the literature. Most importantly, our implementations are based on reasonably simple

[7] https://github.com/ldhulipala/gbbs

Problem	Speedup	Alg.	Work	Depth
Breadth-First Search (BFS)	68	–	$O(m)$	$O(\text{diam}(G)\lg n)$
Integral-Weight SSSP (weighted BFS)	64	[115]	$O(m)^*$	$O(\text{diam}(G)\lg n)^\dagger$
General-Weight SSSP (Bellman-Ford)	67	[108]	$O(\text{diam}(G)m)$	$O(\text{diam}(G)\lg n)$
Single-Source Widest Path (Bellman-Ford)	66	[108]	$O(\text{diam}(G)m)$	$O(\text{diam}(G)\lg n)$
Single-Source Betweenness Centrality (BC)	60	[84]	$O(m)$	$O(\text{diam}(G)\lg n)$
$O(k)$-Spanner	65	[248]	$O(m)$	$O(k\lg n)^\dagger$
Low-Diameter Decomposition (LDD)	59	[246]	$O(m)$	$O(\lg^2 n)^\dagger$
Connectivity	65	[321]	$O(m)^*$	$O(\lg^3 n)^\dagger$
Spanning Forest	67	[321]	$O(m)^*$	$O(\lg^3 n)^\dagger$
Biconnectivity	59	[346]	$O(m)^*$	$O(\text{diam}(G)\lg n + \lg^3 n)^\dagger$
Strongly Connected Components (SCC)*	43	[75]	$O(m\lg n)^*$	$O(\text{diam}(G)\lg n)^\dagger$
Minimum Spanning Forest (MSF)	50	[388]	$O(m\lg n)$	$O(\lg^2 n)$
Maximal Independent Set (MIS)	68	[64]	$O(m)^*$	$O(\lg^2 n)^\dagger$
Maximal Matching (MM)	66	[64]	$O(m)^*$	$O(\lg^2 m)^\dagger$
Graph Coloring	56	[169]	$O(m)$	$O(\lg n + L\lg\Delta)$
Approximate Set Cover	58	[67]	$O(m)^*$	$O(\lg^3 n)^\dagger$
k-core	46	[115]	$O(m)^*$	$O(\rho\lg n)^\dagger$
Approximate Densest Subgraph	73	[38]	$O(m)$	$O(\lg^2 n)$
Triangle Counting (TC)	—	[323]	$O(m^{3/2})$	$O(\lg n)$
PageRank Iteration	74	[85]	$O(n+m)$	$O(\lg n)$

Table 1.4: Parallel speedup (SU) of our algorithms on the symmetrized Hyperlink2012 graph over a single-threaded running time. Theoretical bounds for the algorithms on the TRAM are shown in the last three columns. We mark times that did not finish in 5 hours with —. *SCC was run on the directed version of the graph. †We say that an algorithm has $O(f(n))$ cost ***with high probability (whp)*** if it has $O(k\cdot f(n))$ cost with probability at least $1-1/n^k$. We assume $m=\Omega(n)$.

algorithms with strong bounds on their work and depth.

Parallel Semi-Asymmetric Graph Algorithms

In Chapter 6, we study how to design practical and theoretically-efficient parallel graph algorithms that operate on graphs stored in non-volatile memory. Non-volatile memories are a new class of memory technologies emerging on the market today (e.g., Intel's *Optane DC Peristent Memory*). These devices are significantly cheaper than DRAM on a per-gigabyte basis, provide an order of magnitude greater memory capacity per DIMM than traditional DRAM, and offer byte-addressability and low idle power, thereby providing a realistic and cost-efficient way to equip a commodity multicore machine with multiple terabytes of non-volatile RAM (NVRAM).

A significant challenge in programming for NVRAMs is to design algorithms that cope with an inherent *asymmetry* between reads and writes—write operations are more expensive than reads both in terms of energy and throughput. This property requires rethinking algorithm design and implementations to minimize the number of writes to

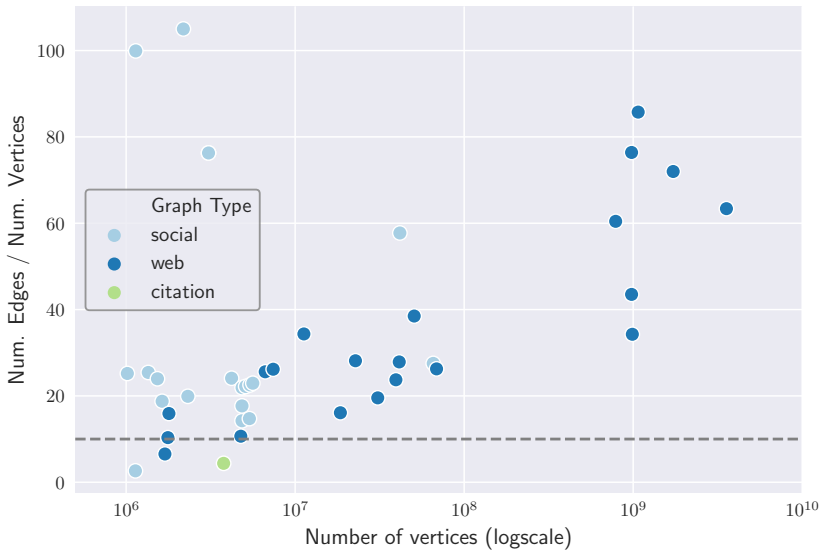


Figure 1.4: Number of vertices (logscale) vs. average degree (m/n) on 42 real-world graphs with $n > 10^6$ from the SNAP [215] and LAW [81] datasets. Over 90% of the graphs have average degree larger than 10 (corresponding to the gray dashed line).

NVRAM [76, 53, 94, 358]. Additionally, the read-bandwidth of NVRAM DIMMs is between 3–4x lower than that of DRAM, which could become a significant bottleneck for algorithms which are memory bound.

Much of the prior work on efficient NVRAM algorithms considered a setting where the system was equipped with purely NVRAM, and a limited amount of DRAM that could be used as a cache. In this thesis, we observe that in practice it is more likely that NVRAM will be used in conjunction with a reasonable amount of DRAM. For example, the experiments in this thesis are done on a 48 core machine that has 8x as much NVRAM as DRAM (and we are aware of machines with 16x as much NVRAM as DRAM [152]), where combined read throughput for all cores from the NVRAM is about 3x slower than reads from the DRAM, and writes on the NVRAM are a further factor of about 4x slower [293, 186] (a factor of 12 total). Thus, if the graph in question fits on NVRAM, and the average degree of the graph is a small constant (say larger than 16), then we can reasonably assume that there is sufficient DRAM to store data proportional to the number of vertices.

An important property of most graphs used in practice is that they are sparse, but still tend to have many more edges than vertices, often from one to two orders of magnitude more. This is true for almost all social network graphs [215], but also for many graphs that are derived from various simulations [113]. In Figure 6.1 we show that over 90% of the large graphs (more than 1 million vertices) from the SNAP [215] and LAW [81] datasets have at least 10 times as many edges as vertices. Given that very large graphs

today can have over 100 billion edges (requiring around a terabyte of storage), but only a few billion vertices, a popular and reasonable assumption both in theory and in practice is that vertices, but not edges, fit in DRAM [1, 138, 204, 236, 239, 254, 274, 339, 385, 386].

The Parallel Semi-Asymmetric Model. We formalize this idea using a new model called the Parallel Semi-Asymmetric Model (PSAM), which consists of a shared asymmetric large-memory with unbounded size that can hold the entire graph, and a shared symmetric small-memory with $O(n)$ words of memory, where n is the number of vertices in the graph. In a relaxed version of the model, we allow small-memory size of $O(n + m/\lg n)$ words, where m is the number of edges in the graph. The PSAM model permits writes to the large-memory, which are $\omega > 1$ times more costly than reads.

In this thesis, we obtain parallel graph algorithms in the PSAM with strong theoretical bounds on their work and depth in the model. Specifically, most of our algorithms are work-efficient (performing asymptotically the same work as the best sequential algorithm for the problem) and have poly-logarithmic depth (parallel time). An interesting aspect of our approach is that we do not use writes to the large-memory, which makes the cost of our algorithms independent of the underlying asymmetry, ω. Our theoretical guarantees ensure that our algorithms perform reasonably well across graphs with different characteristics, machines with different core counts, and NVRAMs with different read-write asymmetries.

Parallel Semi-Asymmetric Graph Algorithms. Our main contribution is *Sage*, a parallel semi-asymmetric graph engine with which we implement provably-efficient (and often work-optimal) algorithms for over a dozen fundamental graph problems. The key innovations are in ensuring that the updated state is associated with vertices and not edges, which is particularly challenging (i) for certain edge-based parallel graph traversals and (ii) for algorithms that "delete" edges as they go along in order to avoid revisiting them once they are no longer needed. We provide general techniques to solve these two problems. For the latter, used by four of our algorithms, we require relaxing the prescribed amount of DRAM to be on the order of one bit per edge.

We show that Sage scales to the largest publicly-available graph, the Hyperlink2012 graph with over 3.5 billion vertices and 128 billion edges (and 225 billion edges for algorithms running on the undirected/symmetrized graph). Compared with the state-of-the-art DRAM codes from GBBS [117], automatically extended to use NVRAM using Memory-Mode,[8] Sage is 1.89x faster on average. Compared with the recently developed codes from [152], which are the current state-of-the-art NVRAM codes available today, our codes are faster on all five graph problems studied in [152], and achieve an average speedup of 1.94x.

[8]Effectively using the DRAM as a cache—see Section 6.8.1.

1.2 *Parallel Batch-Dynamic Graph Algorithms*

In the second part of this thesis, we design efficient algorithms in the parallel batch-dynamic setting. We start in Chapter 7 by studying Euler tour trees, which are classic dynamic data structures that solve the dynamic trees problem, and show that a concurrent version of this data structure can be used to solve a batch-dynamic version of the dynamic trees problem. In Chapter 8 we study the batch-dynamic connectivity problem and show that the classic dynamic connectivity data structure of Holm, de Lichtenbert, and Thorup can be adapted to the parallel batch-dynamic setting, using parallel batch-dynamic Euler tour trees as a crucial sub-routine. Next, we describe both of these results in more details.

Parallel Batch-Dynamic Trees

Chapter 7 studies the parallel batch-dynamic trees problem, and designs an efficient concurrent data structure based on the Euler tour tree structure that can be used as a building block in other batch-dynamic data structures. In the dynamic trees problem, the goal is to maintain a collection of vertex-disjoint trees under *link* and *cut* operations, which respectively join two trees with a new edge, and break a single tree into two by deletion of an edge. In the batch-dynamic version of the dynamic trees problem, the objective is to maintain a forest that undergoes *batches* of link and cut operations. The queries can include whether two vertices are connected in the same tree, and also to compute the sum of an augmented value within a given subtree in the tree.

Although many sequential data structures exist to maintain dynamic trees [328, 19], the only existing batch-dynamic data structure for this problem is a recent result by Acar et al. [4]. Unfortunately, this solution requires transforming the input into a forest with bounded-degree in order to perform contractions efficiently. Obtaining a data structure without this restriction is therefore of interest. Furthermore, it is of intellectual interest whether the arguably simplest solution to the dynamic trees problem, Euler tour trees [173, 249], can be parallelized under batches of edge insertions and deletions.

Euler Tour Trees. An Euler Tour Tree (ETT) is a simple and intuitive dynamic data structure for the dynamic trees problem. The core idea is to represent trees using their Euler tours, which are sequences that intuitively "trace" the tree along its exterior face. The ETT data structure represents the Euler tour of a tree using a balanced binary tree. Links and cuts on the underlying forest can be mapped to a constant number of operations on the tours, which can be easily implemented in $O(\lg n)$ time on the binary trees representing the tours.

Parallel Batch-Dynamic Euler Tour Trees. In this chapter of the thesis, we show that ETTs can be efficiently parallelized by designing sequence data structures that can be efficiently batch *split* and batch *joined*. We show that adding and removing a batch of k edges in our data structure can be done in $O(k \lg(1+n/k))$ expected work and $O(\lg n)$ depth

Algorithm	**Single Update**		**Batch of k Updates**	
	Work	*Depth*	*Work*	*Depth*
Holm et al. [175]	$O(\lg^2 n)$	$O(m)$	$O(\lg^2 n)$	$O(m)$
This thesis	$O(\lg^2 n)^{\ddagger}$	$O(\lg^3 n)$ *whp*	$O\left(\lg n \lg\left(1+\frac{n}{\Delta}\right)\right)^{\ddagger}$	$O(\lg^3 n)$ *whp*

Table 1.5: Work and depth bounds of the parallel batch-dynamic connectivity algorithm studied in this thesis compared to the bounds obtained by Holm et al. The depth bounds are shown for the CRCW PRAM. ‡ indicates that the work bound is expected amortized per edge insertion or deletion. Δ is the average batch size of a deletion operation. The cost of connectivity queries in the classic data structure is $O(\lg n)$ per query. Our structure answers k queries in $O(k \lg(1 + n/k))$ work and $O(\lg n)$ depth. Note that the depth for processing a batch of k insertions in our structure is $O(\lg n)$ *whp*.

whp. Our work bounds match those of Acar et al. [4], and our depth bounds asymptotically improve over their results. We experimentally evaluate our new data structures and show that they achieve between 67–96 self-relative speedup on 72 cores with hyper-threading, and scale to trees with billions of vertices and edges. Our new implementations also outperform the fastest existing sequential dynamic trees data structures, such as link-cut trees and other high-performance sequential implementations, empirically.

Parallel Batch-Dynamic Connectivity

In Chapter 8 we study the parallel batch-dynamic connectivity problem, and design an efficient data structure for this problem based on the seminal dynamic connectivity data structure by Holm, de Lichtenberg, and Thorup [175]. Computing the connected components of a graph is a fundamental problem that has been studied in many different models of computation [316, 290, 25, 175]. The ***dynamic connectivity problem*** is to maintain a data structure over an n vertex undirected graph that supports insertions and deletions of edges, as well as operations which query whether two vertices are currently in the same connected component.

Our main contribution in this chapter is an efficient parallel batch-dynamic algorithm for maintaining graph connectivity. The result crucially uses the batch-dynamic ETT data structure designed in Chapter 7. Our batch-dynamic algorithm runs in $O(\lg^3 n)$ depth *whp* and achieves an improved work bound that is asymptotically faster than the Holm, de Lichtenberg, and Thorup algorithm for sufficiently large batch sizes, and is work-efficient otherwise. We note that our depth bounds hold even when processing the updates one a time, ignoring batching. Our improved work bounds are derived by a novel analysis of the work performed by the algorithm over all batches of deletions. The bounds we obtain are summarized in Table 1.5.

Our Approach. As in the sequential Holm, de Lichtenberg and Thorup (HDT) algorithm, searching for replacement edges after deleting a batch of tree edges is the most interesting part of our parallel algorithm. A natural idea for parallelizing the HDT algorithm is to simply scan all non-tree edges incident on each disconnected component in parallel. Although this approach has low depth per level, it may examine a huge number of candidate edges, but only push down a few non-replacement edges. In general, it is unable to amortize the work performed checking all candidate edges at a level to the edges that experience level decreases.

To amortize the work properly while also searching the edges in parallel we must perform a more careful exploration of the non-tree edges. Our approach is to use a *doubling* technique, in which we geometrically increase the number of non-tree edges explored as long as we have not yet found a replacement edge. We show that using the doubling technique, the work performed (and number of non-tree edges explored) is dominated by the work of the last phase, when we either find a replacement edge, or run out of non-tree edges. Our amortized work-bounds follow by a per-edge charging argument, as in the analysis of the HDT algorithm.

From this simple baseline parallel algorithm, we obtain an improved parallel algorithm with lower depth ($O(\lg^3 n)$ vs. $O(\lg^4 n)$, both *whp*), and potentially better work. The main observation is that the simple algorithm suffers due to resetting the number of edges that it considers after each step that searches for a replacement edge.

Instead, in the refined algorithm, we interleave a step that searches for replacement edges in all currently active components with a step which contracts all replacement edges found in the current search. This approach lets us keep geometrically doubling the number of non-tree edges incident to active components that we consider. Arguing that this shaves a log-factor in the depth is relatively straightforward. We provide a more subtle analysis which shows that this approach can provide an asymptotic improvement over the work performed by the HDT algorithm when the average batch size of edge deletions is sufficiently large, which may be of independent interest.

1.3 Streaming Graph Processing

The final part of this thesis addresses the challenge of designing efficient data structures to dynamically represent graphs that evolve over time, while allowing users concurrently analyzing the graph to be able to run parallel graph analytics and queries on snapshots of the graph. This part of the thesis comprises two chapters. Chapter 9 describes the design of an efficient purely-functional tree structure called a *C*-tree which enables parallelism, compression, lightweight snapshots, and good bounds for batch insertions and deletions. We show that the structure can naturally be used to represent the adjacency information of a graph. In Chapter 10 we present *Aspen*, a graph-streaming system that extends Ligra

with support for inserting or deleting sets of edges or sets of vertices. This chapter also presents an experimental evaluation of Aspen, including a comparison with existing graph-streaming systems. We show that Aspen can scale to represent and update the entire Hyperlink Web graph in the main memory of a single multicore machine.

Compressed Purely-Functional Trees

In Chapter 9 we present an efficient way to represent graphs that admits lightweight snapshots, compression, compression, and parallel batch updates. At the heart of our work is the simple idea of representing graphs using *purely-functional balanced search trees* [2, 264]. Such a representation can use a search tree over the vertices (the vertex-tree), and for each vertex store a search tree of its incident edges (an edge-tree). Because the trees are purely-functional, acquiring an immutable snapshot is as simple as acquiring a pointer to the root of the vertex-tree. Updates can then happen concurrently without affecting the snapshot. In fact, any number of readers (queries) can concurrently acquire independent snapshots without being affected by a writer. A writer can make an individual or bulk update and then set the root to make the changes immediately and atomically visible to the next reader without affecting current active readers. A single update costs $O(\lg n)$ work, and because the trees are purely-functional it is relatively easy and safe to parallelize a bulk update.

Challenges. However, there are several challenges that arise when comparing purely-functional trees to compressed sparse row (CSR), the standard data structure for representing static graphs in shared-memory graph processing [295]. In CSR, the graph is stored as an array of vertices and an array of edges, where each vertex points to the start of its edges in the edge-array. Therefore, in the CSR format, accessing all edges incident to a vertex v takes $O(deg(v))$ work, instead of $O(\lg n + deg(v))$ work for a graph represented using trees. Furthermore, the format requires only one pointer (or index) per vertex and edge, instead of a whole tree node, which has much higher overhead due to pointers for the left and right children, as well as metadata needed to maintain balance information. Additionally, as edges are stored contiguously, CSR has good cache locality when accessing the edges incident to a vertex, while tree nodes could be spread across memory. Finally, each set of edges can be compressed internally using graph compression techniques [322], allowing massive graphs to be stored using just one or two bytes per edge [117]. This approach cannot be used directly on trees. This would all seem to put a search tree representation at a severe disadvantage.

Our Data Structure. We overcome these disadvantages of purely-functional trees by designing a purely-functional tree data structure that supports structural compression. Specifically, we propose a compressed purely-functional tree data structure that we call a C-tree, which addresses the poor space usage and locality of purely-functional trees. The C-tree data structure allows us to take advantage of graph compression techniques,

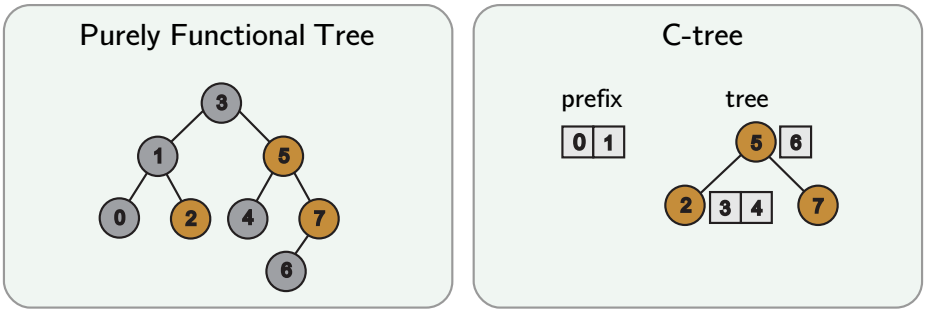


Figure 1.5: A purely functional balanced search tree, and a *C*-tree on the same set of elements.

and thereby store very large graphs on a single machine. The key idea of a *C*-tree is to chunk the elements represented by the tree and store each chunk contiguously in an array. Because elements in a chunk are stored contiguously, the structure achieves good locality. By ensuring that each chunk is large enough, we significantly reduce the space used for tree nodes. Although the idea of chunking is intuitive, designing a chunking scheme which admits asymptotically-efficient algorithms for batch-updates and also performs well in practice is challenging. We note that our chunking scheme is independent of the underlying balancing scheme used, and works for any type of element. In the context of graphs, because each chunk in a *C*-tree stores a sorted set of integers, we can compress by applying difference coding within each block and integer code the differences.

Aspen

In Chapter 10 we present *Aspen*, a multicore graph-streaming framework built using *C*-trees that enables concurrent, low-latency processing of queries and updates, along with several algorithms using the framework. The Aspen interface is an extension of the graph processing interface presented in Chapter 3. It includes the full Ligra interface—vertexSubsets, EDGEMAP, as well as other extensions of the interface described in Chapter 3. Fundamentally, the interface extends the interface with a set of functions for updating the graph—in particular, for inserting or deleting sets of edges or sets of vertices. All of the functions for processing and updating the graph work on a *fixed and immutable version (snapshot)* of the graph. The updates are functional, and therefore instead of mutating the graph, return a handle to a new graph.

1.4 Research Presented in this Thesis

The results presented in this thesis are the result of collaborative research done over the course of my graduate studies with my co-authors. These works are listed below:

- Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. "Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2017, pp. 293–304 (Chapter 3 and Chapter 4)
- Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. "Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2018, pp. 293–304 (Chapter 3 and Chapter 5)
- Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. "The Graph Based Benchmark Suite (GBBS)". in: *International Workshop on Graph Data Management Experiences and Systems (GRADES) and Network Data Analytics (NDA)*. 2020, 11:1–11:8 (Chapter 3 and Chapter 5)
- Laxman Dhulipala, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E Blelloch, Phillip B Gibbons, and Julian Shun. "Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs". In: *Proc. VLDB Endow.* 13.9 (2020), pp. 1598–1613 (Chapter 6)
- Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. "Batch-Parallel Euler Tour Trees". In: *Algorithm Engineering and Experiments (ALENEX)* (2019), pp. 92–106 (Chapter 7)
- Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. "Parallel Batch-Dynamic Graph Connectivity". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2019, pp. 381–392 (Chapter 8)
- Laxman Dhulipala, Guy E Blelloch, and Julian Shun. "Low-Latency Graph Streaming using Compressed Purely-Functional Trees". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2019, pp. 918–934 (Chapters 9 and 10)

Other Research. Several other papers written by the thesis author and collaborators during graduate school are not included in this thesis [121], but reflect on topics studied in this thesis in interesting ways. For example [121] shows that the Euler tour tree structure presented in Chapter 7 results in an efficient $O(1)$-round batch-dynamic tree structure in the Massively Parallel Computation (MPC) setting. This work also presents a different batch-dynamic connectivity algorithm for the MPC setting. Continuing the theme of searching for powerful (and implementable) parallel models [47] studies a model called the *Adaptive Massively Parallel Computation* model which augments the massively parallel model with the ability to perform a limited amount of random reads from a read-only shared memory within a round of computation. Algorithms for basic graph problems in this model achieve exponential speedups over existing MPC algorithms for various

problems. Finally, we have also developed a low round-complexity connectivity algorithm in the standard MPC setting [48].

1.5 Broader Outlook and Thesis Statement

A fundamental goal of research in theoretical computer science is to design relatively simple theoretical models of computation whose theoretical predictions can be reliably applied in practice. The task is challenging since to be applicable in a broad set of situations, the model must necessarily abstract away from many details of the hardware that it models. One of the remarkable success stories of the field is how well, broadly speaking, theoretical differences in the complexity of algorithms in different models of computation (e.g., the external-memory model [12], or cache-oblivious model [147]) translate into practical differences in performance.

In the context of parallel algorithms, theoretical research in the early days (the late 1970s and 1980s) was done in an era where Moore's law was still providing reliable speedups in single-processor performance. This fact, coupled with a diversity of parallel machine architectures at the time, led to a proliferation of theoretical models for parallel algorithms, and also theoretical results that were sometimes tailored to specific aspects of these models. Unfortunately, probably due to the relative difficulty of programming and performing experiments on parallel machines at the time, most of the theoretical work on parallel graph algorithms from these early years was not experimentally evaluated.

Today, we are living in a very different era of parallel computing. Single-core performance has plateaued with the end of Moore's law, and chip designers have turned to multicore parallelism for performance. This shift has placed a burden on algorithm designers, since obtaining speedups today requires rethinking algorithms to take advantage of parallelism. Furthermore, chips today only provide a modest number of general-purpose cores, and there are significant energy-efficiency challenges in the way of obtaining chips with 1000s of general-purpose cores [136].[9] Fortunately, multicore machines today are cheap and easy to obtain—it has never been easier to design, implement, and evaluate shared-memory parallel algorithms.

But what parallel algorithms should we implement to observe good performance and reliable speedups today, and in what kind of environments should these algorithms be implemented to achieve good practical performance on massive datasets and low cost? As an example of challenges faced by a practitioner trying to develop practical and cost-efficient algorithm implementations, consider a person trying to solve the connected components problem. A quick search through the parallel computing literature for algorithms solving

[9]Note that GPUs can provide 1000s of lightweight cores, but are challenging to apply when solving large-scale graph problems due to the small amount of memory that they can address, and the irregular memory access patterns present in most graph algorithms.

connected components reveals dozens of algorithms that have been proposed over the past 40 years, some of which are work-efficient, and others which are slightly work-inefficient. Daunted, they may try looking at practical systems for graph processing to see what kind of connected components algorithms these systems implement. However, these practical frameworks for parallel graph processing, such as Giraph and Apache Spark, by and large implement very simple algorithms for this problem, and implementing more complex algorithms from the literature is challenging due to the limited set of algorithmic primitives provided by these frameworks.

Our Study of Parallel Graph Algorithms

This thesis addresses this problem by presenting a comprehensive study of parallel graph algorithms, covering a much broader set of problems and algorithms than any other study of parallel graph algorithms in the literature. One of the main findings of this thesis is that algorithms which are theoretically-efficient (and often work-efficient) in a theoretical model called the *binary-forking model* are also practically efficient. Our study builds a set of parallel graph algorithms, many of which are based on classic results in the parallel computing literature that are originally designed for PRAMs, which are practical on modern shared-memory machines, and scale to the largest publicly available graphs using only a reasonably modest multicore machine.

One advantage of performing a study like the one in this thesis, is that we can identify similarities between different algorithms, and extract these common idioms and techniques into reusable, high-performance components. For example, at first glance, the peeling algorithm for computing the coreness of every vertex (the largest non-empty k-core containing a vertex), and the classic Δ-stepping algorithm for parallel shortest paths on positive weight graphs would seem to have nothing in common, but a closer look reveals that both algorithms rely on dynamically storing vertices in sets of buckets for efficiency. In k-core, the buckets represent the current upper-bound on the vertex's coreness, whereas in Δ-stepping, the buckets represent a tentative shortest-path distance between the source vertex and a given vertex. By providing a high-level interface for bucketing (Chapter 4) we are able to (i) design a reusable, high performance implementation of the interface and (ii) apply the interface to implement other algorithms that can be expressed using bucketing, such as parallel approximate set-cover [67].

Similarly, the study of parallel graph algorithms we present in Chapter 5, in addition to showing the capabilities of shared-memory graph processing to quickly solve a very broad set of fundamental problems on massive real-world graphs on a single machine, also identifies a general interface for shared-memory graph processing that the thesis author believes may be useful in implementing other parallel graph algorithms in the future. For example, many graph algorithms employ the following 'reduction' idiom: given a subset of vertices U, for each vertex v neighboring U compute the number of edges incident to

v where the other endpoint is contained in U. Similarly, some algorithms (e.g., certain connectivity algorithms) contract a graph given a labeling of the vertices to obtain a new graph, and others require filtering a sets of edges from the graph either for correctness, or to improve work-efficiency.

Implementing algorithms using these idioms in existing frameworks and systems is challenging because the frameworks have not been designed to enable the user to perform these more complicated tasks. Thus, one of the important contributions of our study is a much richer graph interface than interfaces in existing systems, which also has clear parallel cost bounds on all of its primitives. Using the interface enables clear and relatively concise expression of all of the algorithms studied in this thesis.

Parallel Batch-Dynamic Algorithms and Streaming Graph Processing

Both parallel batch-dynamic algorithms, and streaming graph processing are relatively new areas of research that we study in this thesis (at least relative to the study of parallel graph algorithms). Despite these areas still being in their infancy, there has been a large amount of interest in both areas due to the enormous practical applicability of being able to represent an up-to-date copy of a graph and analyze it in nearly real-time.

The second part of this thesis presents a study of parallel algorithms for two fundamental problems in the batch-dynamic setting: the dynamic trees problem, and the dynamic connectivity problem. Both algorithms, for the batch-dynamic trees problem (Chapter 7) and the batch-dynamic connectivity problem (Chapter 8) are work-efficient relative to their sequential dynamic algorithm counterparts. Our results provide positive evidence for the possibility of designing theoretically-efficient parallel batch-dynamic algorithms in a shared-memory setting. Furthermore, our experimental results provide evidence that the approach is practical and can scale to high update rates on a modest multicore machine.

Since the work presented in Chapters 7 and Chapter 8 appeared in their respective venues, more work on parallel batch-dynamic algorithms has begun to emerge both in shared-memory models, as well as more powerful models such as the Massively Parallel Model [263] and the k-Machine Model [151]. Although these results in more powerful models are of clear intellectual interest, their usefulness in practice still needs and deserves to be experimentally studied. On the other hand, there is a clear path to implementing and experimentally evaluating more shared-memory parallel batch-dynamic algorithms like the algorithms evaluated in Chapter 7, and comparing these algorithms to strong static shared-memory baselines like those in Chapters 4–6.

The final part of this thesis studies the graph-streaming setting and presents a new purely-functional compressed tree structure called a C-tree, and a graph-streaming system called Aspen based on this data structure. Central to our results in this part of the thesis is the idea of using tree data structures that are carefully designed for parallelism in a shared-memory setting [63, 341]. Our work builds on this prior work to show that carefully

designed parallel batch update algorithms can achieve both strong theoretical bounds on the update costs, as well as achieve update rates of hundreds of millions of edges per second on a relatively modest multicore machine. Our results provide the first dynamic graph representation that admits efficient parallel batch updates, enjoys low memory usage due to compression, and enables lightweight snapshots.

Based on these results, this thesis presents the following thesis statement.

Thesis Statement: *This thesis contends that shared-memory algorithms can serve as the foundation of a graph processing toolkit for static and evolving graphs that is affordable, practical, scalable, and provably-efficient.*

2 Preliminaries and Notation

2.1 Notation

Graph Notation

We use the following notation to describe graphs, and related objects throughout this thesis. We denote an unweighted graph by $G(V, E)$, where V is the set of vertices and E is the set of edges in the graph. A weighted graph is denoted by $G = (V, E, w)$, where w is a function which maps an edge to a real value (its weight). The number of vertices in a graph is $n = |V|$, and the number of edges is $m = |E|$. Vertices are assumed to be indexed from 0 to $n - 1$. We call these unique integer identifiers for vertices *vertex IDs*.

For undirected graphs we use $N(v)$ to denote the neighbors of vertex v and $d(v)$ to denote its degree. For directed graphs, we use $N^-(v)$ and $N^+(v)$ to denote the in-neighbors and out-neighbors of a vertex v, and $d^-(v)$ and $d^+(v)$ to denote its in-neighbors and out-degree, respectively. We assume that there are no self-edges or duplicate edges in the graph. We refer to graphs stored as a list of edges as being stored in the ***edgelist*** format and the compressed-sparse row formats as ***CSR*** format. In the edgelist format, we are given an array of pairs (u, v) corresponding to the endpoints of the edges. In CSR, we are given two arrays, I and A, where the incident edges of a vertex v are stored in $\{A[I[v]], \ldots, A[l[v + 1] - 1]\}$.[1]

Other Notation. We say that an algorithm has $O(f(n))$ cost ***with high probability (whp)*** if it has $O(k \cdot f(n))$ cost with probability at least $1 - 1/n^k$.

Definitions used in Graph Algorithms

Diameter, Eccentricity. We use $\mathrm{diam}(G)$ to refer to the ***diameter*** of the graph, or the longest shortest path distance between any vertex s and any vertex v reachable from s. We use r_s to denote the ***eccentricity*** of a vertex s, or longest shortest path distance between a vertex s and any vertex v reachable from s. Note that the diameter is simply the maximum r_s for any $s \in V$.

Density, k-Cores, and Coreness. Given an undirected graph $G = (V, E)$ the ***density*** of a set $S \subseteq V$, or $\mathcal{D}(S)$, is equal to $\frac{|E(S)|}{|S|}$ where $E(S)$ are the edges in the induced subgraph on S. Δ is used to denote the maximum degree of the graph. A ***k-core*** of an undirected graph G is a maximal connected subgraph where every vertex has induced-degree at least

[1] We assume $A[n] = m$.

k. The ***coreness*** of a vertex, v is the largest non-empty k-core containing the vertex. A parallel algorithm to compute all k-cores of the graph works as follows. In each *peeling* step, the algorithm removes all vertices with degree equal to the current minimum degree in the graph. This process repeats until all vertices have been removed from the graph. We define ρ to be the ***peeling-complexity*** of a graph, which is the number of steps required to peel the graph to an empty graph.

Betweenness Centrality and Dependency Values. Define σ_{st} to be the total number of s–t shortest paths, $\sigma_{st}(v)$ to be the number of s–t shortest paths that pass through v, and $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$ to be the ***pair-dependency*** of s and t on v. [2] The ***betweenness centrality*** of a vertex v is equal to $\sum_{s \neq v \neq t \in V} \delta_{st}(v)$, i.e. the sum of pair-dependencies of shortest-paths passing through v. Brandes [84] proposes an algorithm to compute the betweenness centrality values based on the following notion of 'dependencies': the ***dependency*** of a vertex r on a vertex v is $\delta_r(v) = \sum_{t \in V} \delta_{rt}(v)$. In this thesis, we consider the single-source betweenness centrality, which is to compute the dependency values for each vertex from a source vertex.

2.2 Atomic Operations

We use four common atomic primitives in our algorithms and implementations: testAndSet (TS), compareAndSwap (CAS), fetchAndAdd (FA), and priorityWrite (PW). A **testAndSet**($\&x$) checks if x is false, and if so atomically sets it to true and returns true; otherwise it returns false. A **compareAndSwap**($\&x$, *oldV*, *newV*) takes three arguments—a memory location x, an old value *oldV* and a new value *newV*. If the value stored at x is equal to *oldV*, the compareAndSwap atomically updates the value at x to be *newV* and returns *true*; otherwise the CAS returns *false*. compareAndSwap is supported in almost all modern processors. A **fetchAndAdd** ($\&x$) atomically returns the current value of x and then increments x. A **priorityWrite**($\&x, v, p$) atomically compares v with the current value of x using the priority function p, and if v has higher priority than the value of x according to p it sets x to v and returns true; otherwise it returns false.

2.3 Parallel Model and Cost

In the analysis of algorithms in this thesis we primarily use a work-depth model which is closely related to the PRAM but better models current machines and programming paradigms that are asynchronous and allows dynamic forking. We can simulate this model on the CRCW PRAM equipped with the same operations with an additional $O(\lg^* n)$ factor

[2] Note that $\sigma_{st}(v) = 0$ if $v \in \{s, t\}$.

in the depth *whp* due to load-balancing [72]. Furthermore, a PRAM algorithm using P processors and T time can be simulated in our model with PT work and T depth.

Model

The ***Binary-Forking Model*** [62, 72] consists of a set of threads that share an unbounded memory. Each thread is basically equivalent to a Random Access Machine—it works on a program stored in memory, has a constant number of registers, and has standard RAM instructions (including an `end` to finish the computation). The binary-forking model extends the RAM with a `fork` instruction that forks 2 new child threads. Each child thread receives a unique integer in the range $[1, 2]$ in its first register and otherwise has the identical state as the parent, which has a 0 in that register. They all start by running the next instruction. When a thread performs a `fork`, it is suspended until both of its children terminate (execute an `end` instruction). A computation starts with a single root thread and finishes when that root thread ends. Processes can perform reads and writes to the shared memory, as well as the TESTANDSET instruction. This model supports what is often referred to as nested parallelism. If the root thread never does a fork, it is a standard sequential program.

Work and Depth

A computation can be viewed as a series-parallel DAG in which each instruction is a vertex, sequential instructions are composed in series, and the forked threads are composed in parallel. The ***work*** of a computation is the number of vertices and the ***depth*** is the length of the longest path in the DAG. As is standard with the RAM model, we assume that the memory locations and registers have at most $O(\lg M)$ bits, where M is the total size of the memory used.

Model Variants

In this thesis, we augment the binary-forking model described above with two atomic instructions that are used by our algorithms: FETCHANDADD and PRIORITYWRITE and discuss the model with these instruction as the FA-BF, and PW-BF variants of the binary-forking model, respectively. We abbreviate the basic binary-forking model with only the TESTANDSET instruction as the BF ***model***. Note that the basic binary-forking model includes a TESTANDSET, as this instruction is necessary to implement joining tasks in a parallel schedulers (see for example [77, 26]), and since all modern multicore architectures include the TESTANDSET instruction.

2.4 Parallel Primitives

The following parallel procedures are used throughout this thesis. We first define the concept of a monoid, which is used in several primitives.

A ***monoid*** over a type E is an object consisting of an associative function $\oplus : \mathsf{E} \times \mathsf{E} \to \mathsf{E}$, and an identity element $\bot : \mathsf{E}$. A monoid is specified as a pair, $(\bot, \oplus)$.

The ***scan*** operation takes as input an array A of length n, and a monoid $(\bot, \oplus)$, and returns the array $(\bot, \bot \oplus A[0], \bot \oplus A[0] \oplus A[1], \ldots, \bot \oplus_{i=0}^{n-2} A[i])$ as well as the overall sum, $\bot \oplus_{i=0}^{n-1} A[i]$. Scan can be done in $O(n)$ work and $O(\lg n)$ depth (assuming $\oplus$ takes $O(1)$ work) [188].

The ***reduce*** operation takes an array A and a monoid $(\bot, \oplus)$ and returns the sum of the elements in A with respect to $\oplus$. Reduce can be done in $O(n)$ work and $O(\lg n)$ depth (assuming $\oplus$ takes $O(1)$ work)

A ***filter*** takes an array A and a predicate f and returns a new array containing $a \in A$ for which $f(a)$ is true, in the same order as in A. Filter can be done in $O(n)$ work and $O(\lg n)$ depth (assuming f takes $O(1)$ work).

A ***semisort*** takes an input array of elements, where each element has an associated key and reorders the elements so that elements with equal keys are contiguous, but elements with different keys are not necessarily ordered. The purpose is to collect equal keys together, rather than sort them. Semisorting a sequence of length n can be performed in $O(n)$ expected work and $O(\lg n)$ depth *whp* assuming access to a uniformly random hash function mapping keys to integers in the range $\left[1, n^{O(1)}\right]$ [288, 161].

A ***parallel dictionary*** data structure supports batch insertion, batch deletion, and batch lookups of elements from some universe with hashing. Gil et al. describe a parallel dictionary that uses linear space and achieves $O(k)$ work and $O(\lg^* k)$ depth *whp* for a batch of k operations in the CRCW PRAM [150]. This hashing algorithm can be simulated in the binary-forking model in the same work and $O(\lg k \lg^* k)$ depth *whp*. Instead, in the BF model, parallel hashing can be done in $O(k)$ expected work, and $O(\lg k)$ depth *whp* by inserting elements using linear-probing into an open-addressed table using the TESTANDSET primitive provided by the model to atomically acquire a cell.

The ***pack*** operation takes an n-length sequence A and an n-length sequence B of booleans as input. The output is a sequence A' of all the elements $a \in A$ such that the corresponding entry in B is *true*. The elements of A' appear in the same order that they appear in A. Packing can be implemented in $O(n)$ work and $O(\lg n)$ depth [188].

Finally, the ***PointerJump*** primitive takes an array P of parent pointers which represent a directed rooted forest (i.e., $P[v]$ is the parent of vertex v) and returns an array R where $R[v]$ is the root of the directed tree containing v. This primitive can be implemented in $O(n)$ work, and $O(\lg n)$ depth *whp* in the binary-forking model [72].

2.5 Pseudocode Conventions

The pseudocode for many of the algorithms in this thesis make use of the graph processing interface described in Chapter 3, as well as the atomic primitives compareAndSwap, testAndSet, fetchAndAdd, and priorityWrite (defined in Section 2.2). In our pseudocode, we use _ as a wildcard to bind values that are not used. We use anonymous functions in the pseudocode for consciseness, and adopt a syntax similar to how anonymous functions are defined in the ML language. An anonymous function is introduced using the **fn** keyword. For example,

$$\textbf{fn}\ (u, v, w_{uv}) : \text{edge} \rightarrow \textbf{return}\ \mathit{Rank}[v]$$

is an anonymous function taking a triple representing an edge, and returning the *Rank* of the vertex v. We drop type annotations when the argument types are clear from context. The option type, **E option**, provides a distinction between some value of type E (**Some(*e*)**) and no value (**None**). We use the array initializer notation $A[0, \ldots, e) = \mathit{value}$ to denote an array consisting of e elements all initialized to *value* in parallel. We use standard functional sequence primitives, such as map and filter on arrays. Assuming that the user-defined map and filter functions cost $O(1)$ work to apply, these primitives cost $O(n)$ work and $O(\lg n)$ depth on a sequence of length n. We use the syntax $\forall i \in [s, e)$ as shorthand for a parallel loop over the indices $[s, \ldots, e)$. For example, $\forall i \in [0, e), A[i] = i \cdot A[i]$ updates the i'th value of $A[i]$ to $i \cdot A[i]$ in parallel for $0 \leq i < e$.

2.6 Ligra and Ligra+

We make use of the Ligra and Ligra+ frameworks for shared-memory graph processing in this thesis and review components from these frameworks here [319, 322]. Ligra provides data structures for representing a graph $G = (V, E)$, vertexSubsets (subsets of the vertices). We make use of the edgeMap function provided by Ligra, which we use for mapping over edges. edgeMap takes as input a graph $G(V, E)$, a vertexSubset U, and two boolean functions F and C. edgeMap applies F to $(u, v) \in E$ such that $u \in U$ and $C(v) = \mathit{true}$ (call this subset of edges E_a), and returns a vertexSubset U' where $u \in U'$ if and only if $(u, v) \in E_a$ and $F(u, v) = \mathit{true}$. F can side-effect data structures associated with the vertices. edgeMap runs in $O(\sum_{u \in U} \mathit{deg}(u))$ work and $O(\lg n)$ depth assuming F and C take $O(1)$ work. edgeMap either applies a ***sparse*** or ***dense*** method based on the number of edges incident to the current frontier. Both methods run in $O(\sum_{u \in U} \mathit{deg}(u))$ work and $O(\lg n)$ depth. We note that in our experiments we use an optimized version of the dense method which examines in-edges sequentially and stops once C returns *false*. This optimization lets us potentially examine significantly fewer edges than the $O(\lg n)$ depth version, but at the cost of $O(\mathit{in\text{-}deg}(v))$ depth.

Graph Dataset	Num. Vertices	Num. Edges	diam	ρ	$k_{\max}$
LiveJournal	4,847,571	68,993,773	16	~	~
LiveJournal-Sym	4,847,571	85,702,474	20	3480	372
com-Orkut	3,072,627	234,370,166	9	5,667	253
Twitter	41,652,231	1,468,365,182	65*	~	~
Twitter-Sym	41,652,231	2,405,026,092	23*	14,963	2488
3D-Torus	1,000,000,000	6,000,000,000	1500*	1	6
ClueWeb	978,408,098	42,574,107,469	821*	~	~
ClueWeb-Sym	978,408,098	74,744,358,622	132*	106,819	4244
Hyperlink2014	1,724,573,718	64,422,807,961	793*	~	~
Hyperlink2014-Sym	1,724,573,718	124,141,874,032	207*	58,711	4160
Hyperlink2012	3,563,602,789	128,736,914,167	5275*	~	~
Hyperlink2012-Sym	3,563,602,789	225,840,663,232	331*	130,728	10565

Table 2.1: Graph inputs studied in this thesis, including vertices and edges. diam is the diameter of the graph. For undirected graphs, ρ and $k_{\max}$ are the number of peeling rounds, and the largest non-empty core (degeneracy). We mark diam values where we are unable to calculate the exact diameter with * and report the effective diameter observed during our experiments, which is a lower bound on the actual diameter.

2.7 Shared-Memory Experimental Setup

The shared-memory experiments reported in this thesis are run on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4 × 2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use simple parallel primitives such as **fork**, **join**, and a parallel for-loop, **parallel-for** to express parallelism. Unless otherwise specified, all of our programs are compiled with the `g++` compiler (version 7.3.0) with the `-O3` flag.

By using a work-stealing scheduler, like the one implemented in Cilk, we are able obtain an expected running time of $W/P + O(D)$ for an algorithm with W work and D depth on P processors [77]. We note that some of our programs use a work-stealing scheduler that we implemented, based on the algorithm of Arora et al. [26]. In this thesis we will always make it clear what parallel scheduler we use. For the parallel experiments, we use the command `numactl -i all` to balance the memory allocations across the sockets. All of the speedup numbers we report are the running times of our parallel implementation on 72-cores with hyper-threading over the running time of the implementation on a single thread.

2.8 Graphs Studied in this Thesis

The experiments reported in this thesis are run on a broad set of real-world graphs, which we describe in this section. Most of the graphs we study are Web graphs and social networks—low diameter graphs that are frequently used in practice. To test some of our algorithms on large diameter graphs, we also use road networks, and 3-dimensional tori where each vertex is connected to its 2 neighbors in each dimension.

Unweighted Graphs. We list the graphs used in our experiments, along with their size, approximate diameter, peeling complexity (ρ), and degeneracy (for undirected graphs) in Table 2.1. ***LiveJournal*** is a directed graph of the social network obtained from a snapshot in 2008 [81]. ***com-Orkut*** is an undirected graph of the Orkut social network. ***Twitter*** is a directed graph of the Twitter network, where edges represent the follower relationship [208]. ***ClueWeb*** is a Web graph from the Lemur project at CMU [81]. ***Hyperlink2012*** and ***Hyperlink2014*** are directed hyperlink graphs obtained from the WebDataCommons dataset where nodes represent web pages [241]. The Hyperlink2012 graph is the same as the WebDataCommons hyperlink graph mentioned throughout this thesis. ***3D-Torus*** is a 3-dimensional torus with 1B vertices and 6B edges. We mark symmetric (undirected) versions of the directed graphs with the suffix -Sym.

Weighted Graphs. Some of our algorithms require weighted graphs. Unfortunately there are no realistic large-scale weighted networks available today, to the best of our knowledge. Thus, we create weighted graphs for evaluating our algorithms by selecting edge weights between $[1, \lg n)$ uniformly at random.

2.9 Problem Definitions

In this section we describe I/O specifications of the graph problems considered in this thesis.

2.9.1 Shortest Path Problems

Breadth-First Search (BFS).
Input: $G = (V, E)$, an unweighted graph, $src \in V$.
Output: D, a mapping containing the distance between src and vertex in V. Specifically,

- $D[src] = 0$,
- $D[v] = \infty$ if v is unreachable from src, and
- $D[v] = \text{dist}_G(src, v)$, i.e., the shortest path distance in G between src and v.

Integral-Weight SSSP (weighted BFS).
Input: $G = (V, E, w)$, a weighted graph with integral edge weights, $src \in V$.
Output: D, a mapping where $D[v]$ is the shortest path distance from src to v in G. $D[v] = \infty$ if v is unreachable.

General-Weight SSSP (Bellman-Ford).
Input: $G = (V, E, w)$, a weighted graph, $src \in V$.
Output: D, a mapping where $D[v]$ is the shortest path distance from src to v in G. $D[v] = \infty$ if v is unreachable. If the graph contains any negative-weight cycles reachable from src, the vertices of these negative-weight cycles and vertices reachable from them must have a distance of $-\infty$.

Single-Source Betweenness Centrality (BC).
Input: $G = (V, E)$, an undirected graph, $src \in V$.
Output: D, a mapping from each vertex v to the dependency value of this vertex with respect to src.

Section 2.1 provides the definition of dependencies.

Widest Path.
Input: $G = (V, E, w)$, a weighted graph with integral edge weights, $src \in V$.
Output: D, a mapping where $D[v]$ is the maximum over all paths between src and v in G of the minimum weight on the path. $D[v] = \infty$ if v is unreachable.

$O(k)$-Spanner.
Input: $G = (V, E)$, an undirected, unweighted graph, and an integer stretch factor, k.
Output: $H \subseteq E$, a set of edges such that for every $u, v \in V$ connected in G, $\text{dist}_H(u, v) \leq O(k) \cdot \text{dist}_G(u, v)$.

2.9.2 Connectivity Problems

Low-Diameter Decomposition.
Input: $G = (V, E)$, a directed graph, $0 < \beta < 1$.
Output: $\mathcal{L}$, a mapping from each vertex to a cluster ID representing a $(O(\beta), O((\lg n)/\beta))$ decomposition. A (β, d)-decomposition partitions V into $C_1, \ldots, C_k$ such that:

- The shortest path between two vertices in C_i using only vertices in C_i is at most d.
- The number of edges (u, v) where $u \in C_i, v \in C_j, j \neq i$ is at most βm.

Connectivity.
Input: $G = (V, E)$, an undirected graph.
Output: $\mathcal{L}$, a mapping from each vertex to a unique label for its connected component.

Spanning Forest.
Input: $G = (V, E)$, an undirected graph.
Output: T, a set of edges representing a spanning forest of G.

Biconnectivity.
Input: $G = (V, E)$, an undirected graph.
Output: $\mathcal{L}$, a mapping from each edge to the label of its biconnected component.

Minimum Spanning Forest.
Input: $G = (V, E, w)$, a weighted graph.
Output: T, a set of edges representing a minimum spanning forest of G.

Strongly Connected Components.
Input: $G(V, E)$, a directed graph.
Output: $\mathcal{L}$, a mapping from each vertex to the label of its strongly connected component.

2.9.3 Covering Problems

Maximal Independent Set.
Input: $G = (V, E)$, an undirected graph.
Output: $U \subseteq V$, a set of vertices such that no two vertices in U are neighbors and all vertices in $V \setminus U$ have a neighbor in U.

Maximal Matching.
Input: $G = (V, E)$, an undirected graph.
Output: $E' \subseteq E$, a set of edges such that no two edges in E' share an endpoint and all edges in $E \setminus E'$ share an endpoint with some edge in E'.

Graph Coloring.
Input: $G = (V, E)$, an undirected graph.
Output: C, a mapping from each vertex to a color such that for each edge $(u, v) \in E$, $C(u) \neq C(v)$, using at most $\Delta + 1$ colors.

Approximate Set Cover.
Input: $G = (V = (S, E), A)$, an undirected bipartite graph representing an unweighted set cover instance.
Output: $S' \subseteq S$, a set of sets such that $\cup_{s \in S'} N(s) = E$, and $|S'|$ is an $O(\lg n)$-approximation to the optimal cover.

2.9.4 Substructure Problems

k-core.
Input: $G = (V, E)$, an undirected graph.
Output: D, a mapping from each vertex to its coreness value. Section 2.1 provides the definition of a k-core and the coreness value of a vertex.

Approximate Densest Subgraph.
Input: $G = (V, E)$, an undirected graph, and a parameter ϵ.
Output: $U \subseteq V$, a set of vertices such that the density of G_U is a $2(1 + \epsilon)$ approximation of the density of the densest subgraph of G. Section 2.1 provides the definition of the density of a subset of vertices.

Triangle Counting.
Input: $G = (V, E)$, an undirected graph.
Output: T_G, the total number of triangles in G. Each (u, v, w) triangle is counted exactly once.

2.9.5 Eigenvector Problems

PageRank.
Input: $G = (V, E)$, an undirected graph.
Output: $\mathcal{P}$, a mapping from each vertex to its PageRank value after a single iteration of PageRank.

Part I

Shared-Memory Graph Processing

Introduction

This part of the thesis introduces interfaces, techniques, and algorithms for efficiently solving a broad set of fundamental graph problems on very large real-world graphs. Chapter 3 introduces a high-level graph processing interface that extends the Ligra framework. The interface enables easily describing efficient algorithms for a broad set of graph problems, many of which have not been implemented in existing parallel graph processing frameworks. Chapter 4 studies efficient algorithms for parallel bucketing, which is an important part of the graph programming developed in this thesis. The bucketing interface enables a class of bucketing-based graph algorithms which dynamically maintain a mapping between vertices and a set of buckets, and iteratively process the buckets in a certain order. Examples of bucketing-based algorithms include the work-efficient parallel set cover algorithm by Blelloch et al. [67], a work-efficient k-core algorithm designed in the chapter, and parallel algorithms for positive and integer-weight shortest paths. Chapter 5 demonstrates the effectiveness of the interface by describing parallel algorithms (which are often work-efficient) for 20 fundamental graph problems, ranging from connectivity problems such as spanning forest, biconnectivity, and strongly connected components, to substructure problems, such as triangle counting and approximate densest subgraph, which have been open-sourced as part of a publicly-available benchmark suite called *Graph Based Benchmark Suite (GBBS)*. This chapter also presents an evaluation of the algorithm implementations on a collection of large real-world graphs, including the WebDataCommons hyperlink graph, the largest publicly available graph with over 200 billion edges. The results show that a shared-memory approach significantly outperforms existing results in the literature for the WebDataCommons graph and other very large graphs both in terms of running time, and in terms of cost. Chapter 6 extends the results in this part of the thesis to the non-volatile memory setting. The chapter first introduces a parallel model called the Parallel Semi-Asymmetric Model (PSAM), which extends the binary-forking model to separately charge accesses to DRAM and NVRAM. It then presents *Sage*, a graph processing system designed for the PSAM that incorporates new techniques for designing efficient parallel graph algorithms in this setting. The chapter experimentally evaluates Sage algorithms on large graphs, including the WebDataCommons hyperlink graph and finds that Sage algorithms outperform existing state-of-the-art results in the non-volatile memory setting, and achieve performance close to the fastest shared-memory results.

The results in this part of the thesis have appeared in the following publications:

- Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. “Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing”. In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2017, pp. 293–304

- Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. "Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2018, pp. 293–304
- Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. "The Graph Based Benchmark Suite (GBBS)". in: *International Workshop on Graph Data Management Experiences and Systems (GRADES) and Network Data Analytics (NDA)*. 2020, 11:1–11:8
- Laxman Dhulipala, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E Blelloch, Phillip B Gibbons, and Julian Shun. "Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs". In: *Proc. VLDB Endow.* 13.9 (2020), pp. 1598–1613

3 An Interface for Graph Algorithms

So, the real work of any process of design lies in this task of making up the language, from which you can later generate the one particular design.

Christopher Alexander, *The Timeless Way of Building*

3.1 Introduction

In this chapter we describe a high-level graph processing interface that is used to design and implement the algorithm implementations developed in this part of the thesis. The interface extends the Ligra and Ligra+ frameworks with additional parallel primitives that are required to efficiently implement a broad set of parallel graph algorithms. For example, we design an interface for *parallel bucketing*, which is necessary to conciscely and efficiently express certain algorithms that maintain dynamic mappings between vertices and a set of buckets. Our interface also includes primitives that make it easy to implement common tasks in graph algorithms, such as performing functional operations (maps, reductions, extracting edges matching a certain predicate) over vertex neighborhoods, performing histograms over the edges incident to a subset of vertices, and other tasks, which are difficult to express in existing graph processing frameworks and interfaces.

The primitives provided by the interface are all well-suited for parallelism, and admit provable bounds on their work and depth. In this chapter, we also provide a high-level description of how the interface can be implemented, and describes the overall system architecture of the static parallel graph processing systems developed in this thesis. This chapter serves as a prelude for the subsequent chapters, which provide a deeper investigation of how to implement parts of the interface, like the bucketing interface (Chapter 4). Finally, Chapters 5 and 6 utilize the interface and experimentally evaluate the approach in a shared-memory and non-volatile memory setting.

The remainder of this chapter describes the following content:

- The different types of graph representations supported by the interface (Section 3.3).
- Interfaces for representing subsets of vertices (vertexSubset from the Ligra framework) in Section 3.4, for parallel bucketing (Section 3.5), for vertices (Section 3.6), and for graphs (Section 3.7).

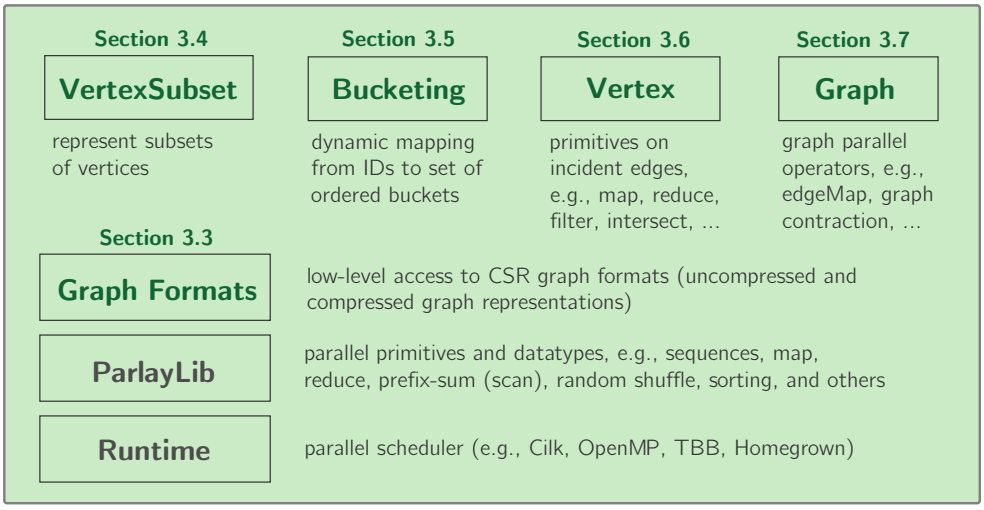


Figure 3.1: System architecture of the graph processing interface used in this thesis. The core interfaces are the vertexSubset (Section 3.4), *bucketing* (Section 3.5), *vertex* (Section 3.6), and *graph* interfaces (Section 3.7). These interfaces utilize parallel primitives and routines from ParlayLib [61]. Parallelism is implemented using a parallel runtime system—Cilk, OpenMP, TBB, or a homegrown scheduler based on the Arora-Blumofe-Plaxton deque [26] that we implemented ourselves—and can be swapped using a command line argument. The vertex and graph interfaces use a compression library that mediates access to the underlying graph, which can either be compressed or uncompressed (see Section 3.3).

3.2 Interface Overview

The interface is built as a number of layers, which we illustrate in Figure 3.1.[1] In the interface, the graph can either be stored in shared-memory, or in non-volatile memory. Chapters 4 and 5 store the graph in shared-memory, and Chapter 6 store the graph in non-volatile memory, which we discuss later in that chapter.

The implementations developed in this thesis based on this interface exploit nested parallelism using scheduler-agnostic parallel primitives, such as ***fork-join*** and parallel-for loops. Thus, they can easily be compiled to use different parallel runtimes such as Cilk, OpenMP, TBB, and also a custom work-stealing scheduler implemented by the authors. Theoretically, our algorithms are implemented and analyzed in the binary-forking model [72], which we described in Section 2.3. Our interface makes use of several new types which are defined in Table 3.1. We also define these types when they are first used in the text.

[1] A brief version of this interface was presented in [124].

Type Name	Definition	Found In
unit	An empty tuple indicating a trivial value (similar to void in languages like C)	—
E option	Either a value of type E (Some(e : E)) or no value (None)	—
E monoid	A pair of an identity element, $\perp$: E, and an associative function, $\oplus$: E $\times$ E $\rightarrow$ E	Section 2.4
E sequence	A parallel sequence containing values of type E	Section 2.4
A $\rightarrow$ B	A function with arguments of type A with results of type B	—
vtxid	A vertex ID (unique integer identifiers for vertices)	Section 2.1
vertexSubset	Data type representing a subset of vertex IDs	Section 3.4
E vertexSubset	An augmented vertexSubset (each vertex ID has an associated value of type E)	Section 3.4
vset	Abbreviation for a vertexSubset	Section 3.4
identifier	A unique integer representing a bucketed object	Section 3.5
bktid	A unique integer for each bucket	Section 3.5
bktorder	The order to traverse buckets in (increasing or decreasing)	Section 3.5
bktdest	Type representing which bucket an identifier is moving to	Section 3.5
edge	A tuple representing an edge in the graph	—
nghlist	Data type representing the neighbors of a vertex	Section 3.6
graph	Data type representing a collection of vertices and edges	Section 3.7

Table 3.1: Type names used in the interface, and their definitions. The third column provides a reference to where the type is defined in the text (if applicable).

3.3 Graph Representations

We first cover the different types of graph representations used in the library. The basic graph format stores graphs in the compressed sparse row format (described below). To efficiently store very large graphs, we also utilize a compressed graph format which encodes sorted neighbor lists using difference encoding that we describe below. Finally, our library also supports arbitrary edge weights, and provides functionality for compressing integer edge weights. As described in Section 2.1, in this thesis, we deal with graphs where vertices are identified by unique integers between 0 to $n-1$. We use the vtxid type to refer to these integer vertex IDs.

Compressed Graphs. Graphs are stored in the ***compressed sparse row (CSR)*** format. CSR stores two arrays, I and A, where the vertices are in the range $[0, n-1]$ and incident edges of a vertex v are stored in $\{A[I[v]], \ldots, A[I[v+1]-1]\}$ (with a special case for vertex $n-1$). The *uncompressed* format is equivalent to the CSR format. The format assumes that the edges incident to a vertex are sorted by the neighboring vertex ID. The interface also supports the *compressed* graph formats from the Ligra+ framework [322]. Specifically, we provide support for graphs where neighbor lists are encoded using byte codes and a parallel generalization of byte codes, which we describe next.

In the ***byte format***, we store a vertex's neighbor list by difference encoding consecutive vertex IDs, with the first difference encoded with respect to the source vertex ID. Decoding is done by sequentially uncompressing each difference, and summing the differences into a

VertexSubset Interface		Work	Depth
size	: unit → int	$O(1)$	$O(1)$
vertexMap	: (vtxid→ unit) → unit	$O(\|U\|)$	$O(\log n)$
vertexMapVal	: (vtxid→ E) → E vset		
vertexFilter	: (vtxid→ bool) → vset		
addToSubset	: (vset ∗ vtxid sequence) → unit	$O(1)$ amortized	$O(\log n)$

Figure 3.2: The core primitives in the vertexSubset interface, including the type definition of each primitive and the cost bounds. We use vset as an abbreviation for vertexSubset in the figure. A vertexSubset is a representation of a set of vertex IDs, which are unique integer identifiers for vertices. If the input vertexSubset is augmented, the user-defined functions supplied to VERTEXMAP and VERTEXFILTER take a pair of the vertex ID and augmented value as input, and the ADDTOSUBSET primitive takes a sequence of *vertexID* and augmented value pairs.

running sum which gives the vertex ID of the next neighbor. As this process is sequential, graph algorithms using the byte format that map over the neighbors of a vertex will have poor depth bounds.

We enable parallelism using the ***parallel-byte format*** from Ligra+ [322]. This format breaks the neighbors of a high-degree vertex into blocks, where each block contains a constant number of neighbors. Each block is difference encoded with respect to the source, and the format stores the blocks in a neighbor list in sorted order. As each block can have a different size, it also stores offsets that point to the start of each block. Using the parallel-byte format, the neighbor vertex IDs of a high-degree vertex can then be decoded in parallel over the blocks. We refer the reader to Ligra+ [322] for a detailed discussion of the idea. We provide many parallel primitives for processing neighbor lists compressed in the parallel-byte format in Section 5.7.

Weighted Graphs. The graph and vertex data types in the interface are generic over the weight type of the graph. Graphs with arbitrary edge weights can be represented by simply changing a template argument to the vertex and graph data types. We treat unweighted graphs as graphs weighted by an implicit null (0-byte) weight.

Both the byte and parallel-byte schemes above provide support for weighted graphs. If the graph weight type is E, the encoder simply interleaves the weighted elements of type E with the differences generated by the byte or parallel byte code. Additionally, the interface supports compressing integer weights using variable-length coding, similar to Ligra+ [322].

Bucketing Interface		Work	Depth
makeBuckets	: int ∗ (identifier → bktid) ∗ bktorder → buckets	$O(n)^{\dagger}$	$O(\log n)^{\ddagger}$
getBucket	: (bktid ∗ bktid) → bktdest	$O(1)$	$O(1)$
nextBucket	: buckets → (bktid, identifier sequence)	presented in Theorem 4.1	$O(\log n)^{\ddagger}$
updateBuckets	: buckets ∗ (identifier, bktdest) sequence → unit		

Figure 3.3: The bucketing interface, including the type definition of each primitive and the cost bounds. The bucketing structure represents a dynamic mapping between a set of identifiers to a set of buckets. The total number of identifiers is denoted by n. † denotes that a bound holds in expectation, and ‡ denotes that a bound holds *whp*. We define the semantics of each operation in the text below.

3.4 *VertexSubset Interface*

Data Types. One of the primary data types used in our interface is the **vertexSubset** data type, which represents a subset of vertices in the graph. Conceptually, a vertexSubset can either be ***sparse*** (represented as a collection of vertex IDs) or ***dense*** (represented as a boolean array or bit-vector of length n). A T vertexSubset is a generic vertexSubset, where each vertex is augmented with a value of type T.

Primitives. We use four primitives defined on vertexSubset, which we illustrate in Figure 3.2. VERTEXMAP takes a vertexSubset and applies a user-defined function f over each vertex. This primitive makes it easy to apply user-defined logic over vertices in a subset in parallel without worrying about the state of the underlying vertexSubset (i.e., whether it is sparse or dense). We also provide a specialized version of the VERTEXMAP primitive, VERTEXMAPVAL through which the user can create an augmented vertexSubset. VERTEXFILTER takes a vertexSubset and a user-defined predicate P and keeps only vertices satisfying P in the output vertexSubset. Finally, ADDTOSUBSET takes a vertexSubset and a sequence of unique vertex identifiers not already contained in the subset, and adds these vertices to the subset. Note that this function mutates the supplied vertexSubset. This primitive is implemented in $O(1)$ amortized work by representing a sparse vertexSubset using a resizable array. The worst case depth of the primitive is $O(\lg n)$ since the primitive scans at most $O(n)$ vertex IDs in parallel.

3.5 *Bucketing Interface*

We use the bucketing interface and data structure from Julienne [115], which represents a dynamic mapping from identifiers to buckets. Each bucket is represented as a vertexSubset, and the interface allows vertices to dynamically be moved through different buckets

as priorities change. The interface enables priority-based graph algorithms, including integral-weight shortest paths, k-core decomposition, and others [115]. Algorithms using the interface iteratively extract the highest priority bucket, potentially update incident vertex priorities, and repeat until all buckets are empty.

The interface is shown in Figure 3.3. The interface uses several types that we now define. An **identifier** is a unique integer representing a bucketed object. An identifier is mapped to a **bktid**, a unique integer for each bucket. The order that buckets are traversed in is given by the **bktorder** type. **bktdest** is an opaque type representing where an identifier is moving inside of the structure. Once the structure is created, an object of type **buckets** is returned to the user.

The structure is created by calling makeBuckets and providing n, the number of identifiers, D, a function which maps identifiers to bktids and O, a bktorder. Initially, some identifiers may not be mapped to a bucket, so we add nullbkt, a special bktid which lets D indicate this. Buckets in the structure are accessed monotonically in the order specified by O. After the structure is created, the nextBucket primitive is used to access the next non-empty bucket in non-decreasing (respectively, non-increasing) order. The getBucket primitive is how users indicate that an identifier is moving buckets. It requires supplying both the current bktid and next bktid for the identifier that is moving buckets, and returns an element with the bktdest type. Lastly, the updateBuckets primitive updates the bktids for multiple identifiers by supplying the bucket structure and a sequence of identifier and bktdest pairs.

The costs for using the bucket structure can be summarized by the following theorem from [115]:

Theorem 1. *When there are n identifiers, T total buckets, K calls to updateBuckets, each of which updates a set S_i of identifiers, and L calls to nextBucket, parallel bucketing takes $O(n + T + \sum_{i=0}^{K} |S_i|)$ expected work and $O((K + L) \lg n)$ depth* whp.

Chapter 4 provides more details about the interface and its implementation.

3.6 Vertex Interface

The interface provides vertex data types for both symmetric and asymmetric vertices, used for undirected and directed graphs, respectively. The vertex data type interface (see Figure 3.4) provides functional primitives over vertex neighborhoods, such as map, reduce, scan, count (a special case of reduce over the $(0, +)$ monoid where the map function is a boolean function), as well as primitives to extract a subset of the neighborhood satisfying a predicate (filter) and an internal primitive to mutate the vertex neighborhood and delete edges that do not satisfy a given predicate (pack). Since pack mutates the underlying vertex neighborhood in the graph, which requires updating the number of edges remaining

Vertex Interface		Work	Depth
Neighborhood operators:	map : (edge → unit) → unit reduce : (edge → R) ∗ R monoid → R scan : (edge → R) ∗ R monoid → R count : (edge → bool) → int filter : (edge → bool) → edge sequence pack : (edge → bool) → unit	$O(\|N(v)\|)$	$O(\log n)$
	iterate : (edge → bool) → unit	$O(d_{it})$	$O(d_{it})$
	i-th : int → edge degree : unit → int getNeighbors : unit → nghlist	$O(1)$	$O(1)$
Vertex-Vertex operators:	intersection : (nghlist ∗ nghlist) → int union : (nghlist ∗ nghlist) → int difference : (nghlist ∗ nghlist) → int	$O(l \log(h/l+1))$	$O(\log n)$

Figure 3.4: The core vertex interface, including the type definition of each primitive and the cost bounds for our implementations on uncompressed graphs. Note that for directed graphs, each of the neighborhood operators has two versions, one for the in-neighbors and one for the out-neighbors of the vertex. The cost bounds for the primitives on compressed graphs are identical assuming the compression block size is $O(\lg n)$. The cost bounds shown here assume that the user-defined functions supplied to MAP, REDUCE, SCAN, COUNT, FILTER, PACK, and ITERATE all cost $O(1)$ work to evaluate. d_{it} is the number of times the function supplied to ITERATE returns true. nghlist is an abstract type for the neighbors of a vertex, and is used by the vertex-vertex operators. The edge type is a triple (u, v, w_{uv}) where the first two entries are the ids of the endpoints, and the last entry is the weight of the edge. l and h are the degrees of the smaller and larger degree vertices supplied to a vertex-vertex operator, respectively.

in the graph, we do not expose it to the user, and instead provide APIs to pack a graph in-place using the PACKGRAPH and (NGH/SRC)PACK primitives described later. The interface also provides a sequential iterator that takes as input a function f from edges to booleans, and applies f to each successive neighbor, terminating once f returns false. Note that for directed graphs, each of the neighborhood operators has two versions, one for the in-neighbors and one for the out-neighbors of the vertex.

Finally, the interface provides vertex-vertex operators for computing the INTERSECTION, UNION, or DIFFERENCE between the set of neighbors of two vertices. We also include natural generalizations of each vertex-vertex operator that take a user-defined function f and apply it to neighbor found in the intersection (union or difference). Note that the vertex-vertex operators take the abstract nghlist type, which makes it easy to perform more complex tasks such as intersecting the in-neighbors of one vertex and the out-neighbors of a different vertex.

The cost bounds for the interface are derived by applying known bounds for efficient se-

Graph Interface		Work	Depth
Graph operators:	numVertices : unit → int numEdges : unit → int getVertex : int → vertex	$O(1)$	$O(1)$
	filterGraph : (edge → bool) → graph packGraph : (edge → bool) → unit extractEdges : (edge → bool) → edge sequence	$O(n+m)$	$O(\log n)$
	contractGraph : int sequence → graph	$O(n+m)^{\dagger}$	$O(\log n)^{\ddagger}$
VertexSubset operators:	edgeMap : vset ∗ (edge → bool) ∗ (vtxid → bool) → vset edgeMapVal : vset ∗ (edge → O option) ∗ (vtxid → bool) → O vset	$O\left(\sum_{u \in U} d(u)\right)$	$O(\log n)$
	srcReduce : vset ∗ (edge → O) ∗ O monoid ∗ (vtxid → bool) → O vset srcCount : vset ∗ (edge → bool) ∗ (vtxid → bool) → int vset srcPack : vset ∗ (edge → bool) ∗ (vtxid → bool) → int vset	$O\left(\lvert U \rvert + \sum_{u \in U'} d(u)\right)$	$O(\log n)$
	nghReduce : vset ∗ (edge → R) ∗ R monoid ∗ (vtxid → bool) ∗ (R → O option) → O vset nghCount : vset ∗ (edge → bool) ∗ (vtxid → bool) ∗ (int → O option) → O vset	$O\left(\sum_{u \in U'} d(u)\right)^{\dagger}$	$O(\log n)^{\ddagger}$

Figure 3.5: The core graph interface, including the type definition of each primitive and the cost bounds for our implementations on uncompressed graphs. vset is an abbreviation for vertexSubset when providing a type definition. Note that for directed graphs, the interface provides two versions of each vertexSubset operator, one for the in-neighbors and one for the out-neighbors of the vertex. The edge type is a triple (u, v, w_{uv}) where the first two entries are the ids of the endpoints, and the last entry is the weight of the edge. The vertexSubset operators can take both unaugmented and augmented vertexSubsets as input, but ignore the augmented values in the input. U is the vertexSubset supplied as input to a vertexSubset operator. For the src-based primitives, $U' \subseteq U$ is the set of vertices that are matched by the condition function (see the text below). The cost bounds for the primitives on compressed graphs are identical assuming the compression block size is $O(\lg n)$. The cost bounds shown here assume that the user-defined functions supplied to the vertexSubset operators all cost $O(1)$ work to evaluate. † denotes that a bound holds in expectation, and ‡ denotes that a bound holds *whp*.

quence primitives (see Chapter 2). We provide additional details about the implementations of our compressed implementations in Section 5.7.4.

3.7 Graph Interface

The interface provides graph data types for both *symmetric* and *asymmetric* graphs. Distinguishing between these graph types is important for statically enforcing arguments to problems and routines that require a symmetric input (for example, it does not make sense to call connectivity, maximal independent set, or biconnectivity on a directed input). Aside from standard functions to query the number of vertices and edges, the core graph interface consists of two types of operators: (i) *graph operators*, which provide information about a graph and enable users to perform graph-parallel operations, and (ii) vertexSubset *operators*, which take as input a vertexSubset, apply user-defined functions on edges incident to the vertexSubset in the graph in parallel and return vertexSubsets as outputs.

3.7.1 Graph Operators

The graph operators, their types, and the cost bounds provided by our implementation are shown in the top half of Figure 3.5. The interface provides primitives for querying the number of vertices and edges in the graph (numVertices and numEdges), and for fetching the vertex object for the i'th vertex (getVertex).

filterGraph. The filterGraph primitive takes as input a graph $G(V, E)$, and a boolean function P over edges specifying edges to preserve. filterGraph removes all edges in the graph where $P(u, v, w_{uv}) = \text{false}$, and returns a new graph containing only edges where $P(u, v, w_{uv}) = \text{true}$. The filterGraph primitive is useful for our triangle counting algorithm, which requires directing the edges of an undirected graph to reduce overall work.

packGraph. The interface also provides a primitive over edges called packGraph which operates similarly to filterGraph, but works in-place and mutates the underlying graph. packGraph takes as input a graph $G(V, E)$, and a boolean function P over the edges specifying edges to preserve. packGraph mutates the input graph to remove all edges that do not satisfy the predicate. This primitive is used by the biconnectivity (Algorithm 12), strongly connected components (Algorithm 14), maximal matching (Algorithm 16), and minimum spanning forest (Algorithm 13) algorithms studied in this thesis.

extractEdges. The extractEdges primitive takes as input a graph $G(V, E)$, and a boolean function P over edges which specifies edges to extract, and returns an array containing all edges where $P(u, v, w_{uv}) = \text{true}$. This primitive is useful in algorithms studied in this thesis such as maximal matching (Algorithm 16) and minimum spanning forest (Algorithm 13) where it is used to extract subsets of edges from a CSR representation of the graph, which are then processed using an algorithm operating over edge arrays.

contractGraph. Lastly, the contractGraph primitive takes a graph and an integer cluster labeling L, i.e., a mapping from vertices to cluster ids, and returns the graph

$G' = (V', E')$ where $E' = \{(L(u), L(v) \mid (u, v) \in E\}$, with any duplicate edges or self-loops removed. V' is V with all vertices with no incident edges in E' removed. This primitive is used by the connectivity (Algorithm 10) and spanning forest (Algorithm 11) algorithms studied in this thesis. The primitive can naturally be generalized to weighted graphs by specifying how to reweight parallel edges (e.g., by averaging, or taking a minimum or maximum), although this generalization is not necessary for the algorithms studied in this thesis.

Implementations and Cost Bounds. filterGraph, packGraph, and extractEdges are implementd by invoking filter and pack on each vertex in the graph in parallel. The overall work and depth comes from the fact that every edge is processed once by each endpoint, and since all vertices are filtered (packed) in parallel. contractGraph can be implemented in $O(n + m)$ expected work and $O(\lg n)$ depth *whp* in the binary-forking model using semisorting [161, 72]. In practice, contractGraph is implemented using parallel hashing [320], and we refer the reader to [321] for the implementation details.

3.7.2 VertexSubset Operators

The second part of the graph interface consists of a set of operators over vertexSubsets. At a high level, each of these primitives take as input a vertexSubset, apply a given user-defined function over the edges neighboring the vertexSubset, and output a vertexSubset. The primitives include the edgeMap primitive from Ligra, as well as several extensions and generalizations of the edgeMap primitive.

edgeMap. The edgeMap primitive takes as input a graph $G(V, E)$, a vertexSubset U, and two boolean functions F and C. edgeMap applies F to $(u, v) \in E$ such that $u \in U$ and $C(v) = \text{true}$ (call this subset of edges E_a), and returns a vertexSubset U' where $u \in U'$ if and only if $(u, v) \in E_a$ and $F(u, v) = \text{true}$. Our interface defines the edgeMap primitive identically to Ligra. This primitive is used in many of the algorithms studied in this thesis.

edgeMapData. The edgeMapData primitive works similarly to edgeMap, but returns an augmented vertexSubset. Like edgeMap, it takes as input a graph $G(V, E)$, a vertexSubset U, a function F returning a value of type R option, and a boolean function C. edgeMapData applies F to $(u, v) \in E$ such that $u \in U$ and $C(v) = \text{true}$ (call this subset of edges E_a), and returns a R vertexSubset U' where $(u, r) \in U'$ (r is the augmented value associated with u) if and only if $(u, v) \in E_a$ and $F(u, v) = \text{Some}(r)$.

The primitive is only used in the weighted breadth-first search algorithm in this thesis, where the augmented value is used to store the distance to a vertex at the start of a computation round (Algorithm 5).

srcReduce and srcCount. The ***srcReduce*** primitive takes as input a graph $G(V, E)$ and a vertexSubset U, a map function M over edges returning values of type R, a boolean function C, and a monoid A over values of type R, and returns a R vertexSubset. srcReduce

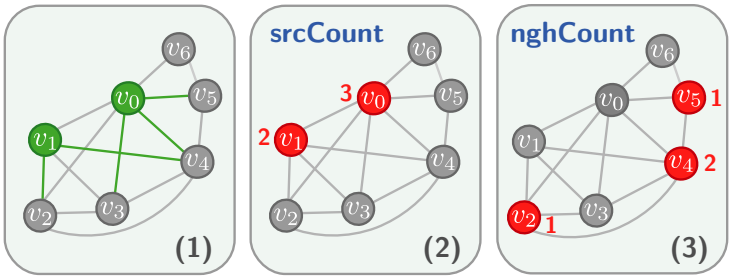


Figure 3.6: Illustration of srcCount and nghCount primitives. The input is illustrated in Panel (1), and consists of a graph and a vertexSubset, with vertices in the vertexSubset illustrated in green. The green edges are edges for which the user-defined predicate, P, returns true. Panel (2) and Panel (3) show the results of applying srcCount and nghCount, respectively. In Panel (2), the cond function C returns true for both vertices in the input vertexSubset. In Panel (3), the condition function C only returns true for v_2, v_4, and v_5, and false for v_0, v_1, v_3, and v_6. The output is an augmented int vertexSubset, illustrated in red, where each source (neighbor) vertex v s.t. $C(v) =$ true has an augmented value containing the number of incident edges where P returns true.

applies M to each $(u, v) \in E$ s.t. $u \in U$ and $C(u) =$ true (let M_u be the set of values of type R from applying M to edges incident to u), and returns a R vertexSubset U' containing (u, r) where r is the result of reducing all values in M_u using the monoid A.

The ***srcCount*** primitive is a specialization of srcReduce, where R = int, the monoid A is $(0, +)$, and the map function is specialized to a boolean (predicate) function P over edges. This primitive is useful for building a vertexSubset where the augmented value for each vertex is the number of incident edges satisfying some condition. srcCount is used in our parallel approximate set cover algorithm (Algorithm 18).

srcPack. The srcPack primitive is defined similarly to srcCount, but also removes edges that do not satsify the given predicate. Specifically, it takes as input a graph $G(V, E)$, a vertexSubset U, and two boolean functions, P, and C. For each $u \in U$ where $C(u) =$ true, the function applies P to all $(u, v) \in E$ and removes edges that do not satisfy P. The function returns an augmented vertexSubset containing all sources (neighbors), v, where $C(v) =$ true. Each of these vertices is augmented with an integer value storing the new degree of the vertex after applying the pack.

nghReduce and nghCount. The nghReduce primitive is defined similarly to srcReduce above, but aggregates the results for neighbors of the input vertexSubset. It takes as input a graph $G(V, E)$, a vertexSubset U, a map function M over edges returning values of type R, a boolean function C, a monoid A over values of type R, and lastly an update function T from values of type R to O option. It returns a O vertexSubset. This function performs the following logic: M is applies to each edge (u, v) where $u \in U$ and $C(v) =$ true in parallel (let the resulting values of type R be M_v). Next, the mapped values for each

such v are reduced in parallel using the monoid A to obtain a single value, R_v. Finally, T is called on the pair (v, R_v) and the vertex and augmented value pair (v, o) is emitted to the output vertexSubset if and only if T returns Some(o). nghReduce is used in our PageRank algorithm (Algorithm 21).

The ***nghCount*** primitive is a specialization of nghReduce, where R = int, the monoid A is $(0, +)$, and the map function is specialized to a boolean (predicate) function P over edges. nghCount is used in our k-core (Algorithm 1) and approximate densest subgraph (Algorithm 19) algorithms.

Implementations and Cost Bounds. Our implementation of edgeMap in this thesis is based on the edgeMapBlocked primitive introduced in Section 5.7.2. The same implementation is used to implement edgeMapData.

The src- primitives (srcReduce, srcCount, and srcPack) are relatively easy to implement. These implementations work by iterating over the vertices in the input vertexSubset in parallel, applying the condition function C, and then applying a corresponding vertex primitive on the incident edges. The work for source operators is $O(|U| + \sum_{u \in U'} d(u))$, where $U' \subseteq U$ consists of all vertices $u \in U$ where $C(u) =$ true, and the depth is $O(\lg n)$ assuming that the boolean functions and monoid cost $O(1)$ work to apply.

The ngh- primitives require are somewhat trickier to implement compared to the src- primitives, since these primitives require performing non-local reductions at the neighboring endpoints of edges. Both nghReduce and nghCount can be implemented by first writing out all neighbors of the input vertexSubset satisfying C to an array, A (along with their augmented values). A has size at most $O(\sum_{u \in U} d(u))$. The next step applies a work-efficient semisort to store all pairs of neighbor and value keyed by the same neighbor contiguously. The final step is to apply a prefix sum over the array, combining values keyed by the same neighbor using the reduction operation defined by the monoid, and to use a prefix sum and map to build the output vertexSubset, augmented with the final value in the array for each neighbor. The overall work is proportional to semisorting and applying prefix-sums on arrays of $|A|$ which is $O(\sum_{u \in U} d(u))$ in expectation, and the depth is $O(\lg n)$ *whp* [161, 72]. In practice, our implementations use the work-efficient histogram technique described in Section 5.7.1 for both nghReduce and nghCount.

Optimizations. We observe that for ngh- operators there is a potential to achieve speedups by applying the direction-optimization technique proposed by Beamer for the BFS problem [44] and applied to other problems by Shun and Blelloch [319]. Recall that this technique maps over all vertices $v \in V$, and for those where $C(v) =$ true, and scans over the in-edges (v, u, w_{vu}) applying F to edges where u is in the input vertexSubset, until $C(v)$ is no longer true. We can apply the same technique for nghReduce and nghCount by performing a reduction over the in-neighbors of all vertices satisfying $C(v)$. This optimization can be applied whenever the number edges incident to the input vertexSubset is a constant fraction of m. The advantage is that the direction-optimized version runs

in $O(n)$ space and performs inexpensive reads over the in-neighbors, whereas the more costly semisort or histogram based approach runs in $O(\sum_{u \in U} d(u))$ space and requires performing multiple writes per incident edge.

4 Work-Efficient Bucketing

4.1 Introduction

Both the size and availability of real-world graphs has increased dramatically over the past decade. Due to the need to process this data quickly, many frameworks for processing massive graphs have been developed for both distributed-memory and shared-memory parallel machines such as Pregel [229], GraphLab [223, 222], PowerGraph [154], and Ligra [319]. Implementing algorithms using frameworks instead of as one-off programs enables users to easily take advantage of optimizations already implemented by the framework, such as direction-optimization, compression and parallelization over both the vertices and edges of a set of vertices [44, 322].

The performance of algorithms in these frameworks is often determined by the total amount of work performed. Unfortunately, the simplest algorithms to implement in existing frameworks are often work-inefficient, i.e., they perform asymptotically more work than the most efficient sequential algorithm. While work-inefficient algorithms can exhibit excellent self-relative speedup, their absolute performance can be an order of magnitude worse than the running time of the baseline sequential algorithm, even on a very large number of cores [238].

Many commonly implemented graph algorithms in existing frameworks are *frontier-based* algorithms. Frontier-based algorithms proceed in rounds, where each round performs some computation on vertices in the current frontier, and frontiers can change from round to round. For example, in breadth-first search (BFS), the frontier on round i is the set of vertices at distance i from the source of the search. In label propagation implementations of graph connectivity [154, 319], the frontier on each round consists of vertices whose labels changed in the previous round.

Bucketing-Based Graph Algorithms. However, several fundamental graph algorithms cannot be expressed as frontier-based algorithms. These algorithms, which we call *bucketing-based* algorithms, maintain vertices in a set of ordered buckets. In each round, the algorithm extracts the vertices contained in the lowest (or highest) bucket and performs some computation on these vertices. It can then update the buckets containing either the extracted vertices or their neighbors. Frontier-based algorithms are a special case of bucketing-based algorithms, specifically they are bucketing-based algorithms that only use one bucket.

As an example, consider the weighted breadth-first search (wBFS) algorithm, which solves the single-source shortest path problem (SSSP) with nonnegative, integral edge weights in parallel [125]. Like BFS, wBFS processes vertices level by level, where level i

Algorithm	Work	Depth	Parameters
k-core	$O(\|E\| + \|V\|)$	$O(\rho \lg \|V\|)$	ρ: peeling complexity, see Chapter 2.
wBFS	$O(r_{src} + \|E\|)$	$O(r_{src} \lg \|V\|)$	r_{src}: eccentricity from the source vertex *src*, see Section 2.1.
Δ-stepping	$O(w_\Delta)$	$O(d_\Delta \lg \|V\|)$	w_Δ, d_Δ: work and number of rounds of the original Δ-stepping algorithm.
Approximate Set Cover	$O(M)$	$O(\lg^3 M)$	M: sum of the sizes of the sets.

Table 4.1: Cost bounds for the ordered graph algorithms studied in this thesis. The work bounds are in expectation and the depth bounds are with high probability.

contains all vertices at distance exactly i from *src*, the source vertex. The i'th round relaxes the neighbors of vertices in level i and updates any distances that change. Unlike a BFS, where the unvisited neighbors of the current level are in the next level, the neighbors of a level in wBFS can be spread across multiple levels. Because of this, wBFS maintains the levels in an ordered set of buckets. On round i, if a vertex v can decrease the distance to a neighbor u it places u in bucket $i + d(v, u)$. Finding the vertices in a given level can then easily be done using the bucket structure. We can show that the work of this algorithm is $O(r_{src} + |E|)$ and the depth is $O(r_{src} \lg |V|)$ where r_{src} is the eccentricity from *src* (defined in Section 2.1). However, without bucketing, the algorithm has to scan all vertices in each round to compute the current level, which makes it perform $O(r_{src}|V| + |E|)$ work and the same depth, which is not work-efficient.

Our Results. In this chapter, we study four bucketing-based graph algorithms—k-core (coreness), Δ-stepping, weighted breadth-first search (wBFS), and approximate set-cover. To provide simple and theoretically-efficient implementations of these algorithms, we design and implement a work-efficient interface for bucketing in the Ligra shared-memory graph processing framework [319]. Our extended framework, which we call ***Julienne***, enables us to write short (under 100 lines of code) implementations of the algorithms that are efficient and achieve good parallel speedup (up to 43x on 72 cores with two-way hyper-threading). Furthermore we are able to process the largest publicly-available real-world graph containing over 225 billion edges in the memory of a single multicore machine [241]. This graph must be compressed in order to be processed even on a machine with 1TB of main memory. Because Julienne supports the compression features of Ligra+, we were able to run our codes on this graph without extra modifications [322]. All of our implementations either outperform or are competitive with hand-optimized codes for the same problem. We summarize the cost bounds for the algorithms developed in this chapter in Table 4.1.

Using our framework, we obtain the first work-efficient algorithm for k-core with

nontrivial parallelism. The sequential requires performs $O(n + m)$ work [41], however the best prior parallel algorithms [319, 275, 133, 252, 110] require at least $O(k_{max}n + m)$ work where k_{max} is the largest core number in the graph—this is because these algorithms scan all remaining vertices when computing vertices in a particular core. By using bucketing, our algorithm only scans the edges of vertices with minimum degree, which makes it work-efficient. On a graph with 225B edges using 72 cores with two-way hyper-threading, our work-efficient implementation takes under 4 minutes to complete, whereas the work-inefficient implementation does not finish even after 3 hours.

Contributions. The main contributions of this chapter are as follows.

1. A simple interface for dynamically maintaining sets of identifiers in buckets.
2. A theoretically efficient parallel algorithm that implements our bucketing interface, and four applications implemented using the interface.
3. The first work-efficient implementation of k-core with non-trivial parallelism.
4. Experimental results on the largest publicly available graphs, showing that our codes achieve high performance while remaining simple. To the best of our knowledge, the results in this chapter were the first time graphs at the scale of billions of vertices and hundreds of billions of edges have been analyzed in minutes in the memory of a single shared-memory server.

4.2 Motivation

The bucket structure maintains a dynamic mapping from identifiers to bktids. The purpose of the structure is to provide efficient access to the inverse map—given a bktid, b, retrieve all identifiers currently mapped to b.

Motivation. As a motivating example, consider the weighted breadth-first search (wBFS) algorithm, which solves the single-source shortest path problem (SSSP) with nonnegative, integral edge weights in parallel [125]. Like BFS, wBFS processes vertices level by level, where level i contains all vertices at distance exactly i from *src*, the source vertex. The i'th round relaxes the neighbors of vertices in level i and updates any distances that change. Unlike a BFS, where the unvisited neighbors of the current level are in the next level, the neighbors of a level in wBFS can be spread across multiple levels. Because of this, wBFS maintains the levels in an ordered set of buckets. On round i, if a vertex v can decrease the distance to a neighbor u it places u in bucket $i + d(v, u)$. Finding the vertices in a given level can then easily be done using the bucket structure. We can show that the work of this algorithm is $O(r_{src} + |E|)$ and the depth is $O(r_{src} \lg |V|)$ where r_{src} is the eccentricity from *src*. However, without bucketing, the algorithm has to scan all vertices in each round

to compute the current level, which makes it perform $O(r_{src}|V| + |E|)$ work and the same depth, which is not work-efficient.

4.3 Bucketing Interface

The bucket structure uses several types that we now define. An **identifier** is a unique integer representing a bucketed object. An identifier is mapped to a **bktid**, a unique integer for each bucket. The order that buckets are traversed in is given by the **bktord** type. **bktdest** is an opaque type representing where an identifier is moving inside of the structure. Once the structure is created, an object of type **buckets** is returned to the user.

The structure is created by calling makeBuckets and providing n, the number of identifiers, D, a function which maps identifiers to bktids and O, a bktorder. Initially, some identifiers may not be mapped to a bucket, so we add nullbkt, a special bktid which lets D indicate this. Buckets in the structure are accessed monotonically in the order specified by O. While the interface can easily be modified to support random-access to buckets, we do not know of any algorithms that require it. Although we currently only use identifiers to represent vertices, our interface is not specific to storing and retrieving vertices, and may have applications other than graph algorithms. Even in the context of graphs, we envision algorithms where identifiers represent other objects such as edges, triangles, or graph motifs.

After the structure is created, nextBucket can be used to access the next non-empty bucket in non-decreasing (resp. non-increasing) order while updateBuckets updates the bktids for multiple identifiers. To iterate through the buckets, the structure internally maintains a variable cur which stores the value of the current bucket being processed. Note that the cur bucket can potentially be returned more than once by nextBucket if identifiers are inserted back into cur. The getBucket primitive is how users indicate that an identifier is moving buckets. We added this primitive to allow implementations to perform certain optimizations without extra involvement from the user. We describe these optimizations and present the rationale for the getBucket primitive in Section 4.5.

The full list of functions is therefore:

- **makeBuckets**(n : int, D : identifier → bktid, O : bktorder) : buckets
  Creates a bucket structure containing n identifiers in the range $[0, n)$ where the bktid for identifier i is $D(i)$. The structure iterates over the buckets in order O which is either increasing or decreasing.

- **nextBucket**() : (bktid, identifiers)
  Returns the bktid of the next non-empty bucket and the set of identifiers contained in it. When no identifiers are left in the bucket structure, the pair (nullbkt, {}) is returned.

- **getBucket**(prev : bktid, next : bktid) : bktdest
  Computes a bktdest for an identifier moving from bktid prev to next. Returns nullbkt if the identifier does not need to be updated, or if next< cur.

- **updateBuckets**(F : int → (identifier, bktdest), k : int) : unit
  Updates k identifiers in the bucket structure. The i'th identifier and its bktdest are given by $F(i)$.

4.4 *Bucketing Algorithms*

We first discuss a sequential algorithm implementing the interface and analyze its cost. The sequential algorithm shares the same underlying ideas as the parallel algorithm, so we go through it in some detail. Both algorithms in this section represent buckets exactly and so the bktdest and bktid types are identical (in particular getBucket just returns next).

Sequential Bucketing. We represent each bucket using a dynamic array, and the set of buckets using a dynamic array B (B_i is the dynamic array for bucket i). For simplicity, we describe the algorithm in the case when buckets are processed in increasing order. The structure is initialized by computing the initial number of buckets by iterating over D and allocating a dynamic array of this size. Next, we iterate over the identifiers, inserting identifier i into bucket $B_{D(i)}$ if $D(i)$ is not nullbkt, resizing if necessary. Updates are handled lazily. When updateBuckets is called, we leave the identifier in B_{prev} and just insert it into B_{next}, opening new buckets if next is outside the current range of buckets. As discussed in Section 4.3, buckets are extracted by maintaining a variable cur which is initially the first bucket. When nextBucket is called, we check to see whether B_{cur} is empty. If it is, we increment cur and repeat. Otherwise, we compact B_{cur}, only keeping identifiers $i \in B_{\text{cur}}$ where $D(i) = \text{cur}$, and return the resulting set of identifiers if it is nonempty, and repeat if it is empty.

We now discuss the total work done by the sequential algorithm. The work done by initialization is $O(n + T)$ work where T is the largest bucket used by the structure, as T is an upper bound on the number of buckets when the structure was initialized. Now, suppose the structure receives K calls to updateBuckets after being initialized, each of which updates a set S_i of identifiers where $0 \leq i < K$. By amortizing the cost of creating new buckets against T, and noticing that each update that didn't create a new bucket can be done in $O(1)$ work, the total work across all calls to updateBuckets is $O(T + \sum_{i=0}^{K} |S_i|)$.

We now argue that the total work done over all calls to nextBucket is also $O(T + \sum_{i=0}^{K} |S_i|)$. If cur is empty, we increment it and repeat, which can happen at most T times. Otherwise, there are some number of identifiers $i \in A_{\text{cur}}$. By charging each identifier, which can either be dead ($D(i) \neq \text{cur}$) or live ($D(i) == \text{cur}$), to the operation that inserted

it into the current bucket, we obtain the bound. Summing the work for each primitive gives the following lemma.

Lemma 1. *The total work performed by sequential bucketing when there are n identifiers, T total buckets and K calls to* updateBuckets *each of which updates a set S_i of identifiers is $O(n + T + \sum_{i=0}^{K} |S_i|)$.*

As discussed in Section 4.3 a given bucket can be returned multiple times by nextBucket, and the same identifiers can be reinserted into the structure multiple times using updateBuckets, so the total work of the bucket structure can potentially be much larger than $O(n)$. Some of our applications have the property that $\sum_{i=0}^{K} |S_i| = O(m)$, while also bounding T, the total number of buckets, as $O(n)$. For these applications, the cost of using the bucket-structure is $O(m + n)$.

Parallel Bucketing. In this section we describe a work-efficient parallel algorithm for our interface. The algorithm performs initialization, K calls to updateBuckets, and L calls to nextBucket in the same work as the sequential algorithm and $O((K + L) \lg n)$ depth *whp*. As before, we maintain a dynamic array B of buckets. We initialize the structure by calculating the number of initial buckets in parallel using reduce in $O(n)$ work and $O(\lg n)$ depth and allocating a dynamic array containing the initial number of buckets. Inserting identifiers into B can be done by then calling updateBuckets(D, n). nextBucket performs a filter to keep $i \in A_{\text{CUR}}$ with $D(i) ==$ cur in parallel which can be done in $O(k)$ work and $O(\lg k)$ depth on a bucket containing k identifiers.

We now describe our parallel implementation of updateBuckets, which on a set of k updates inserts the identifiers into their new buckets in $O(k)$ expected work and $O(\lg n)$ depth *whp*. The key to achieving these bounds is a work-efficient parallel *semisort* (as described in Chapter 2).

Our algorithm first creates an array of (identifier, bktid) pairs and then calls the semisort routine, using bktids as keys. The output of the semisort is an array of (identifier, bktid) pairs where all pairs with the same bktid are contiguous. Next, we map an indicator function over the semisorted pairs which outputs 1 if the index is the start of a distinct bktid and 0 otherwise. We then pack this mapped array to produce an array of indices corresponding to the start of each distinct bucket. Both steps can be done in $O(k)$ work and $O(\lg k)$ depth. Using the offsets, we calculate the number of identifiers moving to each bucket and, in parallel, resize all buckets that have identifiers moving to them. Because all identifiers moving to a particular bucket are stored contiguously in the output of the semisort, we can simply copy them to the newly resized bucket in parallel.

Semisorting the pairs requires $O(k)$ expected work and $O(\lg n)$ depth *whp*. As in the sequential algorithm, the expected work done by K calls to updateBuckets where the i'th call updates a set S_i of identifiers is $O(\sum_{i=0}^{K} |S_i|)$. Finally, because each substep of the routine requires at most $O(\lg n)$ depth, each call to updateBuckets runs in $O(\lg n)$

depth *whp*. As NEXTBUCKET also runs in $O(\lg n)$ depth, we have that a total of K calls to UPDATEBUCKETS, and L calls to NEXTBUCKET runs in $O((K+L)\lg n)$ depth *whp*. This gives the following lemma.

Lemma 2. *When there are n identifiers, T total buckets, K calls to* UPDATEBUCKETS*, each of which updates a set S_i of identifiers and L calls to* NEXTBUCKET *parallel bucketing takes $O(n + T + \sum_{i=0}^{K} |S_i|)$ expected work and $O((K+L)\lg n)$ depth* whp.

4.5 Optimizations

In practice, while many of our applications initialize the bucket structure with a large number of buckets (even $O(n)$ buckets), they only process a small fraction of them. In other applications like wBFS, the number of buckets needed by the algorithm is initially unknown. However, as the eccentricity of Web graphs and social networks tends to be small, few buckets are usually needed [356].

To make our code more efficient in situations where few buckets are being accessed, or identifiers are moved many times, we let the user specify a parameter n_B. We then only represent a range of n_B buckets (initially the first n_B buckets), and store identifiers in the remaining buckets in an 'overflow' bucket. We only move an identifier that is logically moving from its current bucket to a new bucket if its new bucket is in the current range, or if it is not yet in any bucket. This optimization is enabled by the GETBUCKET primitive, which has the user supply both the current bktid and next bktid for the identifier. Once the current range is finished, we remove identifiers in the overflow bucket and insert them back into the structure, where the n_B buckets are now used to represent the next range of n_B buckets in the algorithm.

The main benefit of this optimization is a potential reduction in the number of identifiers UPDATEBUCKETS must move as a small value of n_B can cause most of the movement to occur in the overflow bucket. We tried supporting this implementation strategy without requiring the GETBUCKET primitive by having the bucket structure maintain an extra internal mapping from identifiers to bktids. However, we found that the cost of maintaining this array of size $O(n)$ was significant (about 30% more expensive) in our applications, due to the cost of an extra random-access read and write per identifier in UPDATEBUCKETS.

Additionally, while implementing UPDATEBUCKETS using a semisort is theoretically efficient, we found that it was slow in practice due to the extra data movement that occurs when shuffling the updates. Instead, our implementation of UPDATEBUCKETS directly writes identifiers to their destination buckets and avoids the shuffle phase. We first break the array of updates into n/M blocks of length M (we set M to 2048 in our implementation). Next, we count the number of identifiers going to each bucket in each block and store these per-block histograms in an array. We then scan the array with a stride of n_B to compute

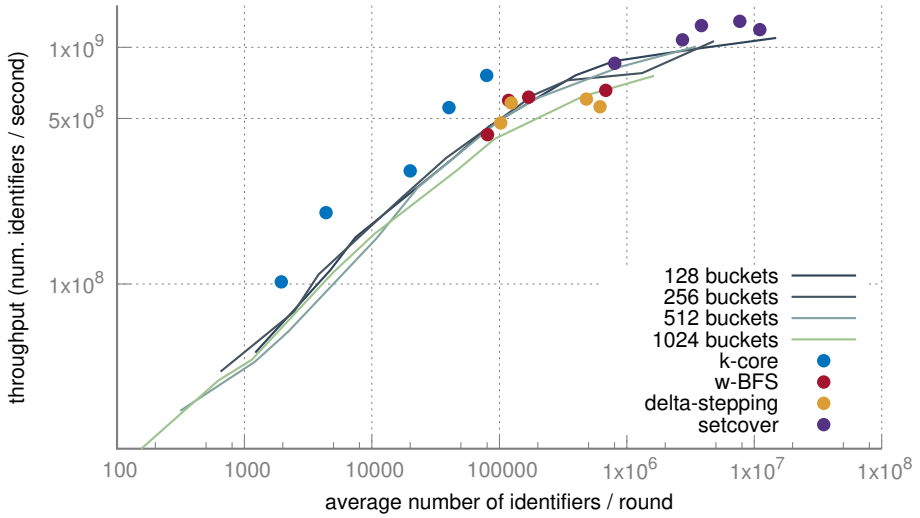


Figure 4.1: Log-log plot of throughput (billions of identifiers per second) vs. average number of identifiers processed per round.

the total number of identifiers moving to each bucket and resize the buckets. Finally, we iterate over each block again, compute a unique offset into the target bucket using the scanned value, and insert the identifier into the target bucket at this location. The total depth of this implementation of UPDATEBUCKETS is $O(M + \lg n)$ as each block is processed sequentially and the scan takes $O(\lg n)$. For small values of n_B (our default value is 128), we found that this implementation is much faster than a semisort.

4.6 *Performance*

In this section we study the performance of our parallel implementation of bucketing on a synthetic workload designed to simulate how our applications use the bucket structure. We describe the shared-memory environment used in these results in Section 2.7 of Chapter 2. The only major difference is that our experiments use the g++ compiler (version 5.4.1) with the `-O3` flag. All of our programs are run using the Cilk Plus scheduler.

Microbenchmark. The microbenchmark simulates the behavior of a bucketing-based algorithm such as k-core and Δ-stepping. On each round, these applications extract a bucket containing a set S of identifiers (vertices), and update the buckets for identifiers in $N(S)$. The microbenchmark simulates this behavior on a degree-8 random graph. Given two inputs, b, the number of initial buckets, and n, the number of identifiers, it starts by bucketing the identifiers uniformly at random and iterating over the buckets in INCREASING order. On each round, it extracts a set S of identifiers and for each extracted identifier, it picks 8 randomly chosen neighbors $\{v_0, \ldots, v_7\}$, checks whether the bucket for v_i is

greater than CUR, and if so updates its bucket to $\max(\text{CUR}, D(v_i)/2)$. If $D(v_i) \leq \text{CUR}$, it sets v_i's bucket to NULLBKT which ensures that identifiers extracted from the bucket structure are never reinserted.

We profile the performance of the bucket structure while varying b, the number of buckets. As our applications request at most about 1000 buckets, we run the microbenchmark to see how it performs when b is in the range $[128, 256, 512, 1024]$. For a given number of buckets, we vary the number of identifiers to generate different data points. The throughput of the bucket structure is calculated as the total number of identifiers extracted by NEXTBUCKET, plus the number of identifiers that move from their current bucket to a new bucket. Because identifiers moving to the NULLBKT-bucket are inexpensively handled by the bucket structure, (such requests are ignored by UPDATEBUCKETS and do not incur any random reads or writes) we exclude these requests from our total count.

We plot the throughput achieved by the structure vs. the average number of identifiers per round in Figure 5.1. The average number of identifiers per round is the total number of identifiers that are extracted and updated, divided by the number of rounds required to process all of the buckets. Using this data, we calculated the peak throughput supported by the bucket structure, and the half-performance length[1] which are approximately 1 billion identifiers per second, and an average of 500,000 identifiers per round, respectively.

Applications. We also plot points corresponding to the throughput and average number of identifiers per round achieved by our applications when run on our graphs in Figure 5.1. We observe that the benchmark throughput is a useful guideline for throughput achieved by our applications. We note that the average number of identifiers per round in k-core is noticeably lower than our other applications—this is because of the large number of rounds necessary to compute the coreness of each vertex using the peeling algorithm in our graphs (up to about 130,000). We discuss more details about our algorithms in Section 4.7 and their performance in Section 7.6.

4.7 Applications

In this section, we describe four bucketing-based algorithms and discuss how Julienne can be used to produce theoretically efficient implementations of them.

4.7.1 k-core and Coreness

A ***k*-core** of an undirected graph G is a maximal connected subgraph where every vertex has induced-degree at least k. k-cores are widely studied in the context of data mining and social network analysis because participation in a large k-core is indicative of the

[1]The number of identifiers when the system achieves half of its peak performance.

importance of a node in the graph. The coreness problem is to compute for each $v \in V$ the ***coreness*** of a vertex, or the maximum k-core v is in.

Related Work. The notion of a k-core was introduced independently by Seidman [305], and by Matula and Beck [232] (who used the term k-linkage) and identifies the subgraphs of G that satisfy the induced degree property as the k-cores of G. Anderson and Mayr showed that the decision problem for k-core can be solved in NC for $k \leq 2$, but is P-complete[2] for $k \geq 3$ [23]. Since being defined, k-cores and coreness values have found many applications from graph mining, network visualization, fraud detection, and studying biological networks [20, 317, 373].

Matula and Beck give the first algorithm which computes all coreness values. Their algorithm bucket-sorts vertices by their degree, and then repeatedly deletes the minimum-degree vertex. The affected neighbors are then moved to a new bucket corresponding to their induced degree. The total work of their algorithm is $O(m+n)$. Batagelj and Zaversnik (BZ) give an implementation of the Matula-Beck algorithm that runs in the same time bounds [41].

Parallel Peeling Algorithms. While the sequential algorithm requires $O(m+n)$ work, all existing parallel algorithms with non-trivial parallelism take at least $O(m+k_{max}n)$ work where k_{max} is the largest core number in the graph [319, 275, 133, 252, 110]. This is because the implementations do not bucket the vertices and must scan all remaining vertices when computing each core number. Our parallel algorithm as well as some existing parallel algorithms are based on a peeling procedure, where on each iteration of the procedure, vertices below a certain degree are removed from the graph. The peeling process on random (hyper)graphs has been studied and it has been shown that $O(\lg n)$ rounds of peeling suffices [189, 10], although for arbitrary graphs the number of rounds could be linear in the worst case. We note that computing a particular k-core from the coreness numbers requires finding the largest induced subgraph among vertices with coreness at least k, which can be done efficiently in parallel [104, 321].

Our Algorithm. The algorithm initializes the initial coreness value of each vertex to its degree (Line 3), and inserts the vertices into a bucketing data-structure based on their degree (Line 4). In each round, while all of the vertices have not yet been processed the algorithm performs the following steps. It first removes (or peels) the vertices in the minimum bucket, k (Line 7). Next, it computes the number of edges removed from each neighbor using the nghCount primitive. The apply function supplied to the primitive (Lines 10–18) takes a pair of a vertex, and the number of incident edges removed (v, *edgesRemoved*), updates the current coreness of the vertex v and emits a vertex and bucket identifier into the output vertexSubset if and only if the vertex needs to move to a new bucket (the return value of the getBucket primitive). The output is an augmented vertexSubset where each vertex is augmented with the bucket (a value of type bktdest) that it moves to. The last step is to

[2]There is no polylogarithmic depth algorithm for this problem unless P = NC.

Algorithm 1 k-core (Coreness)

```
1:  Coreness[0, ..., n) := 0
2:  procedure CORENESS(G(V, E))
3:      VERTEXMAP(V, fn v → Coreness[v] := d(v_i))          ▷ initialized to initial degrees
4:      B := MAKEBUCKETS(|V|, Coreness, INCREASING)         ▷ buckets processed in increasing order
5:      Finished := 0
6:      while (Finished < |V|) do
7:          (k, ids) := B.NEXTBUCKET()                      ▷ current core number, and vertices peeled this step
8:          Finished := Finished + |ids|
9:          condFn := fn v → return true
10:         applyFn := fn (v, edgesRemoved) →
11:             inducedD := D[v]
12:             if (inducedD > k) then
13:                 newD := max(inducedD − edgesRemoved, k)
14:                 Coreness[v] := newD
15:                 bkt := B.GETBUCKET(inducedD, newD)
16:                 if (bkt ≠ NULLBKT) then
17:                     return SOME(bkt)
18:             return NONE
19:         Moved := NGHCOUNT(G, ids, condFn, applyFn)      ▷ Moved is an bktdest vertexSubset
20:         B.UPDATEBUCKETS(Moved)                          ▷ update the buckets of vertices in Moved
21:     return Coreness
```

update the buckets of affected neighbors (Line 20). Once all buckets have been processed (all cores have been peeled), the algorithm returns the array *Coreness*, which contains the final coreness values of each vertex at the end of the algorithm.

We now analyze the complexity of our algorithm by plugging in quantities into Lemma 2. We can bound $\sum_{i=0}^{K} |S_i| \leq 2m$, as in the worst case each removed edge will cause an independent request to the bucket structure. Furthermore, the total number of buckets, T is at most n, as vertices are initialized into a bucket corresponding to their degree. Plugging these quantities into Lemma 2 gives us $O(m + n)$ expected work, which makes our algorithm work-efficient.

To analyze the depth of our algorithm, we define ρ to be the ***peeling-complexity*** of a graph, or the number of steps needed to peel the graph completely. A step in the peeling process removes all vertices with minimum degree, decrements the degrees of all adjacent neighbors and repeats. On graphs with peeling-complexity ρ, our algorithm runs in $O(\rho \lg n)$ depth *whp*, as each peeling-step potentially requires a call to the bucket structure to update the buckets for affected neighbors. While ρ can be as large as n in the worst-case, in practice ρ is significantly smaller than n. Our algorithm is the first work-efficient algorithm for coreness with non-trivial parallelism. The bounds are summarized

in the following theorem.

Theorem 2. *Our algorithm for coreness requires $O(m+n)$ expected work and $O(\rho \lg n)$ depth with high probability, where ρ is the peeling-complexity of the graph.*

Our serial implementation of coreness is based on an implementation of the BZ algorithm written in Khaouid et al. [199]. We re-wrote their code in C++ and integrated it into the Ligra+ framework [322], which lets us run our implementation on our largest graphs.

4.7.2 Δ-stepping and wBFS

The ***single-source shortest path (SSSP)*** problem takes as input a weighted graph $G = (V, E, w(E))$ and a source vertex *src*, and computes the shortest path distance from *src* to each vertex in V, with unreachable vertices having distance ∞. On graphs with non-negative edge weights, the problem can be solved in $O(m + n \lg n)$ work by using Dijkstra's algorithm [126] with Fibonacci heaps [146]. While Dijkstra's algorithm cannot be used on graphs with negative edge-weights, the Bellman-Ford algorithm can, but at the cost of an increased worst-case work-bound of $O(mn)$ [108]. Bellman-Ford often performs very well in parallel, but is work-inefficient for graphs with only non-negative edge weights.

Both Dijkstra and Bellman-Ford work by relaxing vertices. We denote the shortest path to each vertex by *Distance*. A relaxation occurs over a directed edge (u, v) when vertex u checks whether $Distance(u) + w(u, v) < Distance(v)$, updating $Distance(v)$ to the smaller value if this is the case. In Dijkstra's algorithm, only the vertex, v, that is closest to the source is relaxed—as the graph is assumed to have non-negative edge-weights, we are guaranteed that $Distance(v)$ is correct, and so each vertex only relaxes its outgoing edges once. In the simplest form of Bellman-Ford, all vertices relax their neighbors in each step, and so each step costs $O(m)$. The number of steps needed for Bellman-Ford to converge is proportional to the largest number of hops in a shortest path from *src* to any $v \in V$, which can be as large as $O(n)$.

Weighted Breadth-First Search (wBFS). Weighed breadth-first search (wBFS) is a version of Dijkstra's algorithm that works well for small integer edge weights and low-diameter graphs [125]. wBFS keeps a bucket for each possible distance and goes through them one by one from the lowest. Each bucket acts like a frontier as in BFS, but when we process a vertex v in a frontier i instead of placing its unvisited neighbors in the next frontier $i + 1$ we place each neighbor u in the bucket $i + d(v, u)$. wBFS turns out to be a special case of Δ-stepping, and hence we return to it later.

Δ-Stepping. The Δ-stepping algorithm provides a way to trade-off between the work-efficiency of Dijkstra's algorithm and the increased parallelism of Bellman-Ford [242]. In Δ-stepping, computation is broken up into a number of steps. On step i, vertices in the annulus at distance $[i\Delta, (i + 1)\Delta)$ are relaxed until no further distances change. The

algorithm then proceeds to the next annulus, repeating until the shortest-path distances for all reachable vertices are set. Note that when $\Delta = \infty$, this algorithm is equivalent to Bellman-Ford.

While Bellman-Ford is easy to implement in parallel, previous work has identified the difficulty in producing a scalable implementation of bucketing [170], which is required in the Δ-stepping algorithm [242]. Due to the difficulty of bucketing in parallel, many implementations of SSSP in graph-processing frameworks use the Bellman-Ford algorithm [319, 154]. Implementations of Δ-stepping do exist, but the algorithms are not easily expressed in existing frameworks, so they are either provided as primitives in a graph processing framework [259, 365] or are stand-alone implementations [227, 170, 112, 45, 228]. There are other parallel algorithms for SSSP, but for some of the algorithms, there is low parallelism [270, 86], and for others no parallel implementations exist [203, 334, 312, 102, 73]. Note that there is currently no parallel algorithm for single-source shortest paths with non-negative edge weights that matches the work of the sequential algorithm and has polylogarithmic depth. Our bucketing interface allows us to give a simple implementation of Δ-stepping with work matching that of the original algorithm [242].

Our Algorithm. The pseudocode for our implementation is shown in Algorithm 2. Shortest-path distances are stored in an array *Distance*, which are initially all ∞, except for the source, *src* which has an entry of 0. We also maintain an array of flags, *Fl*, which are used by edgeMap to remove duplicates. The bucket structure is created by specifying *n*, *Distance*, and the keyword increasing (line 20). The i'th bucket represents the annulus of vertices between distance $[i\Delta, (i+1)\Delta)$ from the source. Each Δ-step processes the closest unfinished annulus and so the buckets are processed in increasing order. On line 21 we extract the next bucket. The loop condition checks if it is nullbkt, and terminates if so. Otherwise, we explore the outgoing edges of the set of vertices in the bucket using edgeMapData. In the Update function passed to edgeMapData (lines 5–13), a neighboring vertex, d, is visited over the edge (s, d, w). s checks whether it relaxes d, i.e., $Distance[s] + w < Distance[d]$. If it can, it first uses a compareAndSwap to test whether it is the unique neighbor of d that read its value before any modifications in this round (line 10) setting this distance to be the return value (line 11) if the compareAndSwap succeeds. s then uses an atomic priorityWrite operation to update the distance to d (line 12). Unsuccessful visitors return None, which signals that they did not capture the old value of d. The result of edgeMap is a int vertexSubset where the value stored for each vertex is the distance before any modifications in this round.

Next, we call vertexMapVal (line 24), which calls the Reset function (lines 14–17) on each visited neighbor, v, that had its distance updated. Reset first resets the flag for v (line 15) to enable v to be correctly visited again on a future round. It then calculates the new bucket for v (line 17) and returns this value. The output is another vertexSubset called *NewBuckets* containing the neighbors and their new buckets. Then, on line 25, we

Algorithm 2 Δ-stepping

```
1:  Distance[0, ..., n) := ∞                                   ▷ initialized to all ∞
2:  Flags[0, ..., n) := 0                                      ▷ initialized to all 0
3:  procedure GetBucketNum(i) return ⌊Distance[i]/Δ⌋
4:  procedure Cond(v) return true
5:  procedure Update(s, d, w)
6:      nDist := Distance[s] + w
7:      oDist := Distance[d]
8:      res := None
9:      if (nDist < oDist) then
10:         if (compareAndSwap(&Flags[d], 0, 1) then
11:             res := Some()(oDist)                           ▷ the distance at the start of this round
12:         priorityWrite(&Distance[d], nDist, <)
13:     return res
14: procedure Reset(v, oldDist)
15:     Flags[v] := 0
16:     newDist := Distance[d]
17:     return B.get_bucket(⌊oldDist/Δ⌋, ⌊newDist/Δ⌋)
18: procedure Δ-stepping(G, Δ, src)
19:     Distance[r] := 0
20:     B := makeBuckets(G.n, GetBucketNum, increasing)
21:     (id, ids) := B.nextBucket()
22:     while id ≠ nullbkt do
23:         Moved := edgeMapData(G, ids, Update, Cond)
24:         NewBuckets := vertexMapVal(Moved, Reset)
25:         B.updateBuckets(NewBuckets, |NewBuckets|)
26:         (id, ids) := B.nextBucket()
27:     return Distance
```

update the buckets containing each neighbor that had its distance lowered, by calling UpdateBuckets on the vertexSubset *NewBuckets*. Lastly, we extract the next bucket from the bucket structure (line 26). We repeat these steps until the bucket structure is empty. While we describe visitors from the current frontier compareAndSwap'ing values in a separate array of flags, *Fl*, our actual implementation uses the highest-bit of *Distance* to represent *Fl*, as this reduces the number of random-memory accesses and improves performance in practice.

The original description of Δ-stepping by Meyer and Sanders [242] separates edges into *light edges* and *heavy edges*, where light edges are of length at most Δ. Inside each annulus, light edges may be processed multiple times but heavy edges only need to be processed

once, which reduces the amount of redundant work. We implemented this optimization but did not find a significant improvement in performance for our input graphs. Note that this optimization can fit into our framework by creating two graphs, one containing just the light edges and the other just the heavy edges. Light edges can be processed multiple times until the bucket number changes, at which point we relax the heavy edges once for the vertices in the bucket.

We will now argue that our implementation of Δ-stepping (with the light-heavy edge optimization) does the same amount of work as the original algorithm. The original algorithm takes at most $(d_c/\Delta)l_{\max}$ rounds to finish, where d_c is the maximum distance in the graph and $l_{\max}$ is the maximum number of light edges in a path with total weight at most Δ. Our implementation takes the same number of rounds to finish because we are relaxing exactly the same vertices as the original algorithm on each round. Using our work-efficient bucketing implementation, by Lemma 2 the work per round is linear in the number of vertices and outgoing edges processed, which matches that of the original algorithm. The depth of our algorithm is $O(\lg n)$ times the number of rounds *whp*.

Extension to wBFS. When edge weights are integers, and $\Delta = 1$, Δ-stepping becomes wBFS. This is because there can only be one round within each step. In this case we have the following strong bound on work-efficiency.

Theorem 3. *Our algorithm for wBFS (equivalent to Δ-stepping with integral weights and $\Delta = 1$) when run on a graph with m edges and eccentricity r_{src} from the source src, runs in $O(r_{src} + m)$ expected work and $O(r_{src} \lg n)$ depth* whp.

Proof. The work follows directly from the fact we do no more work than the sequential algorithm, charging only $O(1)$ work per bucket insertion and removal, which is proportional to the number of edges (every edge does at most one insertion and is later removed). The depth comes from the number of rounds and the fact that each round takes $O(\lg n)$ depth *whp* for the bucketing. □

4.7.3 Approximate Set Cover

The ***set cover*** problem takes as input a universe $\mathcal{U}$ of ground elements, $\mathcal{F}$ a collection of sets of $\mathcal{U}$ s.t. $\bigcup \mathcal{F} = \mathcal{U}$ and a cost function $c : \mathcal{F} \to \mathbb{R}_+$. The problem is to find the cheapest collection of sets $\mathcal{A} \subseteq \mathcal{F}$ that covers U, where the cost of a solution $\mathcal{A}$ is $c(\mathcal{A}) = \sum_{S \in \mathcal{A}} c(S)$. This problem can be modeled as a bipartite graph where sets and elements are vertices, with an edge connecting a set to an element if and only if the set covers that element.

Related Work. Finding the cheapest collection of sets is an NP-complete problem, and a sequential greedy algorithm [190] gives a H_n-approximation, where $H_n = \sum_{k=1}^{n} 1/k$, in $O(m)$ work for unweighted sets and $O(m \lg m)$ work for weighted sets, where m is the sum

of the sizes of the sets, or equivalently the number of edges in the bipartite graph. Parallel algorithms have been designed for approximating the set cover [283, 101, 54, 337, 67, 68, 207], and Blelloch et al. [67] present a work-efficient parallel algorithm for the problem, which takes $O(m)$ work and $O(\lg^3 m)$ depth, and gives a $(1+\epsilon)H_n$-approximation to the set cover problem. Blelloch et al. [68] later present a multicore implementation of the parallel set cover algorithm. Their code, however, is special-purpose, not being part of any general framework, and is not work-efficient. In this section, we give a work-efficient implementation of their algorithm using our bucketing interface, and we compare the performance of the codes in Section 7.6.

The Blelloch et al. algorithm works by first bucketing all sets based on their cost. In the weighted case, the algorithm first ensures that the ratio between the costliest set and cheapest set is polynomially bounded, so that the total number of buckets is kept logarithmic (see Lemma 4.2 of [68]). It does this by discarding sets that are costlier than a threshold, and including sets cheaper than another threshold in the cover. The remaining sets are bucketed based on their normalized cost (the cost per element). In order to guarantee polylogarithmic depth, only $O(\lg m)$ buckets are maintained, with a set having cost C going into bucket $\lfloor \lg_{1+\epsilon} C \rfloor$. The main loop of the algorithm iterates over the buckets from the least to most costly bucket. Each step invokes a subroutine to compute a maximal nearly-independent set (MaNIS) of sets in the current bucket. MaNIS computes a subset of the sets in the current bucket that are almost non-overlapping in the sense that each set chosen by MaNIS covers many elements that are not covered by any other chosen set. For sets not chosen by MaNIS, the number of uncovered elements they cover is shrunk by a constant factor *whp*. We refer the reader to the original paper for proofs on both MaNIS and the set cover algorithm. We now describe our algorithm for unweighted set cover, and note that it can be easily modified for the weighted case as well.

Our Julienne Implementation. The pseudocode for our implementation of the Blelloch et al. algorithm is shown in Algorithm 3. We assume that the set cover instance is represented as an undirected bipartite graph with sets and elements on opposite sides. The array *El* contains the set each element is assigned to (Line 1). The array *Fl* specifies whether elements are covered (Line 2). Initially all elements are not covered. The array D contains the number of remaining elements covered by each set (Line 3). As sets are represented by vertices, each entry of D is initially just the degree of that vertex. b stores the current bucket id (Line 4), which is updated when the algorithm extracts the next bucket (Lines 24 and Line 35). The bucket structure is created by specifying $n = |S|$, BucketNum, and the keyword decreasing (Line 23), as the algorithm processes sets in decreasing order based on the number of uncovered elements they cover.

Each round of the algorithm starts by extracting the next non-empty bucket (Lines 24 and Line 35). The degrees of sets are updated lazily, so the first phase of the algorithm packs out edges to covered elements and computes the sets that still cover enough elements

Algorithm 3 Approximate Set Cover

```
1:  El[0, ..., |E|) := ∞
2:  Fl[0, ..., |E|) := uncovered
3:  D[0, ..., |S|) := {deg(s_0), ..., deg(s_{n-1})}        ▷ initialized to the initial degree of s ∈ S
4:  b                                                      ▷ current bucket number
5:  procedure BucketNum(s) return ⌊lg_{1+ε} D[s]⌋
6:  procedure ElmUncovered(e) return Fl[e] = uncovered
7:  procedure UpdateD(s, d) D[s] := d
8:  procedure AboveThreshold(s, d) return d ≥ ⌈(1 + ε)^{max(b,0)}⌉
9:  procedure WonElm(s, e) return s = El[e]
10: procedure InCover(s) return D[s] = ∞
11: procedure VisitElms(s, e) priorityWrite(&El[e], s, <)
12: procedure WonEnough(s, elmsWon)
13:     threshold := ⌈(1 + ε)^{max(b−1,0)}⌉
14:     if (elmsWon > threshold) then
15:         D[s] := ∞                                      ▷ place s in the set cover
16: procedure ResetElms(s, e)
17:     if (El[e] = s) then
18:         if (InCover(s)) then
19:             Fl[e] := covered                           ▷ e is covered by s
20:         else
21:             El[e] := ∞                                 ▷ reset e
22: procedure SetCover(G = (S ∪ E, A))
23:     B := makeBuckets(|S|, BucketNum, decreasing)
24:     (b, Sets) := B.nextBucket()
25:     while b ≠ nullbkt do
26:         SetsD := srcPack(G, Sets, ElmUncovered)
27:         vertexMap(SetsD, UpdateD)
28:         Active := vertexFilter(SetsD, AboveThreshold)
29:         edgeMap(G, Active, VisitElms, ElmUncovered)
30:         ActiveCts := srcCount(G, Active, WonElm)
31:         vertexMap(ActiveCts, WonEnough)
32:         edgeMap(G, Active, ResetElms)
33:         Rebucket :={(s, B.get_bucket(b, BucketNum(s)) |
                s ∈ Sets and not InCover(s)}
34:         B.updateBuckets(Rebucket, |Rebucket|)
35:         (b, Sets) := B.nextBucket()
36:     return {i | InCover(i) = true}
```

to be active in this round. On Line 26, the algorithm calls srcPack with the function ElmUncovered, which packs out any covered elements in the sets' adjacency lists and updates their degrees. The return value of srcPack is a int vertexSubset (*SetsD*), where the associated value with each set is its new degree. Next, the algorithm applies vertexMap over *SetsD* with the function UpdateD, which updates *D* with the new degrees (Line 27). Finally, the algorithm calls vertexFilter with the function AboveThreshold to compute the vertexSubset, *Active*, which is the subset of *SetsD* that still have sufficient degree (Line 28).

The next phase of the algorithm implements one step of MaNIS. Note that instead of implementing MaNIS as a separate subroutine, our implementation implicitly compute it by fusing the loop that computes a MaNIS with the loop that iterates over the buckets. On Line 29 active sets reserve uncovered elements using an edgeMap, breaking ties based on their IDs using priorityWrite. edgeMap checks whether a neighboring element is uncovered using ElmUncovered (Line 6), and if so calls VisitElms (Line 29), which uses a priorityWrite to atomically update the parent of e. Next, the algorithm computes a vertexSubset, *ActiveCts*, by calling srcCount with the function WonElm (Line 30). The value associated with each set in *ActiveCts* is the number of elements successfully reserved by it. We then apply vertexMap over *ActiveCts* (Line 31) with the function WonEnough (Lines 12–15), which checks whether the number of elements reserved is above a threshold (Line 14), and if so updates the set to be in the cover.

The last phase of the algorithm marks elements that are newly covered, resets elements whose sets did not make it into the cover, and finally reinserts sets that did not make it into the cover back into the bucket structure. On Line 32, the algorithm calls edgeMap with the supplied function ResetElms (Lines 16–21) which first checks that s is the set which reserved e (Line 17). If s joined the cover, then the algorithm marks e as covered (Line 19). Otherwise, it reset $El[e] = \infty$ (Line 21) so that e can be correctly visited on future rounds. Finally, the algorithm computes *Rebucket*, a vertexSubset containing the sets that did not join the cover in this round, where the value associated with each set is its bktdest (Line 33). The bucket structure is updated with the sets in *Rebucket* on Line 34. Finally, after all rounds are over, the algorithm returns the subset of sets whose ids are in the cover (Line 36).

4.8 Experiments

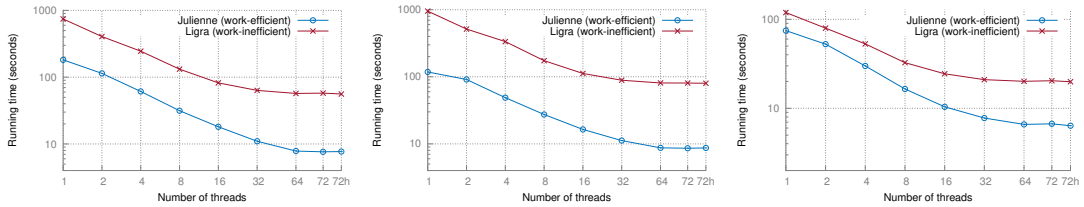
All of our experiments are run on the same machine configuration as in Section 4.6. We list the graph inputs used in our experiments in this chapter in Table 4.2. All but one of these inputs are described in Section 2.8. The remaining input, ***Hyperlink2012-Host***, is a directed hyperlink graph from the WebDataCommons dataset where nodes represent a collection of web pages belonging to the same hostname [241]. Unless mentioned

Input Graph	Num. Vertices	Num. Edges	ρ
com-Orkut	3,072,627	234,370,166	5,667
Twitter	41,652,231	1,468,365,182	–
Twitter-Sym	41,652,231	2,405,026,092	14,963
Friendster	124,836,180	3,612,134,270	10,034
Hyperlink2012-Host	101,717,775	2,043,203,933	–
Hyperlink2012-Host-Sym	101,717,775	3,880,015,728	19,063
Hyperlink2012	3,563,602,789	128,736,914,167	–
Hyperlink2012-Sym	3,563,602,789	225,840,663,232	58,710
Hyperlink2014	1,724,573,718	64,422,807,961	–
Hyperlink2014-Sym	1,724,573,718	124,141,874,032	130,728

Table 4.2: Graph inputs used in the experimental evaluation in this chapter, including both vertices, edges, and the peeling-complexity of the graph (ρ).

otherwise, the input graph is assumed to be directed, with the symmetrized version of the graph denoted with the suffix ***Sym***.

We create weighted graphs for evaluating wBFS by selecting edge weights between $[1, \lg n)$ uniformly at random. These graphs are not suitable for testing Δ-stepping, as we found that $\Delta = 1$ was always faster than a larger value of Δ. To understand the performance of our Δ-stepping implementation, we generate another family of weighted graphs with edge weights picked uniformly between $[1, 10^5)$. We successfully added edge-weights between $[1, \lg n)$ to the Hyperlink2014 graph. However, due to space limitations on our machine, we were unable to store the Hyperlink2012 graph with edge-weights between $[1, \lg n)$ and both the Hyperlink2012 and Hyperlink2014 graphs with edge-weights between $[1, 10^5)$. We use 'in parallel' to refer to running times using 144 hyper-threads.



(a) Friendster (b) Hyperlink2012-Host-Sym (c) Twitter-Sym

Figure 4.2: Running time of k-core in seconds on a 72-core machine (with hyper-threading). "72h" refers to 144 hyper-threads.

***k*-core (coreness).** Table 4.3 shows the running time of the work-efficient implementation of k-core from Julienne and the work-inefficient implementation of k-core from Ligra. Figure 4.2 shows the running time of both implementations as a function of thread count. We see that our work-efficient implementation achieves between 4-41x parallel speedup over the implementation running on a single thread. Our speedups are smaller on graphs

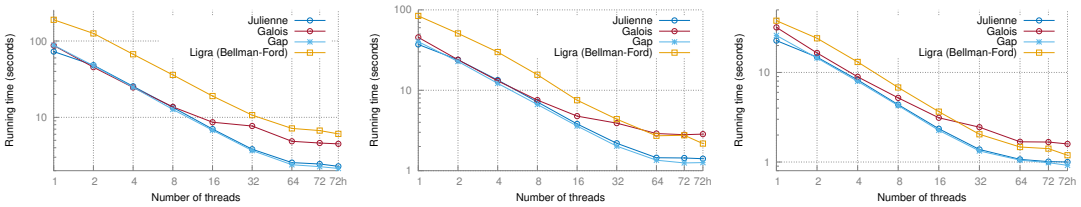
Application	com-Orkut			Twitter			Friendster			Hyperlink2012-Host			Hyperlink2012			Hyperlink2014		
	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)
***k*-core (Julienne)**	**5.43**	**1.3**	4.17	**74.6**	**6.37**	11.7	**182**	**7.7**	23.6	**118**	**8.7**	13.5	**8515**	**206**	41.3	**2820**	**97.2**	29.0
k-core (Ligra)	11.6	3.35	3.46	119	19.9	5.97	745	56	13.3	953	80.1	11.9	-	-	-	-	-	-
wBFS (Julienne)*	**2.01**	0.093	21.6	**22.8**	0.987	23.1	**73.9**	2.29	32.2	37.9	1.39	27.2	-	-	-	**392**	**9.02**	43.4
Bellman-Ford (Ligra)*	4.02	0.175	22.9	37.9	1.19	31.8	190	6.08	31.2	84.2	2.17	38.8	-	-	-	2610	35.5	73.5
wBFS (GAP)*	2.35	**0.083**	28.3	25.9	**0.919**	28.1	88.1	**2.14**	41.1	40.4	**1.26**	32.0	-	-	-	-	-	-
wBFS (Galois)*	3.46	0.319	10.8	31.9	1.59	20.06	87.6	4.49	19.5	45.5	2.85	15.9	-	-	-	-	-	-
wBFS (DIMACS)*	3.488	-	-	26.54	-	-	78.19	-	-	**35.38**	-	-	-	-	-	-	-	-
Δ-stepping (Julienne)†	**3.18**	**.167**	19.0	36.3	**2.01**	18.0	112	**3.45**	32.4	49.0	**2.09**	23.4	-	-	-	-	-	-
Bellman-Ford (Ligra)†	10.2	0.423	24.1	111	3.64	30.4	613	18.2	33.6	295	7.84	37.6	-	-	-	-	-	-
Δ-stepping (GAP)†	4.33	.294	14.7	67.6	2.39	28.2	175	4.23	41.3	57.9	2.33	24.8	-	-	-	-	-	-
Δ-stepping (Galois)†	5.1	.487	10.4	64.1	2.58	24.8	122	5.56	21.9	53.8	3.17	16.9	-	-	-	-	-	-
Δ-stepping (DIMACS)†	4.44	-	-	**35.7**	-	-	**105**	-	-	55.5	-	-	-	-	-	-	-	-
Set Cover (Julienne)	**3.66**	0.844	4.33	55.4	**3.23**	17.1	165	**6.6**	25.0	93.5	**4.83**	19.3	**3720**	**104**	35.7	**1070**	**45.1**	23.7
Set Cover (PBBS)	4.47	**0.665**	6.72	**48.4**	6.71	7.21	**137**	6.86	19.9	**71.6**	8.58	8.34	-	-	-	-	-	-

Table 4.3: Running times in seconds of Julienne algorithms over various inputs on a 72-core machine (with hyper-threading) where (1) is the single-thread time, (72h) is the 72 core time using hyper-threading and (SU) is the speedup of the application (single-thread time divided by 72-core time). Applications marked with $*$ and † use graphs with weights uniformly distributed in $[1, \lg n)$ and $[1, 10^5)$ respectively. We display the fastest sequential and parallel time for each problem in each column in bold.

where ρ is large while n and m are relatively small, such as com-Orkut and Twitter-Sym. We also ran the Batagelj and Zaversnik (BZ) algorithm described in Section 4.7.1 and found that our single-thread times are always about 1.3x faster than that of the BZ algorithm. This is because on each round we move a vertex to a new bucket just once, even if many edges are deleted from it whereas the BZ algorithm will move that vertex many times. As our algorithm on a single thread is always faster than the BZ algorithm, we report self-relative speedup, which is a lower bound on speedup over the BZ algorithm.

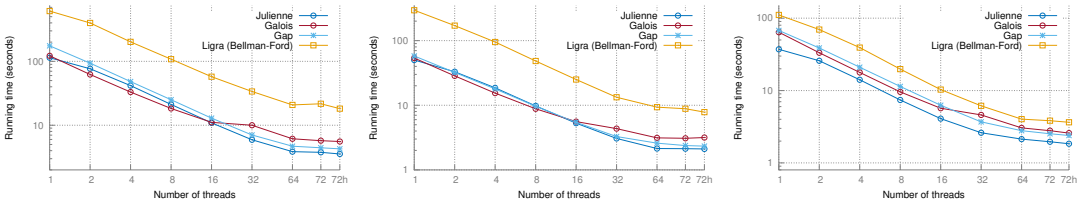
Unfortunately, we were unable to obtain the code for the ParK algorithm [110], which is to the best of our knowledge the fastest existing parallel implementation of k-core. Instead, we used a similar work-inefficient implementation of k-core available in Ligra. In parallel, our work-efficient implementation is between 2.6–9.2x faster than the work-inefficient implementation from Ligra. On Hyperlink2012-Sym and Hyperlink2014-Sym, the work-inefficient implementation did not terminate in a reasonable amount of time, and so we only report times for our implementation in Julienne. A recent paper also reported experimental results for a different parallel algorithm for k-core that is not work-efficient [299]. On a similar configuration to their machine our implementation is about 10x faster on com-Orkut, the largest graph they test on.

wBFS and Δ-stepping. Table 4.3 shows the running time of the Δ-stepping and wBFS implementations from Julienne and the GAP benchmark suite, the priority-based Bellman-Ford implementation from Galois, the Bellman-Ford implementation from Ligra and the sequential solver from the DIMACS shortest path challenge [259, 45]. Figures 4.3 and 4.4 show the running time of the four parallel implementations as a function of thread count.



(a) Friendster (b) Hyperlink2012-Host-Sym (c) Twitter-Sym

Figure 4.3: Running time of wBFS in seconds on a 72-core machine (with hyper-threading). The graphs have edge weights that are uniformly distributed in $[1, \lg n)$. "72h" refers to 144 hyper-threads.



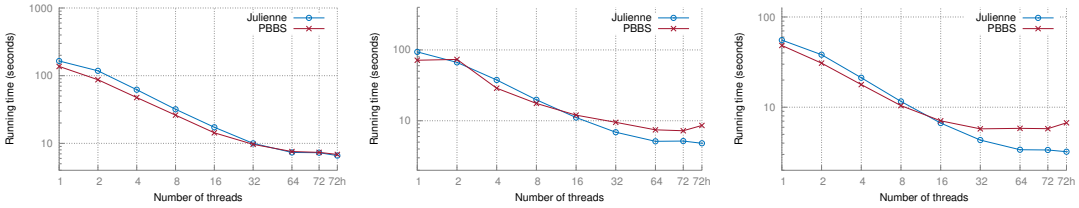
(a) Friendster (b) Hyperlink2012-Host-Sym (c) Twitter-Sym

Figure 4.4: Running time of Δ-stepping in seconds on a 72-core machine (with hyper-threading). The graphs have edge weights that are uniformly distributed in $[1, 10^5)$. "72h" refers to 144 hyper-threads.

To the best of our knowledge, we are not aware of any existing parallel implementations of wBFS, so we test wBFS against the same implementations as Δ-stepping, setting $\Delta = 1$. We see that our work-efficient implementation achieves between 22–43x parallel speedup over the implementation running on a single thread for wBFS and between 18–32.4x parallel speedup over our implementation running on a single thread for Δ-stepping. For Δ-stepping, we found that setting $\Delta = 32768$ performed best in our experiments.

Like our implementation, the SSSP implementation in GAP does not perform the light/heavy optimization described in the original Δ-stepping paper [242]. Instead of having shared buckets, it uses thread-local bins to represent buckets. The Galois algorithm is a version of Bellman-Ford that schedules nodes based on their distance from the source (closer vertices have higher priority). Because the Galois algorithm avoids synchronizing after each annulus, it achieves good speedup on graphs with large diameter, but where paths with few hops are also likely to be the shortest paths in the graph (such as road networks). On such graphs our algorithm performs poorly due to a large amount of synchronization.

All implementations achieve good speedup with an increased number of threads. On a single thread our implementation is usually faster than the single-thread times for other implementations. This is likely because of an optimization we implemented in our EDGEMAP routine, which allows traversals to only write to an amount of memory proportional to the size of the output frontier. In parallel, while the GAP implementation usually outperforms us by a small amount, we remain very competitive, being between 1.07-1.1x slower for wBFS, and between 1.1–1.7x faster for Δ-stepping. We are between 1.6–3.4x faster than the Galois implementation on wBFS and between 1.2–2.9x faster on Δ-stepping. Our implementation is between 1.2–3.9x faster for wBFS and 1.8–5.2x faster for Δ-stepping compared to the Bellman-Ford implementation in Ligra [319]. We note that there is recent work on another parallel algorithm for SSSP [228] and based on their speedups over the Δ-stepping implementation in Galois, our Julienne implementation seems competitive. We leave a detailed comparison for future work.



(a) Friendster (b) Hyperlink2012-Host-Sym (c) Twitter-Sym

Figure 4.5: Running time of set cover in seconds on a 72-core machine (with hyper-threading). "72h" refers to 144 hyper-threads.

Approximate Set Cover. We generated bipartite graphs to use as set cover instances by having vertices represent both the sets and the elements. Table 4.3 shows the running time of the work-efficient implementation of approximate set cover from Julienne and the work-inefficient implementation of approximate set cover from the PBBS benchmark suite [324]. Figure 4.5 shows the running time of both implementations as a function of thread count. We set ϵ to be 0.01 for both implementations. We see that our work-efficient implementation achieves between 4–35x parallel speedup over the implementation running on a single thread. Both implementations achieve poor speedup on com-Orkut, due to the relatively large number of rounds compared to the graph size. Our implementation achieves between 17–35x parallel speedup on our other test graphs.

The PBBS implementation is from Blelloch et al. [68] and implements the same algorithm as us [67]. Both implementations compute the same covers. We note that the PBBS implementation is not work-efficient. Instead of rebucketing the sets that are not chosen in a given step by using a bucket structure, it carries them over to the next step. In parallel, our times are between 1.2x slower to 2x faster compared to the PBBS implementation. On

graphs like Twitter-Sym, the PBBS implementation carries a large number of unchosen sets for many rounds. In these cases, our implementation achieves good speedup over the PBBS implementation because it rebuckets these sets instead of inspecting them on each round.

4.9 Discussion

In this chapter, we presented the Julienne framework for parallel graph algorithms, which allows for simple and theoretically efficient implementations of bucketing-based graph algorithms. Julienne is built by extending the Ligra and Ligra+ frameworks with an interface for parallel bucketing. We developed efficient parallel algorithms for bucketing, and showed that practical variants of these algorithms can achieve high performance on both real-world and synthetic bucketing scenarios. Using Julienne, we obtained the first work-efficient k-core algorithm with non-trivial parallelism. Finally, we showed that our implementations either outperform or are competitive with hand-optimized codes for the same applications, and can process graphs with hundreds of billions of edges in the order of minutes on a single multicore machine with one terabyte of main memory.

5 *Theoretically-Efficient Parallel Graph Algorithms*

Chang Tzu tells us of a persevering man who after three laborious years mastered the art of dragon-slaying. For the rest of his days, he had not a single opportunity to test his skills.

Jorge Luis Borges, *The Book of Imaginary Beings*

5.1 Introduction

Today, the largest publicly-available graph, the WebDataCommons hyperlink graph (Hyperlink 2012), consists of over 3.5 billion vertices and 128 billion edges [241]. This graph presents a significant computational challenge for both distributed and shared memory systems. Indeed, very few algorithms have been applied to this graph, and those that have often take hours to run [385, 225, 191], with the fastest times requiring between 1–6 minutes using a supercomputer [333, 330]. In this chapter, we show that a wide range of fundamental graph problems can be solved quickly on this graph, often in minutes, on a single commodity shared-memory machine with a terabyte of RAM.[1] For example, our k-core implementation takes under 3.5 minutes on 72 cores, whereas Slota et al. [330] report a running time of about 6 minutes for *approximate* k-core on a supercomputer with over 8000 cores. They also report that they can identify the largest connected component on this graph in 63 seconds, whereas we can identify *all* connected components in 25 seconds. Another recent result by Stergiou et al. [336] solves connectivity on the Hyperlink 2012 graph in 341 seconds on a 1000 node cluster with 12000 cores and 128TB of RAM. Compared to this result, our implementation is 13.6x faster on a system with 128x less memory and 166x fewer cores. However, we note that they are able to process a significantly larger private graph that we would not be able to fit into our memory footprint. A more complete comparison between our work and existing work, including both distributed and disk-based systems [385, 225, 191, 111, 174], is given in Section 5.8.

Importantly, all of our implementations have strong theoretical bounds on their work and depth. There are several reasons that algorithms with good theoretical guarantees are desirable. For one, they are robust as even adversarially-chosen inputs will not cause them

[1] These machines are roughly the size of a workstation and can be easily rented in the cloud (e.g., on Amazon EC2).

Problem	(1)	(72h)	(SU)	Alg.	Model	Work	Depth
Breadth-First Search (BFS)	576	8.44	68	–	BF	$O(m)$	$O(\text{diam}(G)\lg n)$
Integral-Weight SSSP (weighted BFS)	3770	58.1	64	[115]	PW-BF	$O(m)^\dagger$	$O(\text{diam}(G)\lg n)^\ddagger$
General-Weight SSSP (Bellman-Ford)	4010	59.4	67	[108]	PW-BF	$O(\text{diam}(G)m)$	$O(\text{diam}(G)\lg n)$
Single-Source Widest Path (Bellman-Ford)	3210	48.4	66	[108]	PW-BF	$O(\text{diam}(G)m)$	$O(\text{diam}(G)\lg n)$
Single-Source Betweenness Centrality (BC)	2260	37.1	60	[84]	FA-BF	$O(m)$	$O(\text{diam}(G)\lg n)$
$O(k)$-Spanner	2390	36.5	65	[248]	BF	$O(m)$	$O(k\lg n)^\ddagger$
Low-Diameter Decomposition (LDD)	980	16.6	59	[246]	BF	$O(m)$	$O(\lg^2 n)^\ddagger$
Connectivity	1640	25.0	65	[321]	BF	$O(m)^\dagger$	$O(\lg^3 n)^\ddagger$
Spanning Forest	2420	35.8	67	[321]	BF	$O(m)^\dagger$	$O(\lg^3 n)^\ddagger$
Biconnectivity	9860	165	59	[346]	FA-BF	$O(m)^\dagger$	$O(\text{diam}(G)\lg n + \lg^3 n)^\ddagger$
Strongly Connected Components (SCC)*	8130	185	43	[75]	PW-BF	$O(m\lg n)^\dagger$	$O(\text{diam}(G)\lg n)^\ddagger$
Minimum Spanning Forest (MSF)	9520	187	50	[388]	PW-BF	$O(m\lg n)$	$O(\lg^2 n)^\ddagger$
Maximal Independent Set (MIS)	2190	32.2	68	[64]	FA-BF	$O(m)^\dagger$	$O(\lg^2 n)^\ddagger$
Maximal Matching (MM)	7150	108	66	[64]	PW-BF	$O(m)^\dagger$	$O(\lg^2 m)^\ddagger$
Graph Coloring	8920	158	56	[169]	FA-BF	$O(m)$	$O(\lg n + L\lg\Delta)$
Approximate Set Cover	5320	90.4	58	[67]	PW-BF	$O(m)^\dagger$	$O(\lg^3 n)^\ddagger$
k-core	8515	184	46	[115]	FA-BF	$O(m)^\dagger$	$O(\rho\lg n)^\ddagger$
Approximate Densest Subgraph	3780	51.4	73	[38]	FA-BF	$O(m)$	$O(\lg^2 n)$
Triangle Counting (TC)	—	1168	—	[323]	BF	$O(m^{3/2})$	$O(\lg n)$
PageRank Iteration	973	13.1	74	[85]	FA-BF	$O(n+m)$	$O(\lg n)$

Table 5.1: Running times (in seconds) of our algorithms on the symmetrized Hyperlink2012 graph where (1) is the single-thread time, (72h) is the 72-core time using hyper-threading, and (SU) is the parallel speedup. Theoretical bounds for the algorithms and the variant of the binary-forking model used are shown in the last three columns. Section 2.3 provides more details about the binary-forking model. We mark times that did not finish in 5 hours with —. *SCC was run on the directed version of the graph. † denotes that a bound holds in expectation, and ‡ denotes that a bound holds with high probability (*whp*). We assume $m = \Omega(n)$.

to perform extremely poorly. Furthermore, they can be designed on pen-and-paper by exploiting properties of the problem instead of tailoring solutions to the particular dataset at hand. Theoretical guarantees also make it likely that the algorithm will continue to perform well even if the underlying data changes. Finally, careful implementations of algorithms that are nearly work-efficient can perform much less work in practice than work-inefficient algorithms. This reduction in work often translates to faster running times on the same number of cores (as shown in Chapter 4). We note that most running times that have been reported in the literature on the WebDataCommons hyperlink graph use parallel algorithms that are not theoretically-efficient.

Our Contributions in this Chapter. In this chapter, we present implementations of parallel algorithms with strong theoretical bounds on their work and depth for connectivity, biconnectivity, strongly connected components, low-diameter decomposition, graph spanners, maximal independent set, maximal matching, graph coloring, breadth-first search, single-source shortest paths, widest (bottleneck) path, betweenness centrality, PageRank, spanning forest, minimum spanning forest, k-core decomposition, approximate set cover,

approximate densest subgraph, and triangle counting. We describe the techniques used to achieve good performance on graphs with billions of vertices and hundreds of billions of edges and share experimental results for the Hyperlink 2012 and Hyperlink 2014 Web crawls, which are the largest and second largest publicly-available graphs, as well as several smaller real-world graphs at various scales. Some of the algorithms we describe are based on previous results from Ligra, Ligra+, and Julienne (see [319, 322] and Chapter 4), and other papers on efficient parallel graph algorithms [64, 169, 323]. However, most existing implementations were changed significantly in order to be more memory efficient. Several algorithm implementations for problems like strongly connected components, minimum spanning forest, and biconnectivity are new, and required implementation techniques to scale that we believe are of independent interest. We also had to extend the compressed representation from Ligra+ [322] to ensure that our graph primitives for mapping, filtering, reducing and packing the neighbors of a vertex were theoretically-efficient. We note that using compression techniques is crucial for representing the symmetrized Hyperlink 2012 graph in 1TB of RAM, as storing this graph in an uncompressed format would require over 900GB to store the edges alone, whereas the graph requires 330GB in our compressed format (less than 1.5 bytes per edge). We show the running times of our algorithms on the Hyperlink 2012 graph as well as their work and depth bounds in Table 5.1. To make it easy to build upon or compare to our work in the future, we describe the Graph Based Benchmark Suite (GBBS), a benchmark suite containing our problems with clear I/O specifications, which we have made publicly-available.[2]

In this chapter, we present an experimental evaluation of all of our implementations, and in almost all cases, the numbers we report are faster than any previous performance numbers for any machines, even much larger supercomputers. We are also able to apply our algorithms to the largest publicly-available graph, in many cases for the first time in the literature, using a reasonably modest machine. Most importantly, our implementations are based on reasonably simple algorithms with strong bounds on their work and depth. We believe that our implementations are likely to scale to larger graphs and lead to efficient algorithms for related problems.

5.2 *Shortest Path Problems*

Although work-efficient polylogarithmic-depth algorithms for single-source shortest paths (SSSP) type problems are not known due to the transitive-closure bottleneck [195], work-efficient algorithms that run in depth proportional to the diameter of the graph are known for the special cases considered in our benchmark. Several work-efficient parallel breadth-first search algorithms are known [36, 212, 70]. On weighted graphs with integral edge weights, SSSP can be solved in $O(m)$ work and $O(\mathrm{diam}(G)\lg n)$ depth [115]. Parallel

[2] https://github.com/ParAlg/gbbs

Algorithm 4 Breadth-First Search

```
1:  Visited[0, ..., n) := false
2:  Distance[0, ..., n) := ∞
3:  curDistance := 0
4:  procedure UPDATE(s, d)
5:      if TESTANDSET(&Visited[d]) then ▷ to ensure d is only added once to the next frontier
6:          Distance[d] := curDistance
7:          return true
8:      return false
9:  procedure COND(v) return !Visited[v]
10: procedure INIT(v)
11:     Distance[v] := 0
12:     Visited[v] := true
13: procedure GENERALIZEDBFS(G(V, E), F)          ▷ F is a vertexSubset of seed vertices
14:     VERTEXMAP(F, INIT)                       ▷ set distances to the seed vertices to 0
15:     while |F| > 0 do
16:         F := EDGEMAP(G, F, UPDATE, COND)     ▷ update F to contain all unvisited neighbors
17:         curDistance := curDistance + 1
18:     return Distance
19: procedure BFS(G(V, E), src)
20:     return GENERALIZEDBFS(G, vertexSubset({src}))
```

algorithms also exist for weighted graphs with positive edge weights [242, 243]. SSSP on graphs with negative integer edge weights can be solved using Bellman-Ford [108], where the number of iterations depends on the diameter of the graph. Betweenness centrality from a single source can be computed using two breadth-first searches [84, 319]. We note that very recently, a breakthrough result of Andoni et al. and Li [24, 216] show that computing $(1 + \epsilon)$-approximate SSSP can be done nearly work-efficiently (up to poly-logarithmic factors) in poly-logarithmic depth. An interesting question for future work is to understand whether ideas from this line of work can result in practical parallel approximation algorithms for SSSP.

In this paper, we present implementations of five SSSP problems that are based on graph search. We also include an algorithm to construct an $O(k)$-spanner which is based on computing low-diameter decompositions. Our implementations of BFS and Bellman-Ford are based on the implementations in Ligra [319]. Our betweenness centrality implementation applies the same broad implementation strategy as the Ligra implementation, but differs significantly in the details, which we describe below. Our wBFS implementation is based on our earlier work on Julienne [115].

Breadth-First Search (BFS)

The BFS problem is to compute a mapping representing distances between the source vertex, *src* and every other vertex. The distances to unreachable vertices should be set to ∞. Algorithm 4 shows pseudocode for our BFS implementation. The BFS procedure takes as input a graph and a source vertex *src*, and calls GeneralizedBFS with an initial vertexSubset containing just the source vertex, *src*. The GeneralizedBFS procedure is used later in our Bellman-Ford algorithm (Algorithm 6).

The GeneralizedBFS algorithm (Lines 13–18) computes the distances between vertices in an input vertexSubset, F, and all vertices reachable from vertices in F. It first initializes the *Distance* and *Visited* values for each vertex in F using a vertexMap (Line 14). Next, while the frontier is not yet empty, the algorithm repeatedly applies the edgeMap operator to generate the next frontier (Line 16). The condition function supplied to edgeMap checks whether the neighbor has been visited (Line 9). The map function (Lines 4–8) applies a testAndSet to try and visit the neighbor. If the testAndSet is successful, the map function returns true, indicating that the neighbor should be emitted in the output vertexSubset (Line 7), and otherwise returns false (Line 8). Finally, at the end of a round the algorithm increments the value of the current distance on Line 17.

Both the GeneralizedBFS and BFS algorithms run in $O(m)$ and $O(\mathsf{diam}(G) \lg n)$ depth on the binary-forking model. We note that emitting a shortest-path tree from a subset of vertices instead of distances can be done using nearly identical code, with the only differences being that (i) the algorithm will store a *Parents* array instead of a *Distances* array, and (ii) the Update function will set the parent of a vertex d to s upon a successful testAndSet. The main change we made to this algorithm compared to the Ligra implementation was to improve the cache-efficiency of the edgeMap implementation using edgeMapBlocked, the block-based version of edgeMap described in Section 5.7.

Integral-Weight SSSP (wBFS)

The integral-weight SSSP problem is to compute the shortest path distances between a source and all other vertices in a graph with positive integer edge weights. Our implementation implements the weighted breadth-first search (wBFS) algorithm, a version of Dijkstra's algorithm that is well suited for low-diameter graphs with small positive integer edge weights. Our implementation uses the bucketing interface from Julienne described in Section 4.3. The idea of our algorithm is to maintain a bucket for each possible distance, and to process them in order of increasing distance. Each bucket is like a frontier in BFS, but unlike BFS, when we process a neighbor u of a vertex v in the current bucket i, we place u in bucket $i + w_{uv}$.

Algorithm 5 shows pseudocode for our weighted BFS implementation from Julienne [115]. Initially, the distances to all vertices are ∞ (Line 1), and the distance to

Algorithm 5 wBFS

```
1:  Distance[0, ..., n) := ∞
2:  Relaxed[0, ..., n) := false
3:  procedure GetBucketNum(v) return Distance[v]
4:  procedure Cond(v) return true
5:  procedure Update(s, d, w_{s,d})
6:      newDist := Distance[s] + w_{s,d}
7:      oldDist := Distance[d]
8:      res := None
9:      if newDist < oldDist then
10:         if testAndSet(&Relaxed[d]) then                ▷ first writer this round
11:             res := Some(oldDist)                       ▷ store and return the original distance
12:         priorityWrite(&Distance[d], newDist, <)
13:     return res
14: procedure Reset(v, oldDist)
15:     Relaxed[v] := 0
16:     newDist := Distance[d]
17:     return B.getBucket(oldDist, newDist)
18: procedure wBFS(G(V, E, W), src)
19:     Distance[src] := 0
20:     B := makeBuckets(|V|, GetBucketNum, increasing)
21:     (bktId, bktContents) := B.nextBucket()
22:     while bktId ≠ nullbkt do
23:         Moved := edgeMapData(G, bktContents, Update, Cond)
24:         NewBuckets := vertexMapVal(Moved, Reset)
25:         B.updateBuckets(NewBuckets)
26:         (bktId, bktContents) := B.nextBucket()
27:     return Distance
```

the source vertex, *src*, is 0 (Line 19). Next, the algorithm buckets the vertices based on their current distance (Line 20). We note that a distance of ∞ places a vertex in a special "unknown" bucket. While the bucketing structure contains vertices, the algorithm extracts the next bucket (Lines 21 and 22) and applies the edgeMapData primitive (see Section 4.3) on all edges incident to the bucket (Line 23). The map function computes the distance along an edge $(s, d, w_{s,d})$, updating the distance to d using a priorityWrite if $D[s] + w_{s,d} < D[d]$ (Lines 5–13). The function also checks if the source vertex relaxing this edge is the first visitor to d during this round by performing a testAndSet on the *Relaxed* array, emitting d, and the old distance to d in the output vertexSubset if so.

The next step in the round applies a vertexMapVal on the augmented vertexSubset

Algorithm 6 Bellman-Ford

```
1:  Relaxed[0, ..., n) := false
2:  Distance[0, ..., n) := ∞
3:  procedure Cond(v)
4:      return true
5:  procedure ResetFlags(v)
6:      Relaxed[v] := false
7:  procedure Update(s, d, w_{s,d})
8:      newDist := Distance[s] + w_{s,d}
9:      if newDist < Distance[d] then
10:         priorityWrite(&Distance[d], newDist, <)
11:         if !Relaxed[d] then          ▷ to ensure d is only added once to the next frontier
12:             return testAndSet(&Relaxed[d])
13:     return false
14: procedure BellmanFord(G(V, E, W), src)
15:     F := vertexSubset({src})
16:     Distance[src] := 0
17:     round := 0
18:     while |F| > 0 do
19:         if round = n then          ▷ only applied if a negative weight cycle is reachable from src
20:             R := GeneralizedBFS(G, F)                          ▷ defined in Algorithm 4
21:             In parallel, set Distance[u] := −∞ for u ∈ R s.t. R[u] ≠ ∞
22:             return Distance
23:         F := edgeMap(G, F, Update, Cond)
24:         vertexMap(F, ResetFlags)
25:         round := round + 1
26:     return Distance
```

Moved. The map function first resets the *Relaxed* flag for each vertex (Line 15), and then computes the new bucket each relaxed vertex should move to using the getBucket primitive (Line 17). The output is an augmented vertexSubset *NewBuckets*, containing vertices and their destination buckets (Line 24). The last step updates the buckets for all vertices in *NewBuckets* (Line 25). The algorithm runs in $O(m)$ work in expectation and $O(\mathsf{diam}(G) \lg n)$ depth *whp* on the PW-BF-binary-forking model, as vertices use priorityWrite to write the minimum distance to a neighboring vertex in each round. The main change we made to this algorithm was to improve the cache-efficiency of edgeMapData using the block-based edgeMapBlocked algorithm described in Section 5.7.

General-Weight SSSP

The General-Weight SSSP problem is to compute a mapping with the shortest path distance between the source vertex and every reachable vertex on a graph with general (positive and negative) edge weights. The mapping should return a distance of ∞ for unreachable vertices. Furthermore, if the graph contains a negative weight cycle, the mapping should set the distance to all vertices in the cycle, and reachable from it to $-\infty$.

Our implementation for this problem is the classic Bellman-Ford algorithm [108]. Algorithm 6 shows pseudocode for our frontier-based version of Bellman-Ford. The algorithm runs over a number of rounds. The initial frontier, F, consists of just the source vertex, *src* (Line 17). In each round, the algorithm applies EDGEMAP over F to produce a new frontier of vertices that had their shortest path distance decrease, and updates F to be this new frontier. The map function supplied to EDGEMAP (Line 7–13) tests whether the distance to a neighbor can be decrased, and uses a PRIORITYWRITE to atomically lower the distance (Line 10). Emitting a neighbor to the next frontier is done using a TESTANDSET on *Relaxed*, an array of flags indicating whether the vertex had its current shortest path distance decrease (Line 12). Finally, at the end of a round the algorithm resets the flags for all vertices in F (Line 24). If the number of rounds in the algorithm reaches n, we call the GENERALIZEDBFS algorithm to compute all vertices reachable from the current frontier (Line 20) and set the distance to the vertices with distances that are not ∞ (i.e., reachable from the negative weight cycle) to $-\infty$ (Line 21).

For inputs without negative-weight cycles, the algorithm runs in $O(\mathsf{diam}(G)m)$ work and $O(\mathsf{diam}(G)\lg n)$ depth on the PW-BF-binary-forking model. If the graph contains a negative-weight cycle, the algorithm runs in $O(nm)$ work and $O(n \lg n)$ depth on the PW-BF-binary-forking model. The main change we made to this algorithm compared to the Ligra implementation was to add a GENERALIZEDBFS implementation and invoke it in the case where the algorithm detects a negative weight cycle. We also improve its cache-efficiency by using the block-based version of EDGEMAP, EDGEMAPBLOCKED, which we describe in Section 5.7.

Algorithm 7 Betweenness Centrality

```
1: Completed[0, ..., n) := false
2: NumPaths[0, ..., n) := 0 ▷ stores the number of shortest paths from r to each vertex, initially all 0
3: Dependencies[0, ..., n) := 0 ▷ stores the dependency scores of each vertex
4: Visited[0, ..., n) := false
5: procedure UPDATE(s, d)
6:     if (!Visited[d] and TESTANDSET(&Visited[d])) then return true
7:     return false
8: procedure COND(v) return !Visited[v]
9: procedure AGGREGATEPATHCONTRIBUTIONS(G, v)
10:    mapfn := fn (s, d) → return if Completed[d] then NumPaths[d] else 0
11:    NumPaths[v] := G.GETVERTEX(v).REDUCEINNGH(mapfn, (0, +))
12: procedure MARKFINISHEDFORWARDS(v)
13:    Completed[v] := true
14: procedure AGGREGATEDEPENDENCIES(G, v)
15:    mapfn := fn (s, d) → return if Completed[d] then Dependencies[d] else 0
16:    Dependencies[v] := G.GETVERTEX(v).REDUCEOUTNGH(mapfn, (0, +))
17: procedure MARKFINISHEDBACKWARDS(v)
18:    Completed[v] := true
19:    Dependencies[v] := Dependencies[v] + (1/NumPaths[v])
20: procedure UPDATEDEPENDENCIES(v)
21:    Dependencies[v] := (Dependencies[v] − (1/NumPaths[v])) · NumPaths[v]
22: procedure BC(G(V, E), r)
23:    F := vertexSubset({r})
24:    round := 0
25:    Levels[1, ..., n) := null
26:    while |F| > 0 do
27:        F := EDGEMAP(G, F, UPDATE, COND) ▷ generate the next frontier of unvisited neighbors
28:        VERTEXMAP(G, F, AGGREGATEPATHCONTRIBUTIONS) ▷ reduce in-neighbor path contributions
29:        VERTEXMAP(F, MARKFINISHEDFORWARDS)
30:        Levels[round] := F ▷ save frontier for the backwards pass
31:        round := round + 1
32:    In parallel ∀v ∈ V, set Completed[v] := false ▷ reset Completed
33:    while round > 0 do
34:        F := Levels[round − 1] ▷ use saved frontier
35:        VERTEXMAP(G, F, AGGREGATEDEPENDENCIES) ▷ reduce out-neighbor dependency contributions
36:        VERTEXMAP(F, MARKFINISHEDBACKWARDS)
37:        round := round − 1
38:    VERTEXMAP(V, UPDATEDEPENDENCIES) ▷ compute the final Dependencies scores
39:    return Dependencies
```

Single-Source Betweenness Centrality

Betweenness centrality is a classic tool in social network analysis for measuring the importance of vertices in a network [258]. Before describing the benchmark and our implementation, we introduce several definitions. Define σ_{st} to be the total number of s–t shortest paths, $\sigma_{st}(v)$ to be the number of s–t shortest paths that pass through v, and $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$ to be the ***pair-dependency*** of s and t on v.[3] The ***betweenness centrality*** of a vertex v is equal to $\sum_{s \neq v \neq t \in V} \delta_{st}(v)$, i.e. the sum of pair-dependencies of shortest-paths passing through v. Brandes [84] proposes an algorithm to compute the betweenness centrality values based on the following notion of 'dependencies': the ***dependency*** of a vertex r on a vertex v is $\delta_r(v) = \sum_{t \in V} \delta_{rt}(v)$. The single-source betweenness centrality problem in this thesis is to compute the dependency values for each vertex given a source vertex, r.

Our implementation is based on Brandes' algorithm, and follows the approach from Ligra [319], although our implementation achieves speedups over this implementation by using contention-avoiding primitives from the GBBS interface. The algorithm works in two phases, which both rely on the structure of a BFS tree rooted at r. The first phase computes σ_{rv}, i.e., the number of shortest paths from the source, r, to each vertex v. In more detail, let $P_r(v)$ be the parents of a vertex v on the previous level of the BFS tree. The first phase computes $\sigma_{rv} = \sum_{u \in P_r(v)} \sigma_{ru}$ by processing the BFS tree in level order and summing the σ_{ru} values for all parents of v in the previous level. Conceptually, the second phase applies the equation $\delta_r(v) = \sum_{w : v \in P_r(w)} \frac{\sigma_{rv}}{\sigma_{rw}} \cdot (1 + \delta_r(w))$ to compute the dependencies for each vertex by processing the levels of the BFS tree in reverse order.

Instead of directly applying the update rule for the second phase above, which requires per-neighbor random accesses to both the array storing the σ_{r*} values, and the array storing $\delta_r(*)$ values, the Ligra implementation performs an optimization which allows accessing a single array (we note that this optimization was not described in the Ligra paper, and thus we describe it here). The idea of the optimization is as follows. The second phase computes an *inverted dependency score*, $\zeta_r(v)$, for each vertex. These scores are updated level-by-level using the update rule $\zeta_r(v) = \frac{1}{\sigma_{rv}} + \sum_{w : v \in P_r(w)} \zeta_r(w)$. At the end of the second phase, a simple proof by induction shows that

$$\zeta_r(v) = \frac{1}{\sigma_{rv}} + \sum_{w \in D_r(v)} \sigma_{vw} \cdot \frac{1}{\sigma_{rw}}$$

where $D_r(v)$ is the set of all descendent vertices through v, i.e., $w \in V$ where a shortest path from r to w passes through v. These final scores can be converted to the dependency

[3] Note that $\sigma_{st}(v) = 0$ if $v \in \{s, t\}$.

scores by first subtracting $\frac{1}{\sigma_{rv}}$ and then multiplying by σ_{rv}, since

$$\sum_{w \in D_r(v)} \sigma_{rv} \cdot \frac{\sigma_{vw}}{\sigma_{rw}} = \sum_{w \in D_r(v)} \frac{\sigma_{rw}(v)}{\sigma_{rw}}$$

Next, we discuss the main difference between our implementation and that of Ligra. The Ligra implementation is based on using EDGEMAP with an map function that uses the FETCHANDADD primitive to update the number of shortest paths (σ_{rv}) in the forward phase, and to update the inverted dependencies ($\zeta_r(v)$) in the reverse phase. The Ligra implementation thus combines the generation of the next BFS frontier with aggregating the number of shortest paths passing through a vertex in the first phase, or the inverted dependency contribution of the vertex in the second phase by using the FETCHANDADD primitive. In our implementation, we observed that for certain graphs, especially those with skewed degree distribution, using a FETCHANDADD to sum up the contributions incurs a large amount of contention, and significant speedups (in our experiments, up to 2x on the Hyperlink2012 graph) can be obtained by (i) separating the computation of the next frontier from the computation of the σ_{rv} and $\delta_r(v)$ values in the two phases and (ii) computing the computation of σ_{rv} and $\delta_r(v)$ using the pull-based approach described below.

The pseudocode for our betweenness centrality implementation is shown in Algorithm 7. The algorithm runs in two phases. The first phase (Lines 26–31) computes a BFS tree rooted at the source vertex r using a NGHMAP using UPDATE, and COND defined identically to the BFS algorithm in Algorithm 4. After computing the new BFS frontier, F, the algorithm maps over the vertices in it using a VERTEXMAP (Line 28), and applies the AGGREGATEPATHCONTRIBUTIONS procedure for each vertex. This procedure (Lines 9–11) performs a reduction over all in-neighbors of the vertex to pull path-scores from vertices that are completed, i.e. $Completed[v] = \text{true}$ (Line 11). The algorithm then applies a second VERTEXMAP over F to mark these vertices as completed (Line 29). The frontier is then saved for use in the second phase (Line 30). At the end of the second phase we reset the *Status* values (Line 32).

The second phase (Lines 33–37) processes the saved frontiers level by level in reverse order. It first extracts a saved frontier (Line 34). It then applies a VERTEXMAP over the frontier applying the AGGREGATEDEPENDENCIES procedure for each vertex (Line 35. This procedure (Lines 14–16) performs a reduction over all out-neighbors of the vertex to pull the inverted dependency scores over completed neighbors. Finally, the algorithm applies a second VERTEXMAP to mark the vertices in it as completed (Line 36). After all frontiers have been processed, the algorithm finalizes the dependency scores by first subtracting the inverted *NumPaths* value, and then multiplying by the *NumPaths* value (Line 38).

Algorithm 8 $O(k)$-Spanner

```
1: procedure SPANNER(G(V, E), k)
2:     β := lg n / 2k
3:     (Clusters, Parents) := LDD(G(V, E), β)                ▷ see Algorithm 9
4:     E_LDD := {(i, Parents[i]) | i ∈ [0, n) and Parents[i] ≠ ∞}   ▷ tree edges used in the LDD
5:     I := one inter-cluster edge for each pair of adjacent clusters in L.
6:     return E_LDD ∪ I
```

Widest Path (Bottleneck Path)

The Widest Path, or Bottleneck Path benchmark in GBBS is to compute $\forall v \in V$ the maximum over all paths of the minimum weight edge on the path between a source vertex, u, and v. The algorithm is an important primitive, used for example in the Ford-Fulkerson maximum flow algorithm [144, 108], as well as other flow algorithms [39]. Sequentially, the algorithm can be solved as quickly as SSSP using a modified version of Dijkstra's algorithm. We note that faster algorithms are known sequentially for sparse graphs [128]. For positive integer-weighted graphs, the problem can also be solved using the work-efficient bucketing data structure from Julienne [115]. The buckets represent the width classes. The buckets are initialized with the out-neighbors of the source, u, and the buckets are traversed using the *decreasing* order. This order specifies that the buckets are traversed from the largest bucket to the smallest bucket. Unlike the other applications in Julienne, using widest path is interesting since the bucket containing a vertex can only increase. The problem can also be solved using the Bellman-Ford approach described above, by performing computations over the (max, min) semi-ring instead of the (min, +) semi-ring. Other than these changes, the pseudocode for the problem is identical to that of Algorithms 5 and 6.

$O(k)$-Spanner

Computing graph spanners is a fundamental problem in combinatorial graph algorithms and graph theory [276]. A graph H is a ***k-spanner*** of a graph G if $\forall u, v \in V$ connected by a path, $\text{dist}_G(u, v) \leq \text{dist}_H(u, v) \leq k \cdot \text{dist}_G(u, v)$ (equivalently, such a subgraph is called a spanner with ***stretch*** k). The spanner problem studied in this thesis is to compute an $O(k)$ spanner for a given k.

Sequentially, classic results give elegant constructions of $(2k-1)$-spanners using $O(n^{1+1/k})$ edges, which are essentially the best possible assuming the girth conjecture [351]. In this chapter, we implement the recent spanner algorithm proposed by Miller, Peng, Xu, and Vladu (MPXV) [248]. The construction results in an $O(k)$-spanner with expected size $O(n^{1+1/k})$, and runs in $O(m)$ work and $O(k \lg n)$ depth on the binary-forking model.

The MPXV spanner algorithm (Algorithm 8) uses the low-diameter decomposition (LDD) algorithm, which will be described in Section 5.3. It takes as input a parameter

k which controls the stretch of the spanner. The idea is to first compute an LDD with $\beta = \lg n/(2k)$ (Line 3). The stretch of each partition is $O(k)$ *whp*, and so the algorithm includes all spanning tree edges generated by the LDD in the spanner (Line 4). Next, the algorithm handles inter-cluster edges by taking a single inter-cluster edge between a boundary vertex and its neighbors (Line 5). In our implementation, we use a parallel hash table to select a single inter-cluster edge between two neighboring clusters.

We note that this procedure is slightly different than the procedure in the MPXV paper, which adds a single edge between *every* boundary vertex of a cluster and each adjacent cluster. Our algorithm only adds a single edge between two clusters, while the MPXV algorithm may add multiple parallel edges between two clusters. Their argument bounding the stretch to $O(k)$ for an edge spanning two clusters is still valid for our modified algorithm, since the endpoints can be first routed to the cluster centers, and then to the single edge that was selected between the two clusters.

5.3 Connectivity Problems

Low-Diameter Decomposition

A (β, d) ***decomposition*** of a graph of a graph is a partition of the vertices into clusters $C_1, \ldots, C_k$ such that (i) the shortest path distance between two vertices in C_i using only vertices within C_i is at most d, and (ii) the number of edges with endpoints belonging to different clusters is at most βm. The low-diameter decomposition problem in this thesis is to compute an $(O(\beta), O((\lg n)/\beta))$ decomposition.

Low-diameter decompositions (LDD) were first introduced in the context of distributed computing [30], and were later used in metric embedding, linear-system solvers, and parallel algorithms. Awerbuch presents a simple sequential algorithm based on ball growing that computes an $(\beta, O((\lg n)/\beta)$ decomposition [30]. Miller, Peng, and Xu (MPX) [246] present a work-efficient parallel algorithm that computes a $(\beta, O((\lg n)/\beta)$ decomposition. For each $v \in V$, the algorithm draws a start time, δ_v, from an exponential distribution with parameter β. The clustering is done by assigning each vertex u to the center v which minimizes $d(u, v) - \delta_v$. This algorithm can be implemented by running a set of parallel breadth-first searches where the initial breadth-first search starts at the vertex with the largest start time, $\delta_{\max}$, and starting breadth-first searches from other $v \in V$ once $\delta_{\max} - \delta_v$ steps have elapsed. In this chapter, we present an implementation of the MPX algorithm which computes an $(2\beta, O(\lg n/\beta))$ decomposition in $O(m)$ expected work and $O(\lg^2 n)$ depth *whp* on the binary-forking model. Our implementation is based on the non-deterministic LDD implementation from Shun et al. [321]. The main changes in our implementation are separating the LDD code from the connectivity implementation.

Algorithm 9 shows pseudocode for the modified version of the Miller-Peng-Xu algo-

Algorithm 9 Low Diameter Decomposition

```
1:  Visited[0, ..., n) := false
2:  Cluster[0, ..., n) := ∞
3:  Parents[0, ..., n) := ∞
4:  procedure Cond(v) return !Visited[v]
5:  procedure Update(s, d)
6:      if testAndSet(&Visited[d]) then
7:          Cluster[d] := Clusters[s]                    ▷ vertex d joins s's cluster
8:          Parents[d] := s                              ▷ vertex d's BFS parent in the LDD ball is s
9:          return true
10:     return false
11: procedure InitializeClusters(u)
12:     Clusters[u] := u
13:     Visited[u] := true
14: procedure Partition(V, β)
15:     P := random permutation of [0, ..., |V|)
16:     B := array of arrays of consecutive elements in P, where |B_i| = ⌊exp(i · β)⌋
17:     return B                                         ▷ B partitions [0, ..., |V|)
18: procedure LDD(G(V, E), β)
19:     B := Partition(V, β)          ▷ permute vertices, and group into O(lg n/β) batches
20:     F := vertexSubset({})                            ▷ an initially empty vertexSubset
21:     for i ∈ [0, |B|) do
22:         newClusters := vertexSubset({b ∈ B[i] | Cluster[v] = ∞})   ▷ vertices not yet clustered
    in B[i]
23:         vertexMap(newClusters, InitializeClusters)
24:         addToSubset(F, newClusters)          ▷ add new cluster centers to the current frontier
25:         F′ := edgeMap(G, F, Update, Cond)
26:         F := F                                       ▷ update the frontier for the next round
27:         round := round + 1
28:     return (Clusters, Parents)
```

rithm from [321], which computes a $(2\beta, O(\lg n/\beta))$ decomposition in $O(m)$ expected work and $O(\lg^2 n)$ depth *whp* on the binary-forking model. The algorithm allows ties to be broken arbitrarily when two searches visit a vertex in the time-step, and one can show that this only affects the number of cut edges by a constant factor [321]. The LDD algorithm starts by first permuting the vertices into $O(\lg n/\beta)$ batches, stored in an array B (Line 19). This partitioning simulates sampling from the exponential distribution by randomly permuting the vertices in parallel (Line 15) and dividing the vertices in the permutation into $O(\lg n/\beta)$ many batches (Line 16). After partitioning, the LDD algorithm then performs a sequence of rounds, where in each round all vertices that are not already

Algorithm 10 Connectivity

```
1: procedure Connectivity(G(V, E), β)
2:     (L, P) := LDD(G(V, E), β)                          ▷ see Algorithm 9
3:     G'(V', E') := contractGraph(G, L)
4:     if |E'| = 0 then
5:         return L
6:     L' := Connectivity(G'(V', E'), β)
7:     L'' := {v → L'[L[v]] | v ∈ V}                      ▷ implemented as a vertexMap over V
8:     return L''
```

clustered in the next batch are added as new cluster centers. Each cluster then tries to acquire unclustered vertices adjacent to it (thus increasing its radius by 1). This procedure is sometimes referred to as ***ball-growing*** in the literature [32, 246, 71].

The first step in the ball-growing loop extracts *newClusters*, which is a vertexSubset of vertices in the i'th batch that are not yet clustered (Line 22). Next, the algorithm applies a vertexMap to update the *Clusters* and *Visited* status of the new clusters (Line 23). The new clusters are then added to the current LDD frontier using the addToSubset primitive (Line 24). On Line 25, the algorithm uses edgeMap to traverse the out edges of the current frontier and non-deterministically acquire unvisited neighboring vertices. The condition and map functions supplied to edgeMap are defined similarly to the ones in BFS.

We note that the pseudocode show in Algorithm 9 actually returns both the LDD clustering, *Clusters*, as well as a *Parents* array. The *Parents* array contains for each vertex v that joins a different vertex's cluster ($Clusters[v] \neq v$) the parent in the BFS tree rooted at $Clusters[v]$. Specifically, for a vertex d that is not in its own cluster, $Parents[d]$ stores the vertex s that succeeds at the testAndSet in Line 6. The *Parents* array is used by both the $O(k)$-spanner and spanning forest algorithms in this chapter to extract the tree edges used in the LDD.

Connectivity

The connectivity problem is to compute a connectivity labeling of an undirected graph, i.e., a mapping from each vertex to a label such that two vertices have the same label if and only if they are in the same component in the graph. Connectivity can easily be solved sequentially in linear work using breadth-first or depth-first search. Parallel algorithms for connectivity have a long history; we refer readers to [321] for a review of the literature. Early work on parallel connectivity discovered many natural algorithms which perform $O(m \lg n)$ work [316, 31, 287, 278]. A number of optimal parallel connectivity algorithms were discovered in subsequent years [148, 104, 162, 163, 279, 277, 321], but to the best of our knowledge the recent algorithm by Shun et al. is the only linear-work polylogarithmic-depth parallel algorithm that has been studied experimentally [321].

Algorithm 11 Spanning Forest

```
1: procedure SpanningForestHelper(G(V, E), M, β)
2:     (Clusters, Parents) := LDD(G(V, E), β)                    ▷ see Algorithm 9
3:     E_LDD := {(i, Parents[i]) | i ∈ [0, n) and Parents[i] ≠ ∞}   ▷ tree edges used in the LDD
4:     E_M := {M(e) | e ∈ E_LDD}                                 ▷ original graph edges corresponding to E_LDD
5:     G'(V', E') := contractGraph(G, L)
6:     if |E'| = 0 then
7:         return E_M
8:     M' := mapping from e' ∈ E' to M(e) where e ∈ E is some edge representing e'
9:     E'' := SpanningForestHelper(G'(V', E'), M', β)
10:    return E_M ∪ E''
11: procedure SpanningForest(G(V, E), β)
12:    return SpanningForestHelper(G, {e → e | e ∈ E}, β))
```

In this chapter, we implement the connectivity algorithm from Shun et al. [321], which runs in $O(m)$ expected work and $O(\lg^3 n)$ depth *whp* on the binary-forking model. The implementation uses the work-efficient algorithm for low-diameter decomposition (LDD) [246] described above. One change we made to the implementation from [321] was to separate the LDD and contraction steps from the connectivity algorithm. Refactoring these sub-routines allowed us to express the main connectivity algorithm in about 50 lines of code.

The connectivity algorithm from Shun et al. [321] (Algorithm 10) takes as input an undirected graph G and a parameter $0 < \beta < 1$. It first runs the LDD algorithm, Algorithm 9 (Line 2), which decomposes the graph into clusters each with diameter $(\lg n)/\beta$, and βm inter-cluster edges in expectation. Next, it builds G' by contracting each cluster to a single vertex and adding inter-cluster edges while removing duplicate edges, self-loops, and isolated vertices (Line 3). It then checks if the contracted graph is empty (Line 4); if so, the current clusters are the components, and it returns the mapping from vertices to clusters (Line 5). Otherwise, it recurses on the contracted graph (Line 6) and returns the connectivity labeling produced by assigning each vertex to the label assigned to its cluster in the recursive call (Lines 7 and 8).

Spanning Forest

The spanning forest problem is to compute a subset of edges in the graph that represent a spanning forest. Finding spanning forests in parallel has been studied largely in conjunction with connectivity algorithms, since most parallel connectivity algorithms can naturally be modified to output a spanning forest (see [321] for a review of the literature).

Our spanning forest algorithm (Algorithm 11) is based on the connectivity algorithm

from Shun et al. [321] which we described earlier. Our algorithm runs in runs in $O(m)$ expected work and $O(\lg^3 n)$ depth *whp* on the binary-forking model. The main difference in the spanning forest algorithm compared to the connectivity algorithm is to include all LDD edges at each level of the recursion (Line 4). These LDD edges are extracted using the *Parents* array returned by the LDD algorithm given in Algorithm 9. Recall that this array has size proportional to the number of vertices, and all entries initialized to ∞. The LDD algorithm uses this array to store the BFS parent of each vertex v that joins a different vertex's cluster ($Clusters[v] \neq v$). The LDD edges are retrieved by checking for each index $i \in [0, n)$ whether $Parents[i] \neq \infty$ and if so taking $(i, Parents[i])$ as an LDD edge.

Furthermore, observe that the LDD edges after the topmost level of recursion are taken from a contracted graph, and need to be mapped back to some edge in the original graph realizing the contracted edge. We decide which edges in G to add by maintaining a mapping from the edges in the current graph at some level of recursion to the original edge set. Initially this mapping, M, is an identity map (Line 12). To compute the mapping to pass to the recursive call, we select any edge e in the input graph G that resulted in $e' \in E'$ and map e' to $M(e)$ (Line 8). In our implementation, we use a parallel hash table to select a single original edge per contracted edge.

Biconnectivity

A ***biconnected component*** of an undirected graph is a maximal subgraph s.t. the subgraph remains connected under the deletion of any single vertex. Two closely related definitions are articulation points and bridge. An ***articulation point*** is a vertex whose deletion increases the number of connected components, and a ***bridge*** is an edge whose deletion increases the number of connected components. Note that by definition an articulation point must have degree greater than one. The ***biconnectivity problem*** is to emit a mapping that maps each edge to the label of its biconnected component.

Sequentially, biconnectivity can be solved using the Hopcroft-Tarjan algorithm [178]. The algorithm uses depth-first search (DFS) to identify articulation points and requires $O(m + n)$ work to label all edges with their biconnectivity label. It is possible to parallelize the sequential algorithm using a parallel DFS, however, the fastest parallel DFS algorithm is not work-efficient [11]. Tarjan and Vishkin present the first work-efficient algorithm for biconnectivity [346] (as stated in the paper the algorithm is not work-efficient, but it can be made so by using a work-efficient connectivity algorithm). The same paper also introduces the Euler-tour technique, which can be used to compute subtree functions on rooted trees in parallel in $O(n)$ work and $O(\lg^2 n)$ depth on the binary-forking model. Another approach relies on the fact that biconnected graphs admit open ear decompositions to solve biconnectivity efficiently [231, 285].

In this chapter, we implement the Tarjan-Vishkin algorithm for biconnectivity in $O(m)$ expected work and $O(\max(\mathsf{diam}(G) \lg n, \lg^3 n))$ depth on the FA-BF-binary-forking model.

Algorithm 12 Biconnectivity

```
1: Parents[0, ..., n)                          ▷ the parent of each vertex in a rooted spanning forest
2: Preorder[0, ..., n)                         ▷ the preorder number of each vertex in a rooted spanning forest
3: Low[0, ..., n)                              ▷ minimum preorder number for a non-tree edge in a vertex's subtree
4: High[0, ..., n)                             ▷ maximum preorder number for a non-tree edge in a vertex's subtree
5: Size[0, ..., n)                             ▷ the size of a vertex's subtree
6: procedure IsArticulationPoint(u)
7:     p_u := Parents[u]
8:     return Preorder[p_u] ≤ Low(u) and High[u] < Preorder[p_u] + Size[p_u]
9: procedure IsNonCriticalEdge(u, v)
10:    cond_v := v = Parents[u] and IsArticulationPoint(v)
11:    cond_u := u = Parents[v] and IsArticulationPoint(u)
12:    critical := cond_u or cond_v                ▷ true if this edge is a bridge
13:    return !critical
14: procedure Biconnectivity(G(V, E))
15:    F := SpanningForest(G)
16:    Parents := root each tree in F at an arbitrary root
17:    Preorder := compute a preorder numbering on each rooted tree in F
18:    For each v ∈ V, compute Low(v), High(v), and Size(v)  ▷ subtree functions defined in the
       text
19:    packGraph(G, IsNonCriticalEdge)         ▷ removes all critical edges from the graph
20:    Labels := Connectivity(G)
21:    return (Labels, Parents)                ▷ sufficient to answer biconnectivity queries
```

Our implementation first computes connectivity labels using our connectivity algorithm, which runs in $O(m)$ expected work and $O(\lg^3 n)$ depth *whp* and picks an arbitrary source vertex from each component. Next, we compute a spanning forest rooted at these sources using breadth-first search, which runs in $O(m)$ work and $O(\mathsf{diam}(G) \lg n)$ depth. We compute *Low*, *High*, and *Size* for each vertex by running leaffix and rootfix sums on the spanning forests produced by BFS with fetchAndAdd, which requires $O(n)$ work and $O(\mathsf{diam}(G))$ depth. Finally, we compute an implicit representation of the biconnectivity labels for each edge, using an idea from [51]. This step computes per-vertex labels by removing all critical edges and computing connectivity on the remaining graph. The resulting vertex labels can be used to assign biconnectivity labels to edges by giving tree edges the connectivity label of the vertex further from the root in the tree, and assigning non-tree edges the label of either endpoint. Summing the cost of each step, the total work of this algorithm is $O(m)$ in expectation and the total depth is $O(\max(\mathsf{diam}(G) \lg n, \lg^3 n))$ *whp*.

Algorithm 12 shows the Tarjan-Vishkin biconnectivity algorithm. It first computes a spanning forest of G using a connectivity algorithm, and roots the trees in this forest

arbitrarily (Lines 15 and 16). Next, the algorithm computes a preorder numbering, *Preorder*, with respect to the roots (Line 17). It then computes for each $v \in V$ $Low(v)$ and $High(v)$, which are the minimum and maximum preorder numbers respectively of all non-tree (u, w) edges where u is a vertex in v's subtree (Line 18). It also computes $Size(v)$, the size of each vertex's subtree. Observe that one can determine whether the parent of a vertex u, p_u is an articulation point by checking $Preorder[p_u] \leq Low(u)$ and $High(u) < Preorder[p_u] + Size[p_u]$. Following [51], we refer to this set of tree edges (u, p_u), where p_u is an articulation point, as ***critical edges*** (Line 9). The last step of the algorithm is to compute a connectivity labeling of the graph with all critical edges removed. Our algorithm removes the critical edges using the packGraph primitive (see Section 4.3).

Given this final connectivity labeling, the biconnectivity label of an edge (u, v) is the connectivity label of the vertex that is further from the root of the tree. The query data structure can thus report biconnectivity labels of edges in $O(1)$ time using $2n$ space; each vertex just stores its connectivity label, and its parent in the rooted forest (for an edge (u, v) either one vertex is the parent of the other, which determines the vertex further from the root, or neither is the parent of the other, which implies that both are the same distance from the root). The same query structure can also report whether an edge is a bridge in $O(1)$ time, and refer the reader to [51] for more details. The low space usage of this query structure is important for our implementations as storing a biconnectivity label per-edge explicitly would require a prohibitive amount of memory for large graphs.

Finally, we discuss some details about our implemetation of the Tarjan-Vishkin algorithm, and give the work and depth of our implementation. Note that the *Preorder*, *Low*, *High*, and *Size* arrays can be computed either using the Euler tour technique, or by using leaffix and rootfix computations on the trees. We use the latter approach used in our implementation. The most costly step in this algorithm is to compute spanning forest and connectivity on the original graph, and so the theoretical algorithm runs in $O(m)$ work in expectation and $O(\lg^3 n)$ depth *whp* Our implementation of the Tarjan-Vishkin algorithm runs in the same work but $O(\max(\text{diam}(G) \lg n, \lg^3 n))$ depth *whp* as it computes a spanning tree using BFS and performs leaffix and rootfix computations on this tree.

Minimum Spanning Forest

The minimum spanning forest problem is to compute a subset of edges in the graph that represents a minimum spanning forest of the graph. Borůvka gave the first known sequential and parallel algorithm for computing a minimum spanning forest (MSF) [83]. Significant effort has gone into finding linear-work MSF algorithms both in the sequential and parallel settings [194, 104, 277]. Unfortunately, the linear-work parallel algorithms are highly involved and do not seem to be practical. Significant effort has also gone into designing practical parallel algorithms for MSF; we discuss relevant experimental work in

Algorithm 13 Minimum Spanning Forest

```
1:  Parents[0, ..., n) := 0
2:  procedure BORŮVKA(n, E)
3:      Forest := {}
4:      while |E| > 0 do
5:          P[0, ..., n) := (∞, ∞)                ▷ array of (weight , index) pairs for each vertex
6:          for i ∈ [0, |E|) in parallel do
7:              (u, v, w) := E[i]                                              ▷ the i'th edge in E
8:              priorityWrite(&P[u], (w, i), <)    ▷ < lexicographically compares the (weight, index) pairs
9:              priorityWrite(&P[v], (w, i), <)
10:         for u ∈ [0, n) where P[u] ≠ (∞, ∞) in parallel do
11:             (w, i) := P[u]                  ▷ the index and weight of the MSF edge incident to u
12:             v := the neighbor of u along the E[i] edge
13:             if v > u and P[v] = (w, i) then   ▷ v also chose E[i] as its MSF edge; symmetry break
14:                 Parents[u] := u                                  ▷ make u the root of a component
15:             else
16:                 Parents[u] := v                  ▷ otherwise v < u; join v's component
17:         Forest := Forest ∪ {edges that won on either endpoint in P}      ▷ add new MSF edges
18:         PointerJump(Parents)                    ▷ compress the parents array (see Chpater 2)
19:         E := map(E, fn (u, v, w) → return (Parents[u], Parents[v], w))          ▷ relabel edges
20:         E := filter(E, fn (u, v, w) → return u ≠ v)                             ▷ remove self-loops
21:     return Forest
22: procedure MinimumSpanningForest(G(V, E, w))
23:     Forest := {}
24:     Rounds := 0
25:     vertexMap(V, fn u → Parents[u] = u)     ▷ initially each vertex is in its own component
26:     while G.numEdges() > 0 do
27:         T := select min(3n/2, m)-th smallest edge weight in G
28:         if Rounds = 5 then T := largest edge weight in G
29:         E_F := extractEdges(G, fn (u, v, w_{u,v}) → return w_{u,v} ≤ T)
30:         Forest := Forest ∪ Borůvka(|V|, E_F)
31:         packGraph(G, fn (u, v, w_{u,v}) → return Parents[u] ≠ Parents[v]) ▷ remove self-loops
32:         Rounds := Rounds + 1
33:     return Forest
```

Section 5.8. Due to the simplicity of Borůvka, many parallel implementations of MSF use variants of it.

In this chapter, we present an implementation of Borůvka's algorithm that runs in

$O(m \lg n)$ work and $O(\lg^2 n)$ depth *whp* on the PW-BF-binary-forking model. Our implementation is based on a recent implementation of Borůvka by Zhou that runs on the edgelist format (graphs represented as a sequence of edges, see Chapter 2) [388]. We made several changes to the algorithm which improve performance and allow us to solve MSF on very large graphs stored in the CSR format (defined in Chapter 2). Storing an integer-weighted graph in edgelist format would require well over 1TB of memory to represent the edges in the WebDataCommons hyperlink graph alone.

Algorithm 13 shows pseudocode for our implementation of Borůvka's algorithm based on shortcutting using pointer-jumping instead of contraction. Our code uses an implementation of Borůvka (Lines 2–21) that works over an edgelist as a subroutine; to make it efficient in practice, we ensure that the size of the lists passed to it are much smaller than m. The main algorithm (Lines 22–33) maintains a *Parents* array that represents the connected components that have been found by the algorithm so far. Initially, each vertex is in its own component (Line 25). The main algorithm performs a constant number of *filtering* steps on a small number of the lowest-weight edges that are extracted from the graph. Each filtering step first solves an approximate k'th smallest problem in order to determine a weight threshold, which is either the weight of approximately the $3n/2$'th lightest edge, or the max edge weight (if the maximum number of filtering rounds are reached) (Line 27). This step can be easily implemented using the vertex primitives in Section 4.3 and binary search. Edges lighter than the threshold are then extracted using the extractEdges primitive, defined in Section 4.3 (Line 29). The algorithm then runs Borůvka on this subset of edges (Line 30), which we describe next. Borůvka returns edges that are in the minimum spanning forest, and additionally compresses the *Parents* array based on the new forst edges. Lastly, the main algorithm removes any edges that are now contained in the same component using the packGraph primitive (Line 31).

The edgelist-based Borůvka implementation (Lines 2–21) takes as input the number of vertices and a subset of lowest weight edges from the graph. The initial forest is empty (Line 3). The algorithm runs over a series of rounds. Within a round, the algorithm first initializes an array P of (weight, index) pairs for all vertices (Line 5). Next, it loops in parallel over all edges in E and perform priorityWrites to P based on the weight on both endpoints of the edge (Lines 8 and 9). This step writes the weight and index-id of a minimum-weight edge incident to a vertex v into $P[v]$. Next, for each vertex u that found an MSF edge incident to it, i.e., $P[u] \neq (\infty, \infty)$ (Line 10), the algorithm determines v, the neighbor of u along this MSF edge (Lines 11–12). If v also selected (u, v, w) as its MSF edge, the algorithm sets the vertex with lower id to be the root of a tree (Line 14), and the vertex with higher id to point to lower one (Line 16). Lastly, the algorithm performs several clean-up steps. First, it updates the forest with all newly identified MSF edges (Line 17). Next, it performs pointer-jumping (see [188]) to compress trees created in *Parents* (Line 18). Note that the pointer-jumping step can be work-efficiently implemented in $O(\lg n)$ depth *whp* in the binary-forking model [72]. Finally, it relabels the edges array E based on the

Algorithm 14 Strongly Connected Components

```
1:  procedure SCC(G(V, E))
2:      B := Partition(V, 1)   ▷ permute and group vertices in O(lg n) batches of increasing size (see Alg.9)
3:      L[0, ..., n) := ∞
4:      Done[0, ..., n) := false
5:      for i ∈ [0, |B|) do
6:          Centers :={v ∈ B[i] | !Done[i]} ▷ vertices starting in the i'th batch that are not yet done
7:          OutL := MarkReachable(G, Centers) ▷ map with pairs (u, c) indicating that c reaches u in G
8:          InL := MarkReachable(G^T, Centers) ▷ map with pairs (u, c) indicating that c reaches u in G^T
9:          for (u, c) ∈ InL ∩ OutL in parallel do
10:             Done[u] := true                                   ▷ mark this vertex as done
11:             priorityWrite(&L[u], c, <) ▷ u's label is set to the minimum id center in u's SCC
12:         packGraph(G, fn (u, v) → return                       ▷ preserve edges within the same subproblem
13:             |InL[u]| = |InL[v]| and |OutL[u]| = |OutL[v]|)
14:     return L
```

new ids in *Parents* (Line 19) and then filters E to remove any self-loops, i.e., edges within the same component after this round (Line 20).

We note that our implementation uses indirection by maintaining a set of active vertices and a using a set of integer edge-ids to represent E in the Borůvka procedure. Applying indirection over the vertices helps in practice as the algorithm can allocate P (Line 5) to have size proportional to the number of active vertices in each round, which may be much smaller than n. Applying indirection over the edges allows the algorithm to perform a filter over just the ids of the edges, instead of triples containing the two endpoints and the weight of each edge.

We point out that the filtering idea used in our main algorithm is similar to the theoretically-efficient algorithm of Cole et al. [104], except that instead of randomly sampling edges, our filtering procedure selects a linear number of the lowest weight edges. Each filtering step costs $O(m)$ work and $O(\lg m)$ depth, but as we only perform a constant number of steps, they do not affect the work and depth asymptotically. In practice, most of the edges are removed after 3–4 filtering steps, and so the remaining edges can be copied into an edgelist and solved in a single Borůvka step. We also note that as the edges are initially represented in both directions, we can pack out the edges so that each undirected edge is only inspected once (we noticed that earlier edgelist-based implementations stored undirected edges in both directions).

Strongly Connected Components

The strongly connected components problem is to compute a labeling L that maps each vertex to a unique label for its strongly connected component (i.e., $L[u] = L[v]$ iff there is a directed path from u to v and from v to u). Tarjan's algorithm is the textbook sequential algorithm for computing the strongly connected components (SCCs) of a directed graph [108]. As it uses depth-first search, we currently do not know how to efficiently parallelize it [11]. The current theoretical state-of-the-art for parallel SCC algorithms with polylogarithmic depth reduces the problem to computing the transitive closure of the graph. This requires $\tilde{O}(n^3)$ work using combinatorial algorithms [149], which is significantly higher than the $O(m + n)$ work done by sequential algorithms. As the transitive-closure based approach performs a significant amount of work even for moderately sized graphs, subsequent research on parallel SCC algorithms has focused on improving the work while potentially sacrificing depth [142, 107, 303, 75]. Conceptually, these algorithms first pick a random pivot and use a reachability-oracle to identify the SCC containing the pivot. They then remove this SCC, which partitions the remaining graph into several disjoint pieces, and recurse on the pieces.

In this chapter, we present the first implementation of the SCC algorithm from Blelloch et al. [75], shown in Algorithm 14. We refer the reader to Section 6.2 of [75] for proofs of correctness and its work and depth bounds. The algorithm is similar in spirit to randomized quicksort. The algorithm first sets the initial label for all vertices as ∞ and marks all vertices as not done (Lines 3 and 4). Next, it randomly permutes the vertices and partitions them into $\lg n$ batches whose sizes increase geometrically (Line 2). This pseudocode for Partition is given in Algorithm 9. Specifically, $B[i]$ contains all vertices that are part of the i'th batch. It then processes the batches one at a time.

For each batch, it first computes *Centers*, which are the vertices in this batch that are not yet done (Line 6). The next step calls MarkReachable from the centers on both G and the transposed graph, G^{T} (Lines 7–8). MarkReachable takes the set of centers and uses a variant of a breadth-first search to compute the sets *OutL* (*IntL*), which for a center $c \in$ *Centers* includes all (v, c) pairs for vertices v that c can reach through its out-edges (in-edges). We describe this procedure in more detail below. Finally, the algorithm computes all (u, c) pairs in the intersection of *InL* and *OutL* in parallel (Line 9). For each pair, the algorithm first marks the vertex as done (Line 10). It then performs a priorityWrite to atomically try and update the label of the vertex to c (Line 11). After the parallel loop on Line 9 finishes, the label for a vertex u that had some vertex in its SCC appear as a center in this batch will be set to the minimum vertex id over all such centers from its SCC.

The last step of the algorithm refines the subproblems in the graph by partitioning it, i.e., deleting all edges which the algorithm identifies as not being in the same SCC. In our implementation, this step is implemented using the packGraph primitive (Line 12), which considers every directed edge in the graph and only preserves edges (u, v) where

the number of centers reaching u and v in *InL* are equal (respectively the number of centers reaching them in *OutL*). We note that the algorithm described in Blelloch et al. [75] suggests that to partition the graph, each reachability search can check whether any edge (u, v) where one endpoint is reachable in the search, and the other is not, can be cut (possibly cutting some edges multiple times). The benefit of our approach is that we can perform a single parallel scan over the edges in the graph and pack out a removed edge exactly once. Our implementation runs in $O(m \lg n)$ expected work and $O(\text{diam}(G) \lg n)$ depth *whp* on the PW-BF-binary-forking model.

One of the challenges in implementing this SCC algorithm is how to compute reachability information from multiple vertices (the centers) simultaneously. Our implementation explicitly materializes the forward and backward reachability sets for the set of centers that are active in the current phase. The sets are represented as hash tables that store tuples of vertices and center IDs, (u, c_i), representing a vertex u in the same subproblem as c_i that is visited by a directed path from c_i. We explain how to make the hash table technique practical in Section 5.7.3. The reachability sets are computed by running simultaneous breadth-first searches from all active centers. In each round of the BFS, we apply EDGEMAP to traverse all out-edges (or in-edges) of the current frontier. When we visit an edge (u, v) we try to add u's center IDs to v. If u succeeds in adding any IDs, it TESTANDSET's a visited flag for v, and returns it in the next frontier if the TESTANDSET succeeded. Each BFS requires at most $O(\text{diam}(G))$ rounds as each search adds the same labels in each round as it would have had it run in isolation.

We also implement an optimized search for the first phase, which just runs two regular BFSs over the in-edges and out-edges from a single pivot and stores the reachability information in bit-vectors instead of hash-tables. It is well known that many directed real-world graphs have a single massive strongly connected component, and so with reasonable probability the first vertex in the permutation will find this giant component [87]. Our implementation also supports a *trimming optimization* that is used by some papers in the literature [237, 332], which eliminates trivial SCCs by removing any vertices that have zero in- or out-degree. We implement a procedure that recursively trims until no zero in- or out-degree vertices remain, or until a maximum number of rounds are reached, although in practice we found that a single trimming step is sufficient to remove the majority of trivial vertices on our graph inputs.

5.4 Covering Problems

Maximal Independent Set

The maximal independent set problem is to compute a subset of vertices U such that no two vertices in U are neighbors, and all vertices in $V \setminus U$ have a neighbor in U. Maximal

Algorithm 15 Maximal Independent Set

```
1:  P := RandomPermutation([0, ..., n − 1])
2:  Fl[0, n) := false
3:  Priority[0, n) := 0
4:  procedure NewlyCovered(s, d)
5:      if testAndSet(&Fl[d]) then
6:          return true
7:      return false
8:  procedure NewlyCoveredCond(v) return !Fl[v]
9:  procedure DecrementPriority(s, d)
10:     if P[s] < P[d] and fetchAndAdd(&Priority[d], −1) = 1 then
11:         return true
12:     return false
13: procedure DecrementPriorityCond(v) return Priority[v] > 0
14: procedure MIS(G(V, E))
15:     vertexMap(V, fn u → ▷ initialize priority to the number of neighbors appearing before u
    in P
16:         Priority[u] := G.getVertex(u).countNghs(fn (u, v) → return P[v] < P[u]))
17:     Roots := vertexSubset({v ∈ V | Priority[v] = 0})
18:     numFinished := 0
19:     I := {}
20:     while numFinished < n do
21:         I := I ∪ Roots
22:         Covered := edgeMap(G, Roots, NewlyCovered, NewlyCoveredCond)
23:         vertexMap(Covered, fn v → Priority[v] = 0)
24:         numFinished := numFinished + |Roots| + |Covered|
25:         Roots := edgeMap(G, Covered, DecrementPriority, DecrementPriorityCond)
26:     return I
```

independent set (MIS) and maximal matching (MM) are easily solved in linear work sequentially using greedy algorithms. Many efficient parallel maximal independent set and matching algorithms have been developed over the years [196, 224, 17, 182, 64, 57]. Blelloch et al. show that when the vertices (or edges) are processed in a random order, the sequential greedy algorithms for MIS and MM can be parallelized efficiently and give practical algorithms [64]. Recently, Fischer and Noever showed an improved depth bound for these MIS and MM algorithms [141].

In this chapter, we implement the rootset-based algorithm for MIS from Blelloch et al. [64] which runs in $O(m)$ expected work and $O(\lg^2 n)$ depth *whp* on the FA-BF-binary-forking model (using the improved depth analysis of Fischer and Noever [141]). To the best of our knowledge this is the first implementation of the rootset-based algorithm; the

implementations from [64] are based on processing appropriately-sized prefixes of an order generated by a random permutation P. Our implementation of the rootset-based algorithm works on a priority-DAG defined by directing edges in the graph from the higher-priority endpoint to the lower-priority endpoint. In each round, we add all roots of the DAG into the MIS, compute $N(Roots)$, the neighbors of the rootset that are still active, and finally decrement the priorities of $N(N(Roots))$. As the vertices in $N(Roots)$ are at arbitrary depths in the priority-DAG, we only decrement the priority along an edge (u, v), $u \in N(Roots)$ if $P[u] < P[v]$. The algorithm runs in $O(m)$ work as we process each edge once; the depth bound is $O(\lg^2 n)$ as the priority-DAG has $O(\lg n)$ depth *whp* [141], and each round takes $O(\lg n)$ depth. We were surprised that this implementation usually outperforms the prefix-based implementation from [64], while also being simple to implement.

Our implementation of the rootset-based MIS algorithm is shown in Algorithm 15. The algorithm first randomly orders the vertices with a random permutation P (Line 1). It then computes an array *Priority* where each vertex is associated with the count of its number of neighbors that have higher priority than it with respect to the permutation P. This computation is done using the countNghs primitive from Section 4.3 (Line 16). Next, on Line 17 we compute the initial rootset, *Roots*, which is the set of all vertices that initially have priority 0. In each round, the algorithm adds the roots to the independent set (Line 21), and computes the set of covered (i.e., removed) vertices, which are neighbors of the rootset that are still active ($Priority[v] > 0$). This step is done using edgeMap over *Roots*, where the map and condition function are defined similarly to BFS, returning true for a neighboring vertex if and only if it has not been visited before (the testAndSet to *Fl* succeeds). The algorithm also sets the *Priority* values of these vertices to 0 (Line 23). Next, the algorithm updates the number of finished vertices (Line 24). Finally, the algorithm computes the next set of roots using a second edgeMap. The map function (Lines 9–12) decrements the priority of all neighbors v visited over an edge (u, v) where $u \in Covered$ and $P[u] < P[v]$ using a fetchAndAdd that returns true for a neighbor v if this edge decrements its priority to 0.

Maximal Matching

The maximal matching problem is to compute a subset of edges $E' \subseteq E$ such that no two edges in E' share an endpoint, and all edges in $E \setminus E'$ share an endpoint with some edge in E'. Our maximal matching implementation is based on the prefix-based algorithm from [64] that takes $O(m)$ expected work and $O(\lg^2 m)$ depth *whp* on the PW-BF-binary-forking model (using the improved depth shown in [141]). We had to make several modifications to run the algorithm on the large graphs in our experiments. The original code from [64] uses an edgelist representation, but we cannot directly use this implementation as uncompressing all edges would require a prohibitive amount of memory for large graphs. Instead, as in our MSF implementation, we simulate the prefix-based approach by performing a constant

Algorithm 16 Maximal Matching

```
1:  Matched[0, ..., n) := false
2:  procedure ParallelGreedyMM(P)
3:      M := {}
4:      P := RandomPermutation(P)          ▷ a random permutation of the edges in the prefix
5:      while |P| > 0 do
6:          W := edges in P with no adjacent edges with higher rank
7:          ∀(u, v) ∈ W, set Matched[u] := true and Matched[v] := true
8:          P ←filter edges incident to newly matched vertices from P
9:      return M
10: procedure MaximalMatching(G(V, E))
11:     Matching := {}
12:     Rounds := 0
13:     while G.numEdges() > 0 do
14:         curM := G.numEdges()
15:         toExtract := if Rounds ≤ 5 then min(3n/2, curM) else curM
16:         P := extractEdges(G, fn (e = (u, v)) →
17:             inPrefix := e ∈ top toExtract highest-priority edges
18:             return u < v and inPrefix)        ▷ u < v to emit an edge in the prefix only once
19:         W := ParallelGreedyMM(P)
20:         packGraph(G, fn (e = (u, v)) → return !(e ∈ W or e incident to W))   ▷
    E := E \ (W ∪ N(W))
21:         Matching := Matching ∪ W
22:         Rounds := Rounds + 1
23:     return Matching
```

number of *filtering* steps. Each filter step packs out $3n/2$ of the highest priority edges, randomly permutes them, and then runs the edgelist based algorithm on the prefix. After computing the new set of edges that are added to the matching, we filter the remaining graph and remove all edges that are incident to matched vertices. In practice, just 3–4 filtering steps are sufficient to remove essentially all edges in the graph. The last step uncompresses any remaining edges into an edgelist and runs the prefix-based algorithm. The filtering steps can be done within the work and depth bounds of the original algorithm.

Our implementation of the prefix-based maximal matching algorithm from Blelloch et al. [64] is shown in Algorithm 16. The algorithm first creates the array *matched*, sets all vertices to be unmatched, and initializes the matching to empty (Line 11). The algorithm runs a constant number of filtering rounds, as described above, where each round fetches some number of highest priority edges that are still active (i.e., neither endpoint is incident to a matched edge). First, it calculates the number of edges to extract (Line 15). It then

extracts the highest priority edges using the PACKGRAPH primitive. The function supplied to PACKGRAPH checks whether an edge e is one of the highest priority edges, and if so, emits it in the output edge array, P and removes this edge from the graph. Our implementation calculates edge priorities by hashing the edge pair. It selects whether an edge is in the prefix by comparing each edge's priority with the priority of approximately the *toExtract*'th smallest priority, computed using approximate median.

Next, the algorithm applies the parallel greedy maximal matching algorithm (Lines 2–9) on it. The parallel greedy algorithm first randomly permutes the edges in the prefix (Line 4). It then repeatedly finds the set of edges that have the lowest rank in the prefix amongst all other edges incident to either endpoint (Line 6), adds them to the matching (Line 7), and filters the edges based on the newly matched edges (Line 8). The edges matched by the greedy algorithm are returned to the MaximalMatching procedure (Line 9). We refer to [64, 141] for a detailed description of the prefix-based algorithm that we implement, and a proof of the work and depth of the PARALLELGREEDYMM algorithm.

The last steps within a round are to filter the remaining edges in the graph based on the newly matched edges using the PACKGRAPH primitive (Line 20). The supplied predicate does not return any edges in the output edgearray, and packs out any edge incident to the partial matching, W. Lastly, the algorithm adds the newly matched edges to the matching Line 21. We note that applying a constant number of filtering rounds before executing PARALLELGREEDYMM does not affect the work and depth bounds.

Graph Coloring

The graph coloring problem is to compute a mapping from each $v \in V$ to a color such that for each edge $(u, v) \in E$, $C(u) \neq C(v)$, using at most $\Delta + 1$ colors. As graph coloring is NP-hard to solve optimally, algorithms like greedy coloring, which guarantees a $(\Delta + 1)$-coloring, are used instead in practice, and often use much fewer than $(\Delta + 1)$ colors on real-world graphs [367, 169]. Jones and Plassmann (JP) parallelize the greedy algorithm using linear work, but unfortunately adversarial inputs exist for the heuristics they consider that may force the algorithm to run in $O(n)$ depth. Hasenplaugh et al. introduce several heuristics that produce high-quality colorings in practice and also achieve provably low-depth regardless of the input graph. These include LLF (largest-log-degree-first), which processes vertices ordered by the log of their degree and SLL (smallest-log-degree-last), which processes vertices by removing all lowest log-degree vertices from the graph, coloring the remaining graph, and finally coloring the removed vertices. For LLF, they show that it runs in $O(m + n)$ work and $O(L \lg \Delta + \lg n)$ depth, where $L = \min\{\sqrt{m}, \Delta\} + \lg^2 \Delta \lg n / \lg \lg n$ in expectation.

In this chapter, we implement a synchronous version of Jones-Plassmann using the LLF heuristic, which runs in $O(m + n)$ work and $O(L \lg \Delta + \lg n)$ depth on the FA-BF-binary-forking model. The algorithm is implemented similarly to our rootset-based algorithm

Algorithm 17 LLF Graph Coloring

```
1:  P := RandomPermutation([0, ..., n − 1])
2:  Color[0, ..., n) := ∞
3:  D[0, ..., n) := 0
4:  Priority[0, n) := 0
5:  procedure AssignColors(u)
6:      Color[u] := c, where c is the first unused color in N(u)
7:  procedure DecrementPriority(s, d)
8:      if fetchAndAdd(&Priority[d], −1) = 1 then return true
9:      return false
10: procedure DecrementPriorityCond(v) return Priority[v] > 0
11: procedure LLF(G(V, E))
12:     vertexMap(V, fn u → D[u] := ⌈lg(d(u))⌉)
13:     countFn := fn (u, v) → return D[v] > D[u] or (D[v] = D[u] and P[v] < P[u])
14:     vertexMap(V, fn u → Priority[u] := G.getVertex(u).countNghs(countFn))
15:     Roots := vertexSubset({v ∈ V | Priority[v] = 0})
16:     Finished := 0
17:     while Finished < n do
18:         vertexMap(Roots, AssignColors)
19:         Finished := Finished + |Roots|
20:         Roots := edgeMap(G, Roots, DecrementPriority, DecrementPriorityCond)
21:     return C
```

for MIS. In each round, after coloring the roots we use a fetchAndAdd to decrement a count on our neighbors, and add the neighbor as a root on the next round if the count is decremented to 0.

Algorithm 17 shows our synchronous implementation of the parallel LLF-Coloring algorithm from [169]. The algorithm first computes priorities for each vertex in parallel using the countNghs primitive (Line 14). This step computes the number of neighbors of a vertex that must run before it by applying the *countFn* predicate (Line 13). This predicate function returns true for a (u, v) edge to a neighbor v if the log-degree of v is greater than u, or, if the log-degrees are equal whether v has a lower-rank in a permutation on the vertices (Line 1) than v. Next, the algorithm computes the vertexSubset *Roots* (Line 15) which consists of all vertices that have no neighbors that are still uncolored that must be run before them based on *countFn*. Note that *Roots* is an independent set. The algorithm then loops while some vertex remains uncolored. Within the loop, it first assigns colors to the roots in parallel (Line 18) by setting each root to the first unused color in its neighborhood (Lines 5–6). Finally, it updates the number of finished vertices by the number of roots (Line 19) and computes the next rootset by applying edgeMap on the

rootset with a map function that decrements the priority over all (u, v) edges incident to *Roots* where $Priority[v] > 0$. The map function returns true only if the priority decrement decreases the priority of the neighboring vertex to 0 (Line 8).

Approximate Set Cover

The set cover problem can be modeled by a bipartite graph where sets and elements are vertices, with an edge between a set and an element if and only if the set covers that element. The approximate set cover problem is as follows: given a bipartite graph $G = (V = (S, E), A)$ representing an unweighted set cover instance, compute a subset $S' \subseteq S$ such that $\cup_{s \in S'} N(s) = E$ and $|S'|$ is an $O(\lg n)$-approximation to the optimal cover. Like graph coloring, the set cover problem is NP-hard to solve optimally, and a sequential greedy algorithm computes an H_n-approximation in $O(m)$ time for unweighted sets, and $O(m \lg m)$ time for weighted sets, where $H_n = \sum_{k=1}^{n} 1/k$ and m is the sum of the sizes of the sets (or the number of edges in the graph). There has been significant work on finding work-efficient parallel algorithms that achieves an H_n-approximation [54, 283, 67, 68, 207].

Algorithm 18 shows pseudocode for the Blelloch et al. algorithm [67] which runs in $O(m)$ work and $O(\lg^3 n)$ depth on the PW-BF-binary-forking model. Our presentation here is based on the bucketing-based implementation from Julienne [115], with one significant change regarding how sets acquire elements which we discuss below. The algorithm first buckets the sets based on their degree, placing a set covering D elements into $\lfloor \lg_{1+\epsilon} D \rfloor$'th bucket (Line 24). It then processes the buckets in decreasing order (Lines 26–38). In each round, the algorithm extracts the highest bucket (*Sets*) (Line 26) and packs out the adjacency lists of vertices in this bucket to remove edges to neighbors that are covered in prior rounds (Line 27). The output is an augmented vertexSubset, *SetsD*, containing each set along with its new degree after packing out all dead edges. It then maps over *SetsD*, updating the degree in D for each set with the new degree (Line 28). The algorithm then filters *SetsD* to build a vertexSubset *Active*, which contains sets that have sufficiently high degree to continue in this round (Line 29).

The next few steps of the algorithm implement one step of MaNIS (Maximal Nearly-Independent Set) [67], to compute a set of sets from *Active* that have little overlap. First, the algorithm assigns a random priority to each currently active set using a random permutation, storing the priorities in the array π (Lines 30–31). Next, it applies EDGEMAP (Line 32) where the map function (Line 12) uses a priority-write on each (s, e) edge to try and acquire an element e using the priority of the visiting set, $\pi[s]$. It then computes the number of elements each set successfully acquired using the SRCCOUNT primitive (Line 33) with the predicate WONELM (Line 10) that checks whether the minimum value stored at an element is the unique priority for the set. The final MaNIS step maps over the vertices and the number of elements they successfully acquired (Line 34) with the map function WONENOUGH (Lines 13–16) which adds sets that covered enough elements to the cover.

Algorithm 18 Approximate Set Cover

```
1: El[0, ..., |E|) := ∞
2: Fl[0, ..., |E|) := uncovered
3: D[0, ..., |S|) := {deg(s_0), ..., deg(s_{n-1})}        ▷ initialized to the initial degree of s ∈ S
4: π[0, |S|) := 0  ▷ map from sets to priorities; entries are updated on each round for active sets
5: b                                                      ▷ current bucket number
6: procedure BucketNum(s) return ⌊lg_{1+ε} D[s]⌋
7: procedure ElmUncovered(s, e) return Fl[e] = uncovered
8: procedure UpdateD(s, d) D[s] := d
9: procedure AboveThreshold(s, d) return d ≥ ⌈(1 + ε)^{max(b,0)}⌉
10: procedure WonElm(s, e) return π[s] = El[e]
11: procedure InCover(s) return D[s] = ∞
12: procedure VisitElms(s, e) priorityWrite(&El[e], π[s], <)
13: procedure WonEnough(s, elmsWon)
14:     threshold := ⌈(1 + ε)^{max(b−1,0)}⌉
15:     if (elmsWon > threshold) then
16:         D[s] := ∞                                     ▷ places s in the set cover
17: procedure ResetElms(s, e)
18:     if (El[e] = s) then
19:         if (InCover(s)) then
20:             Fl[e] := covered                          ▷ e is covered by s
21:         else
22:             El[e] := ∞                                ▷ reset e
23: procedure SetCover(G := (S ∪ E, A))
24:     B := makeBuckets(|S|, BucketNum, decreasing)      ▷ process from largest to smallest
    log-degree
25:     (b, Sets) := B.nextBucket()
26:     while (b ≠ nullbkt) do
27:         SetsD := srcPack(G, Sets, ElmUncovered)       ▷ pack out edges to covered elements
28:         vertexMap(SetsD, UpdateD)                     ▷ update set degrees in D
29:         Active := vertexFilter(SetsD, AboveThreshold) ▷ extract sets with sufficiently
    high degree
30:         π_A := RandomPermutation(|Active|)
31:         ∀i ∈ [0, |Active|), set π[Active[i]] := π_A[i]  ▷ assign each active set a random priority
32:         edgeMap(G, Active, VisitElms, ElmUncovered)   ▷ active sets try to acquire incident
    elements
33:         ActiveCts := srcCount(G, Active, WonElm)      ▷ count number of neighbors won by
    each set
34:         vertexMap(ActiveCts, WonEnough)               ▷ place sets that won enough into the cover
35:         edgeMap(G, Active, ResetElms)                 ▷ update neighboring elements state based on set
    status
36:         Rebucket :={(s, B.getBucket(b, BucketNum(s)) | s ∈ Sets and !InCover(s)}
37:         B.updateBuckets(Rebucket)                     ▷ update buckets of sets that failed to join the cover
38:         (b, Sets) := B.nextBucket()
39:     return {s ∈ S | InCover(s) = true}
```

The final step in a round is to *rebucket* all sets which were not added to the cover to be processed in a subsequent round (Lines 36–37). The rebucketed sets are those in *Sets* that were not added to the cover, and the new bucket they are assigned to is calculated by using the GETBUCKET primitive with the current bucket, b, and a new bucket calculated based on their updated degree (Line 6).

Our implementation of approximate set cover in this thesis is based on the implementation from Julienne in Chapter 4, and we refer the reader to this chapter for more details about the bucketing-based implementation. The main change we made in this chapter is to ensure that we correctly set random priorities for active sets in each round of the algorithm. Both the implementation in Julienne as well as an earlier implementation of the algorithm [68] use the original IDs of sets instead of picking random priorities for all sets that are active on a given round. This approach can cause very few vertices to be added in each round on meshes and other graphs with a large amount of symmetry. Instead, in our implementation, for $\mathcal{A}_S$, the active sets on a round, we generate a random permutation of $[0, \ldots, |\mathcal{A}_S| - 1]$ and write these values into a pre-allocated dense array with size proportional to the number of sets (Lines 30—31). We give experimental details regarding this change in Section 5.8.

5.5 Substructure Problems

k-core

A ***k-core*** of a graph is a maximal subgraph H where the degree of every vertex in H is $\geq k$. The defncoreness of a vertex is the maximum k-core a vertex participates in. The k-core problem in this thesis is to compute a mapping from each vertex to its coreness value. k-cores were defined independently by Seidman [305], and by Matula and Beck [232] who also gave a linear-time algorithm for computing the coreness value of all vertices. Anderson and Mayr showed that k-core (and therefore coreness) is in NC for $k \leq 2$, but is P-complete for $k \geq 3$ [23]. The Matula and Beck algorithm is simple and practical—it first bucket-sorts vertices by their degree, and then repeatedly deletes the minimum-degree vertex. The affected neighbors are moved to a new bucket corresponding to their induced degree. As each edge in each direction and vertex is processed exactly once, the algorithm runs in $O(m + n)$ work. In Chapter 4, we presented a parallel algorithm based on the bucketing interface and data structure designed in that chapter that runs in runs in $O(m+n)$ expected work, and $\rho \lg n$ depth *whp.* ρ is the peeling-complexity of the graph, defined as the number of rounds to peel the graph to an empty graph where each peeling step removes all minimum degree vertices. The k-core algorithm studied in this chapter is the same algorithm and implementation as presented in Chapter 4, and we avoid presenting it redundantly here.

Algorithm 19 Approximate Densest Subgraph

```
1:  D[0, ..., n) := 0
2:  procedure ApproximateDensestSubgraph(G(V, E))
3:      vertexMap(V, fn v → D[v] := d(v))        ▷ induced degrees are initially original degrees
4:      S := V
5:      S_max := ∅
6:      while S ≠ ∅ do
7:          R := vertexSubset({v ∈ S | D[v] < 2(1 + ε)D(S)})        ▷ D(S) := |E(G[S])| / |S|
8:          vertexMap(R, fn v → return A[v] := 0)
9:          condFn := fn v → return true
10:         applyFn := fn (v, edgesRemoved) →
11:             D[v] := D[v] − edgesRemoved
12:             return None
13:         nghCount(G, R, condFn, applyFn)
14:         S := S \ R
15:         if D(S) > D(S_max) then
16:             S_max := S
17:     return S_max
```

Approximate Densest Subgraph

The densest subgraph problem is to find a subset of vertices in an undirected graph with the highest density. The density of a subset of vertices S is the number of edges in the subgraph S divided by the number of vertices. The approximate densest problem is to compute a subset $U \subseteq V$ s.t. the density of U is a $2(1+\epsilon)$ approximation of the density of the densest subgraph of G.

The problem is a classic graph optimization problem that admits exact polynomial-time solutions using either a reduction to flow [153] or LP-rounding [96]. In his paper, Charikar also gives a simple $O(m+n)$ work 2-approximation algorithm based on computing a degeneracy ordering of the graph, and taking the maximum density subgraph over all suffixes of the degeneracy order.[4] The problem has also received attention in parallel models of computation [38, 37]. Bahmani et al. give a $(2+\epsilon)$-approximation running in $O(\lg_{1+\epsilon} n)$ rounds of MapReduce [38]. Subsequently, Bahmani et al. [37] showed that a $(1+\epsilon)$ can be found in $O(\lg n/\epsilon^2)$ rounds of MapReduce by using the multiplicative-weights approach on the dual of the natural LP for densest subgraph. To the best our knowledge, it is open whether the densest subgraph problem can be exactly solved in NC.

In this chapter, we implement the elegant $(2+\epsilon)$-approximation algorithm of Bahmani

[4]We note that the 2-approximation can be work-efficiently solved in the same depth as our k-core algorithm by augmenting the k-core algorithm to return the order in which vertices are peeled. Computing the maximum density subgraph over suffixes of the degeneracy order can be done using scan.

et al. (Algorithm 19). Our implementation of the algorithm runs in $O(m+n)$ work and $O(\lg_{1+\epsilon} n \lg n)$ depth. The algorithm starts with a candidate subgraph, S, consisting of all vertices, and an empty approximate densest subgraph $S_{\max}$ (Lines4–5). It also maintains an array with the induced degree of each vertex in the array D, which is initially just its degree in G (Line 3). The main loop iteratively peels vertices with degree below the density threshold in the current candidate subgraph (Lines 6–16). Specifically, it first finds all vertices with induced degree less than $2(1+\epsilon)\mathcal{D}(S)$ (Line 7). Next, it calls NGHCOUNT (see Section 4.3), which computes for each neighbor of R the number of incident edges removed by deleting vertices in R from the graph, and updates the neighbor's degree in D (Line 11). Finally, it removes vertices in R from S (Line 14). If the density of the updated subgraph S is greater than the density of $S_{\max}$, the algorithm updates $S_{\max}$ to be S.

Bahmani et al. show that this algorithm removes a constant factor of the vertices in each round, but do not consider the work or total number of operations performed by their algorithm. We briefly sketch how the algorithm can be implemented in $O(m+n)$ work and $O(\lg_{1+\epsilon} n \lg n)$ depth. Instead of computing the density of the current subgraph by scanning all edges, we maintain it explicitly in an array, D (Line 3), and update it as vertices are removed from S. Each round of the algorithm does work proportional to vertices in S to compute R (Line 7) but since S decreases by a constant factor in each round the work of these steps is $O(n)$ over all rounds. Computing the new density can be done by computing the number of edges between R and S, which only requires scanning edges incident to vertices in R using NGHCOUNT (Line 13). Therefore, the edges incident to a vertex are scanned exactly once, in the round when it is included in R, and so the algorithm performs $O(m+n)$ work. The depth is $O(\lg_{1+\epsilon} n \lg n)$ since there are $O(\lg_{1+\epsilon} n)$ rounds each of which perform a filter and NGHCOUNT which both run in $O(\lg n)$ depth.

We note that an earlier implementation of our algorithm used the EDGEMAP primitive combined with FETCHANDADD to decrement degrees of neighbors of R. We found that since a large number of vertices are removed in each round, using FETCHANDADD can cause contention, especially on graphs containing vertices with high degrees. Our implementation uses a work-efficient histogram procedure to implement NGHCOUNT (see Section 5.7) which updates the degrees while incurring very little contention.

Triangle Counting

The triangle counting problem is to compute the global count of the number of triangles in the graph. Triangle counting has received significant recent attention due to its numerous applications in Web and social network analysis. There have been dozens of papers on sequential triangle counting [183, 16, 302, 301, 210, 265, 268]. The fastest algorithms rely on matrix multiplication and run in either $O(n^{\omega})$ or $O(m^{2\omega/(1+\omega)})$ work, where ω is the best matrix multiplication exponent [183, 16]. The fastest algorithm that does not rely matrix multiplication requires $O(m^{3/2})$ work [302, 301, 210], which also turns out to be

Algorithm 20 Triangle Counting

```
1: procedure FilterEdge(u, v)
2:     return d(u) > d(v) or (d(u) = d(v) and u > v)
3: procedure TriangleCounting(G(V, E))
4:     G' := filterGraph(G, FilterEdge)          ▷ orient edges from lower to higher degree
5:     vertexCounts[0, ..., n) := 0
6:     vertexMap(V, fn u → vertexCounts[u] :=
7:                 G.getVertex(u).reduceOutNgh(fn (u, v) → return intersection(N+(u), N+(v)), (0, +)))
8:     return reduce(vertexCounts, (0, +))
```

much more practical. Parallel algorithms with $O(m^{3/2})$ work have been designed [323, 3, 223], with Shun and Tangwongsan [323] showing an algorithm that requires $O(\lg n)$ depth on the binary-forking model.[5] The implementation from [323] parallelizes Latapy's *compact-forward* algorithm, which creates a directed graph DG where an edge $(u, v) \in E$ is kept in DG iff $deg(u) < deg(v)$. Although triangle counting can be done directly on the undirected graph in the same work and depth asymptotically, directing the edges helps reduce work, and ensures that every triangle is counted exactly once.

In this chapter we implement the triangle counting algorithm described in [323] (Algorithm 20). The algorithm first uses the filterGraph primitive (Line 4) to direct the edges in the graph from lower-degree to higher-degree, breaking ties lexicographically (Line 2). It then maps over all vertices in the graph in parallel (Line 6), and for each vertex performs a sum-reduction over its out-neighbors, where the value for each neighbor is the intersection size between the directed neighborhoods $N^+(u)$ and $N^+(v)$ (Line 7).

We note that we had to make several significant changes to the implementation in order to run efficiently on large compressed graphs. First, we parallelized the creation of the directed graph; this step creates a directed graph encoded in the parallel-byte format in $O(m)$ work and $O(\lg n)$ depth using the filterGraph primitive. We also parallelized the merge-based intersection algorithm to make it work in the parallel-byte format. We give more details on these techniques in Section 5.7.

5.6 *Eigenvector Problems*

PageRank

PageRank is a centrality algorithm first used at Google to rank webpages [85]. The algorithm takes a graph $G = (V, E)$, a damping factor $0 \leq \gamma \leq 1$ and a constant ϵ which

[5] The algorithm in [323] was described in the Parallel Cache Oblivious model, with a depth of $O(\lg^{3/2} n)$.

Algorithm 21 PageRank

```
1:  P_curr[0,...,n) := 1/n
2:  P_next[0,...,n) := 0
3:  diffs[0,...,n) := 0
4:  procedure PageRank(G)
5:      Frontier := vertexSubset({0,...,n-1})
6:      condFn := fn u -> return true
7:      mapFn := fn (u,v) -> return P_curr[u]/d(u)
8:      applyFn := fn (v, contribution) ->
9:          P_next[v] := γ * contribution + (1-γ)/n
10:         diffs[v] := |P_next[v] - P_curr[v]|
11:         return None
12:     error := ∞
13:     while (error < ε) do
14:         nghReduceApply(G, ids, mapFn, (0,+), condFn, applyFn)
15:         error := reduce(diffs, (0,+))
16:         swap(P_curr, P_next)
17:     return P_curr
```

controls convergence. Initially, the PageRank of each vertex is $1/n$. In each iteration, the algorithm updates the PageRanks of the vertices using the following equation:

$$P_v = \frac{1-\gamma}{n} + \gamma \sum_{u \in N^-(v)} \frac{P_u}{deg^+(u)}$$

The PageRank algorithm terminates once the l_1 norm of the differences between PageRank values between iterations is below ϵ. The algorithm implemented in this chapter is an extension of the implementation of PageRank described in Ligra [319]. The main changes are using a contention-avoiding reduction primitive, which we describe in more detail below. Some PageRank implementations used in practice actually use an algorithm called PageRank-Delta [222], which modifies PageRank by only activating a vertex if its PageRank value has changed sufficiently. However, the work and depth of this algorithm are the same as that of PageRank in the worst case, and therefore we chose to implement the classic algorithm.

We show pseudocode for our PageRank implementation in Algorithm 21. The initial PageRank values are set to $1/n$ (Line 1). The algorithm initializes a frontier containing all vertices (Line 5), and sets the error (the l_1 norm between consecutive PageRank vectors) to ∞ (Line 12). The algorithm then iterates the PageRank update step while the error is above the threshold ϵ (Lines 13–16). The update is implemented using the nghReduce primitive (see Section 4.3 for details on the primitive). The *condFn* function (Line 6)

specifies that value should be aggregated for each vertex with non-zero in-degree. The *mapFn* function pulls a PageRank contribution of $P_{curr}[u]/d(u)$ for each in-neighbor u in the frontier (Line 7). Finally, after the contributions to each neighbor have been summed up, the *applyFn* function is called on a pair of a neighboring vertex v, and its contribution (Lines 8–11). The apply step updates the next PageRank value for the vertex using the PageRank equation above (Line 9) and updates the difference in PageRank values for this vertex in the *diffs* vector (Line 10). The last steps in the loop applies a parallel reduction over the differences vector to update the current error (Line 15) and finally swaps the current and next PageRank vectors (Line 16).

The main modification we made to the implementation from Ligra was to implement the dense iterations of the algorithm using the reduction primitive nghReduce, which can be carried out over the incoming neighbors of a vertex in parallel, without using a fetchAndAdd instruction. Each iteration of our implementation requires $O(m+n)$ work and $O(\lg n)$ depth (note that the bounds hold deterministically since in each iteration we can apply a dense, or pull-based implementation which performs a reduction over the in-neighbors of each vertex). As the number of iterations required for PageRank to finish for a given ϵ depends on the structure of the input graph, our benchmark measures the time for a single iteration of PageRank.

5.7 Implementations and Techniques

In this section, we introduce several general implementation techniques and optimizations that we use in our algorithms. The techniques include a fast histogram implementation useful for reducing contention in the k-core algorithm, a cache-friendly sparse edgeMap implementation that we call edgeMapBlocked, and compression techniques used to efficiently parallelize algorithms on massive graphs.

5.7.1 A Work-efficient Histogram Implementation

Our initial implementation of the peeling-based algorithm for k-core algorithm suffered from poor performance due to a large amount of contention incurred by fetchAndAdds on high-degree vertices. This occurs as many social-networks and web-graphs have large maximum degree, but relatively small degeneracy, or largest non-empty core (labeled k_{max} in Table 2.1). For these graphs, we observed that many early rounds, which process vertices with low coreness perform a large number of fetchAndAdds on memory locations corresponding to high-degree vertices, resulting in high contention [326]. To reduce contention, we designed a work-efficient histogram implementation that can perform this step while only incurring $O(\lg n)$ contention *whp* The ***Histogram*** primitive takes a sequence of $(\mathbf{K}, \mathbf{T})$ pairs, and an associative and commutative operator $R : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$

and computes a sequence of $(\mathbf{K}, \mathbf{T})$ pairs, where each key k only appears once, and its associated value t is the sum of all values associated with keys k in the input, combined with respect to R.

A useful example of histogram to consider is summing for each $v \in N(F)$ for a vertexSubset F, the number of edges (u, v) where $u \in F$ (i.e., the number of incoming neighbors from the frontier). This operation can be implemented by running histogram on a sequence where each $v \in N(F)$ appears once per (u, v) edge as a tuple $(v, 1)$ using the operator +. One theoretically efficient implementation of histogram is to simply semisort the pairs using the work-efficient semisort algorithm from [161]. The semisort places pairs from the sequence into a set of *heavy* and *light* buckets, where heavy buckets contain a single key that appears many times in the input sequence, and light buckets contain at most $O(\lg^2 n)$ distinct keys (k, v) keys, each of which appear at most $O(\lg n)$ times *whp* (heavy and light keys are determined by sampling). We compute the reduced value for heavy buckets using a standard parallel reduction. For each light bucket, we allocate a hash table, and hash the keys in the bucket in parallel to the table, combining multiple values for the same key using R. As each key appears at most $O(\lg n)$ times *whp* we incur at most $O(\lg n)$ contention *whp* The output sequence can be computed by compacting the light tables and heavy arrays.

While the semisort implementation is theoretically efficient, it requires a likely cache miss for each key when inserting into the appropriate hash table. To improve cache performance in this step, we implemented a work-efficient algorithm with $O(n^\epsilon)$ depth based on radix sort. Our implementation is based on the parallel radix sort from PBBS [324]. As in the semisort, we first sample keys from the sequence and determine the set of heavy-keys. Instead of directly moving the elements into light and heavy buckets, we break up the input sequence into $O(n^{1-\epsilon})$ blocks, each of size $O(n^\epsilon)$, and sequentially sort the keys within a block into light and heavy buckets. Within the blocks, we reduce all heavy keys into a single value and compute an array of size $O(n^\epsilon)$ which holds the starting offset of each bucket within the block. Next, we perform a segmented-scan [60] over the arrays of the $O(n^{1-\epsilon})$ blocks to compute the sizes of the light buckets, and the reduced values for the heavy-buckets, which only contain a single key. Finally, we allocate tables for the light buckets, hash the light keys in parallel over the blocks and compact the light tables and heavy keys into the output array. Each step runs in $O(n)$ work and $O(n^\epsilon)$ depth. Compared to the original semisort implementation, this version incurs fewer cache misses because the light keys per block are already sorted and consecutive keys likely go to the same hash table, which fits in cache. We compared our times in the histogram-based version of k-core and the FETCHANDADD-based version of k-core and saw between a 1.1–3.1x improvement from using the histogram.

Algorithm 22 EDGEMAPBLOCKED

```
1:  procedure EDGEMAPBLOCKED(G, U, F)
2:      O := Prefix sums of degrees of u ∈ U
3:      d_U := Σ_{u∈U} deg(u)
4:      nblocks := ⌈d_U / bsize⌉
5:      B := Result of binary search for nblocks indices into O
6:      I := Intermediate array of size Σ_{u∈U} deg(u)
7:      A := Intermediate array of size nblocks
8:      parfor i ∈ B do
9:          Process work in B[i] and pack live neighbors into I[ibsize]
10:         A[i] := Number of live neighbors
11:     R := Prefix sum A and compact I
12:     return R
```

5.7.2 EDGEMAPBLOCKED

One of the core primitives used by our algorithms is EDGEMAP (described in Chapter 2 and Section 3.7). The push-based version of EDGEMAP, EDGEMAPSPARSE, takes a frontier U and iterates over all (u, v) edges incident to it. It applies an update function on each edge that returns a boolean indicating whether or not the neighbor should be included in the next frontier. The standard implementation of EDGEMAPSPARSE first computes prefix-sums of $deg(u), u \in U$ to compute offsets, allocates an array of size $\sum_{u \in U} deg(u)$, and iterates over all $u \in U$ in parallel, writing the ID of the neighbor to the array if the update function F returns *true*, and $\perp$ otherwise. It then filters out the $\perp$ values in the array to produce the output vertexSubset.

In real-world graphs, $|N(U)|$, the number of unique neighbors incident to the current frontier is often much smaller than $\sum_{u \in U} deg(u)$. However, EDGEMAPSPARSE will always perform $\sum_{u \in U} deg(u)$ writes and incur a proportional number of cache misses, despite the size of the output being at most $|N(U)|$. More precisely, the size of the output is at most $LN(U) \leq |N(U)|$, where $LN(U)$ is the number of *live neighbors* of U, where a live neighbor is a neighbor of the current frontier for which F returns *true*. To reduce the number of cache misses we incur in the push-based traversal, we implemented a new version of EDGEMAPSPARSE that performs at most $LN(U)$ writes that we call EDGEMAPBLOCKED. The idea behind EDGEMAPBLOCKED is to logically break the edges incident to the current frontier up into a set of blocks, and iterate over the blocks sequentially, packing live neighbors, compactly for each block. We then simply prefix-sum the number of live neighbors per-block, and compact the block outputs into the output array.

We now describe a theoretically efficient implementation of EDGEMAPBLOCKED (Algorithm 22). As in EDGEMAPSPARSE, we first compute an array of offsets O (Line 1) by prefix summing the degrees of $u \in U$. We process the edges incident to this frontier in blocks

of size *bsize*. As we cannot afford to explicitly write out the edges incident to the current frontier to block them, we instead logically assign the edges to blocks. Each block searches for a range of vertices to process with *bsize* edges; the i'th block binary searches the offsets array to find the vertex incident to the start of the $(i \cdot bsize)$'th edge, storing the result into $B[i]$ (Lines 4–5). The vertices that block i must process are therefore between $B[i]$ and $B[i+1]$. We note that multiple blocks can be assigned to process the edges incident to a high-degree vertex. Next, we allocate an intermediate array I of size d_U (Line 6), but do not initialize the memory, and an array A that stores the number of live neighbors found by each block (Line 7). Next, we process the blocks in parallel by sequentially applying F to each edge in the block and compactly writing any live neighbors to $I[i \cdot bsize]$ (Line 9), and write the number of live neighbors to $A[i]$ (Line 10). Finally, we do a prefix sum on A, which gives offsets into an array of size proportional to the number of live neighbors, and copy the live neighbors in parallel to R, the output array (Line 11).

We found that this optimization helps the most in algorithms where there is a significant imbalance between the size of the output of each EDGEMAP, and $\sum_{u \in U} deg(u)$. For example, in weighted BFS, relatively few of the edges actually relax a neighboring vertex, and so the size of the output, which contains vertices that should be moved to a new bucket, is usually much smaller than the total number of edges incident to the frontier. In this case, we observed as much as a 1.8x improvement in running time by switching from EDGEMAPSPARSE to EDGEMAPBLOCKED.

5.7.3 Techniques for overlapping searches

In this section, we describe how we compute and update the reachability labels for vertices that are visited in a phase of our SCC algorithm. Recall that each phase performs a graph traversal from the set of active centers on this round, C_A, and computes for each center c, all vertices in the weakly-connected component for the subproblem of c that can be reached by a directed path from it. We store this reachability information as a set of (u, c_i) pairs in a hash-table, which represent the fact that u can be reached by a directed path from c_i. A phase performs two graph traversals from the centers to compute $\mathcal{R}_F$ and $\mathcal{R}_B$, the out-reachability set and in-reachability sets respectively. Each traversal allocates an initial hash table and runs rounds of EDGEMAP until no new label information is added to the table.

The main challenge in implementing one round in the traversal is (1) ensuring that the table has sufficient space to store all pairs that will be added this round, and (2) efficiently iterating over all of the pairs associated with a vertex. We implement (1) by performing a parallel reduce to sum over vertices $u \in F$, the current frontier, the number of neighbors v in the same subproblem, multiplied by the number of distinct labels currently assigned to u. This upper-bounds the number of distinct labels that could be added this round, and although we may overestimate the number of actual additions, we will never run out

of space in the table. We update the number of elements currently in the table during concurrent insertions by storing a per-processor count which gets incremented whenever the processor performs a successful insertion. The counts are then summed together at the end of a round and used to update the count of the number of elements in the table.

One simple implementation of (2) is to simply allocate $O(\lg n)$ space for every vertex, as the maximum number of centers that visit any vertex during a phase is at most $O(\lg n)$ *whp* However, this will waste a significant amount of space, as most vertices are visited just a few times. Instead, our implementation stores (u, c) pairs in the table for visited vertices u, and computes hashes based only on the ID of u. As each vertex is only expected to be visited a constant number of times during a phase, the expected probe length is still a constant. Storing the pairs for a vertex in the same probe-sequence is helpful for two reasons. First, we may incur fewer cache misses than if we had hashed the pairs based on both entries, as multiple pairs for a vertex can fit in the same cache line. Second, storing the pairs for a vertex along the same probe sequence makes it extremely easy to find all pairs associated with a vertex u, as we simply perform linear-probing, reporting all pairs that have u as their key until we hit an empty cell. Our experiments show that this technique is practical, and we believe that it may have applications in similar algorithms, such as computing least-element lists or FRT trees in parallel [75, 66].

5.7.4 Primitives on Compressed Graphs

Many of our algorithms are concisely expressed using fundamental primitives such as map, map-reduce, filter, pack, and intersection. To run our algorithms without any modifications on compressed graphs, we wrote new implementations of these primitives using using the parallel-byte format from Ligra+, some of which required some new techniques in order to be theoretically efficient. We first review the byte and parallel-byte formats from [322]. In byte coding, we store a vertex's neighbor list by difference encoding consecutive vertices, with the first vertex difference encoded with respect to the source. Decoding is done by sequentially uncompressing each difference, and summing the differences into a running sum which gives the ID of the next neighbor. As this process is sequential, graph algorithms using the byte format that map over the neighbors of a vertex will require $\Omega(\Delta)$ depth. The parallel-byte format from Ligra+ breaks the neighbors of a high-degree vertex into blocks, where each block contains a constant number of neighbors. Each block is difference encoded with respect to the source. As each block can have a different size, it also stores offsets that point to the start of each block. The format stores the blocks in a neighbor list L in sorted order.

We now describe efficient implementations of primitives used by our algorithms. All descriptions are given for neighbor lists coded in the parallel-byte format. The ***Map*** primitive takes as input neighbor list L, and a map function F, and applies F to each ID in L. This can be implemented with a parallel-for loop across the blocks, where each iteration

decodes its block sequentially. Our implementation of map runs in $O(|L|)$ work and $O(\lg n)$ depth. ***Map-Reduce*** takes as input a neighbor list L, a map function $F : \mathbf{vtx} \to \mathbf{T}$ and a binary associative function R and returns the sum of the mapped elements with respect to R. We perform map-reduce similarly by first mapping over the blocks, then sequentially reducing over the mapped values in each block. We store the accumulated value on the stack or in an allocated array if the number of blocks is large enough. Finally, we reduce the accumulated values using R to compute the output. Our implementation of map-reduce runs in $O(|L|)$ work and $O(\lg n)$ depth.

Filter takes as input a neighbor list L, a predicate P, and an array T into which the vertices satisfying P are written, in the same order as in L. Our implementation of filter also takes as input an array S, which is an array of size $deg(v)$ space for lists L larger than a constant threshold, and null otherwise. In the case where L is large, we implement the filter by first decoding L into S in parallel; each block in L has an offset into S as every block except possibly the last block contains the same number of vertex IDs. We then filter S into the output array T. In the case where L is small we just run the filter sequentially. Our implementation of filter runs in $O(|L|)$ work and $O(\lg n)$ depth. ***Pack*** takes as input a neighbor list L and a predicate P function, and packs L, keeping only vertex IDs that satisfied P. Our implementation of pack takes as input an array S, which an array of size $2 * deg(v)$ for lists larger than a constant threshold, and null otherwise. In the case where L is large, we first decode L in parallel into the first $deg(v)$ cells of S. Next, we filter these vertices into the second $deg(v)$ cells of S, and compute the new length of L. Finally, we recompress the blocks in parallel by first computing the compressed size of each new block. We prefix-sum the sizes to calculate offsets into the array and finally compress the new blocks by writing each block starting at its offset. When L is small we just pack L sequentially. We make use of the pack and filter primitives in our implementations of maximal matching, minimum spanning forest, and triangle counting. Our implementation of pack runs in $O(|L|)$ work and $O(\lg n)$ depth.

The ***Intersection*** primitive takes as input two neighbor lists L_a and L_b and computes the size of the intersection of L_a and L_b ($|L_a| \leq |L_b|$). We implement an algorithm similar to the optimal parallel intersection algorithm for sorted lists. As the blocks are compressed, our implementation works on the first element of each block, which can be quickly decoded. We refer to these elements as block starts. If the number of blocks in both lists sum to less than a constant, we intersect them sequentially. Otherwise, we take the start v_s of the middle block in L_a, and binary search over the starts of L_b to find the first block whose start is less than or equal to v_s. Note that as the closest value less than or equal to v_s could be in the middle of the block, the subproblems we generate must consider elements in the two adjoining blocks of each list, which adds an extra constant factor of work in the base case. Our implementation of intersection runs in $O(|L_a| \lg(1 + |L_b|/|L_a|))$ work and $O(\lg n)$ depth.

Graph Dataset	Num. Vertices	Num. Edges	diam	ρ	$k_{\max}$
LiveJournal	4,847,571	68,993,773	16	~	~
LiveJournal-Sym	4,847,571	85,702,474	20	3480	372
com-Orkut	3,072,627	234,370,166	9	5,667	253
Twitter	41,652,231	1,468,365,182	65*	~	~
Twitter-Sym	41,652,231	2,405,026,092	23*	14,963	2488
3D-Torus	1,000,000,000	6,000,000,000	1500*	1	6
ClueWeb	978,408,098	42,574,107,469	821*	~	~
ClueWeb-Sym	978,408,098	74,744,358,622	132*	106,819	4244
Hyperlink2014	1,724,573,718	64,422,807,961	793*	~	~
Hyperlink2014-Sym	1,724,573,718	124,141,874,032	207*	58,711	4160
Hyperlink2012	3,563,602,789	128,736,914,167	5275*	~	~
Hyperlink2012-Sym	3,563,602,789	225,840,663,232	331*	130,728	10565

Table 5.2: Graph inputs, including vertices and edges. diam is the diameter of the graph. For undirected graphs, ρ and $k_{\max}$ are the number of peeling rounds, and the largest non-empty core (degeneracy). We mark diam values where we are unable to calculate the exact diameter with * and report the effective diameter observed during our experiments, which is a lower bound on the actual diameter.

5.8 Experiments

In this section, we describe our experimental results on a set of real-world graphs and also discuss related experimental work. Tables 5.4 and 5.5 show the running times for our implementations on our graph inputs. For compressed graphs, we use the compression schemes from Ligra+ [322], which we extended to ensure theoretical efficiency.

5.8.1 Experimental Setup and Graph Inputs

Experimental Setup. We ran all of our experiments on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4×2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use Cilk Plus to express parallelism and are compiled with the g++ compiler (version 5.4.1) with the `-O3` flag. By using Cilk's work-stealing scheduler we are able obtain an expected running time of $W/P + O(D)$ for an algorithm with W work and D depth on P processors [77]. For the parallel experiments, we use the command `numactl -i all` to balance the memory allocations across the sockets. All of the speedup numbers we report are the running times of our parallel implementation on 72-cores with hyper-threading over the running time of the implementation on a single thread.

Graph Data. To show how our algorithms perform on graphs at different scales, we selected a representative set of real-world graphs of varying sizes. Most of the graphs are Web graphs and social networks—low diameter graphs that are frequently used in practice.

Graph Dataset	Uncompressed	Compressed	Savings
ClueWeb	324GB	115GB	2.81x
ClueWeb-Sym	285GB	100GB	2.85x
Hyperlink2014	492GB	214GB	2.29x
Hyperlink2014-Sym	474GB	184GB	2.57x
Hyperlink2012	985GB	446GB	2.21x
Hyperlink2012-Sym	867GB	351GB	2.47x

Table 5.3: Compressed graph inputs, including memory required to store the graph in an uncompressed CSR format, memory required to store the graph in the parallel byte-compressed CSR format, and the savings obtained over the uncompressed format by the compressed format. The number of vertices and edges in these graphs are given in Table 5.2.

To test our algorithms on large diameter graphs, we also ran our implementations on 3-dimensional tori where each vertex is connected to its 2 neighbors in each dimension.

We list the graphs used in our experiments, along with their size, approximate diameter, peeling complexity [115], and degeneracy (for undirected graphs) in Table 5.2. ***LiveJournal*** is a directed graph of the social network obtained from a snapshot in 2008 [81]. ***com-Orkut*** is an undirected graph of the Orkut social network. ***Twitter*** is a directed graph of the Twitter network, where edges represent the follower relationship [208]. ***ClueWeb*** is a Web graph from the Lemur project at CMU [81]. ***Hyperlink2012*** and ***Hyperlink2014*** are directed hyperlink graphs obtained from the WebDataCommons dataset where nodes represent web pages [241]. ***3D-Torus*** is a 3-dimensional torus with 1B vertices and 6B edges. We mark symmetric (undirected) versions of the directed graphs with the suffix -Sym. We create weighted graphs for evaluating weighted BFS, Borůvka, widest path, and Bellman-Ford by selecting edge weights between $[1, \lg n)$ uniformly at random. We process LiveJournal, com-Orkut, Twitter, and 3D-Torus in the uncompressed format, and ClueWeb, Hyperlink2014, and Hyperlink2012 in the compressed format.

Table 5.3 lists the size in gigabytes of the compressed graph inputs used in this chapter both with and without compression, and reports the savings obtained by using compression. Note that the largest graph studied in this chapter, the directed Hyperlink2012 graph, barely fits in the main memory of our machine in the uncompressed format, but would leave hardly any memory to be used for an algorithm analyzing this graph. Using compression significantly reduces the memory required to represent each graph (between 2.21–2.85x, and 2.53x on average). We converted the graphs listed in Table 5.3 directly from the WebGraph format to the compressed format used in this chapter by modifying a sequential iterator method from the WebGraph framework [81].

Application	LiveJournal-Sym			com-Orkut			Twitter-Sym			3D-Torus		
	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)
Breadth-First Search (BFS)	0.59	0.018	32.7	0.41	0.012	34.1	5.45	0.137	39.7	301	5.53	54.4
Integral-Weight SSSP (weighted BFS)	1.45	0.107	13.5	2.03	0.095	21.3	33.4	0.995	33.5	437	18.1	24.1
General-Weight SSSP (Bellman-Ford)	3.39	0.086	39.4	3.98	0.168	23.6	48.7	1.56	31.2	6280	133	47.2
Single-Source Widest Path (Bellman-Ford)	3.48	0.090	38.6	4.39	0.098	44.7	42.4	0.749	56.6	580	9.7	59.7
Single-Source Betweenness Centrality (BC)	1.66	0.049	33.8	2.52	0.057	44.2	26.3	0.937	28.0	496	12.5	39.6
$O(k)$-Spanner	1.31	0.041	31.9	2.34	0.046	50.8	41.5	0.768	54.0	380	11.7	32.4
Low-Diameter Decomposition (LDD)	0.54	0.027	20.0	0.33	0.019	17.3	8.48	0.186	45.5	275	7.55	36.4
Connectivity	1.01	0.029	34.8	1.36	0.031	43.8	34.6	0.585	59.1	300	8.71	34.4
Spanning Forest	1.11	0.035	31.7	1.84	0.047	39.1	43.2	0.818	52.8	334	10.1	33.0
Biconnectivity	5.36	0.261	20.5	7.31	0.292	25.0	146	4.86	30.0	1610	59.6	27.0
Strongly Connected Components (SCC)*	1.61	0.116	13.8	~	~	~	13.3	0.495	26.8	~	~	~
Minimum Spanning Forest (MSF)	3.64	0.204	17.8	4.58	0.227	20.1	61.8	3.02	20.4	617	23.6	26.1
Maximal Independent Set (MIS)	1.18	0.034	34.7	2.23	0.052	42.8	34.4	0.759	45.3	236	4.44	53.1
Maximal Matching (MM)	2.42	0.095	25.4	4.65	0.183	25.4	46.7	1.42	32.8	403	11.4	35.3
Graph Coloring	4.69	0.392	11.9	9.05	0.789	11.4	148	6.91	21.4	350	11.3	30.9
Approximate Set Cover	4.65	0.613	7.58	4.51	0.786	5.73	66.4	3.31	20.0	1429	40.2	35.5
k-core	3.75	0.641	5.85	8.32	1.33	6.25	110	6.72	16.3	753	6.58	114.4
Approximate Densest Subgraph	2.89	0.052	55.5	4.71	0.081	58.1	76.0	1.14	66.6	95.4	1.59	60.0
Triangle Counting (TC)	13.5	0.342	39.4	78.1	1.19	65.6	1920	23.5	81.7	168	6.63	25.3
PageRank Iteration	0.861	0.012	71.7	1.28	0.018	71.1	24.16	0.453	53.3	107	2.25	47.5

Table 5.4: Running times (in seconds) of our algorithms over symmetric graph inputs on a 72-core machine (with hyper-threading) where (1) is the single-thread time, (72h) is the 72 core time using hyper-threading, and (SU) is the parallel speedup (single-thread time divided by 72-core time). We mark experiments that are not applicable for a graph with ~, and experiments that did not finish within 5 hours with —. *SCC was run on the directed versions of the input graphs.

5.8.2 SSSP Problems

Our BFS, weighted BFS, Bellman-Ford, and betweenness centrality implementations achieve between a 13–67x speedup across all inputs. We ran all of our shortest path experiments on the *symmetrized* versions of the graph. Our widest path implementation achieves between 38–72x speedup across all inputs, and our spanner implementation achieves between 31–65x speedup across all inputs. We ran our spanner code with $k = 4$. Our experiments show that our weighted BFS and Bellman-Ford implementations perform as well as or better than our prior implementations from Julienne [115]. Our running times for BFS and betweenness centrality are the same as the times of the implementations in Ligra [319]. We note that our running times for weighted BFS on the Hyperlink graphs are larger than the times reported in Julienne. This is because the shortest-path experiments in Julienne were run on directed version of the graph, where the average vertex can reach significantly fewer vertices than on the symmetrized version. We set a flag for our weighted BFS experiments on the ClueWeb and Hyperlink graphs that lets the algorithm switch to a dense EDGEMAP once the frontiers are sufficiently dense, which lets the algorithm run within half of the RAM on our machine. Before this change, our

Application	ClueWeb-Sym			Hyperlink2014-Sym			Hyperlink2012-Sym		
	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)
Breadth-First Search (BFS)	106	2.29	46.2	250	4.50	55.5	576	8.44	68.2
Integral-Weight SSSP (weighted BFS)	736	14.4	51.1	1390	22.3	62.3	3770	58.1	64.8
General-Weight SSSP (Bellman-Ford)	1050	16.2	64.8	1460	22.9	63.7	4010	59.4	67.5
Single-Source Widest Path (Bellman-Ford)	849	11.8	71.9	1211	16.8	72.0	3210	48.4	66.3
Single-Source Betweenness Centrality (BC)	569	27.7	20.5	866	16.3	53.1	2260	37.1	60.9
$O(k)$-Spanner	613	9.79	62.6	906	14.3	63.3	2390	36.3	65.8
Low-Diameter Decomposition (LDD)	176	3.62	48.6	322	6.84	47.0	980	16.6	59.0
Connectivity	381	6.01	63.3	710	11.2	63.3	1640	25.0	65.6
Spanning Forest	936	18.2	51.4	1319	22.4	58.8	2420	35.8	67.5
Biconnectivity	2250	48.7	46.2	3520	71.5	49.2	9860	165	59.7
Strongly Connected Components (SCC)*	1240	38.1	32.5	2140	51.5	41.5	8130	185	43.9
Minimum Spanning Forest (MSF)	2490	45.6	54.6	3580	71.9	49.7	9520	187	50.9
Maximal Independent Set (MIS)	551	8.44	65.2	1020	14.5	70.3	2190	32.2	68.0
Maximal Matching (MM)	1760	31.8	55.3	2980	48.1	61.9	7150	108	66.2
Graph Coloring	2050	49.8	41.1	3310	63.1	52.4	8920	158	56.4
Approximate Set Cover	1490	28.1	53.0	2040	37.6	54.2	5320	90.4	58.8
k-core	2370	62.9	37.6	3480	83.2	41.8	8515	184	46.0
Approximate Densest Subgraph	1380	19.6	70.4	1721	24.3	70.8	4420	61.4	71.9
Triangle Counting (TC)	13997	204	68.6	—	480	—	—	1168	—
PageRank Iteration	256.1	3.49	73.3	385	5.17	74.4	973	13.1	74.2

Table 5.5: Running times (in seconds) of our algorithms over symmetric graph inputs on a 72-core machine (with hyper-threading) where (1) is the single-thread time, (72h) is the 72 core time using hyper-threading, and (SU) is the parallel speedup (single-thread time divided by 72-core time). We mark experiments that are not applicable for a graph with ~, and experiments that did not finish within 5 hours with —. *SCC was run on the directed versions of the input graphs.

weighted BFS implementation would request a large amount of memory when processing the largest frontiers which then caused the graph to become partly evicted from the page cache. For widest path, the times we report are for the Bellman-Ford version of the algorithm, which we were surprised to find is consistently 1.1–1.3x faster than our algorithm based on bucketing. We observe that our spanner algorithm is only slightly more costly than computing connectivity on the same input.

In an earlier paper [115], we compared the running time of our weighted BFS implementation to two existing parallel shortest path implementations from the GAP benchmark suite [45] and Galois [228], as well as a fast sequential shortest path algorithm from the DIMACS shortest path challenge, showing that our implementation is between 1.07–1.1x slower than the Δ-stepping implementation from GAP, and 1.6–3.4x faster than the Galois implementation. Our old version of Bellman-Ford was between 1.2–3.9x slower than weighted BFS; we note that after changing it to use the EDGEMAPBLOCKED optimization, it is now competitive with weighted BFS and is between 1.2x faster and 1.7x slower on our

graphs with the exception of 3D-Torus, where it performs 7.3x slower than weighted BFS, as it performs $O(n^{4/3})$ work on this graph.

5.8.3 Connectivity Problems

Our low-diameter decomposition (LDD) implementation achieves between 17–59x speedup across all inputs. We fixed β to 0.2 in all of the codes that use LDD. The running time of LDD is comparable to the cost of a BFS that visits most of the vertices. We are not aware of any prior experimental work that reports the running times for an LDD implementation.

Our work-efficient implementation of connectivity and spanning forest achieve 25–57x speedup and 31–67x speedup across all inputs, respectively. We note that our implementation does not assume that vertex IDs in the graph are randomly permuted and always generates a random permutation, even on the first round, as adding vertices based on their original IDs can result in poor performance (for example on 3D-Torus). There are several existing implementations of fast parallel connectivity algorithms [273, 324, 321, 332], however, only the implementation from [321], which presents the connectivity algorithm that we implement in this chapter, is theoretically-efficient. The implementation from Shun et al. was compared to both the Multistep [332] and Patwary et al. [273] implementations, and shown to be competitive on a broad set of graphs. We compared our connectivity implementation to the work-efficient connectivity implementation from Shun et al. on our uncompressed graphs and observed that our code is between 1.2–2.1x faster in parallel. Our spanning forest implementation is slightly slower than connectivity due to having to maintain a mapping between the current edge set and the original edge set.

Despite our biconnectivity implementation having $O(\text{diam}(G))$ depth, our implementation achieves between a 20–59x speedup across all inputs, as the diameter of most of our graphs is extremely low. Our biconnectivity implementation is about 3–5 times slower than running connectivity on the graph, which seems reasonable as our current implementation performs two calls to connectivity, and one breadth-first search. There are a several existing implementations of biconnectivity. Cong and Bader [105] parallelize the Tarjan-Vishkin algorithm and demonstrated speedup over the Hopcroft-Tarjan (HT) algorithm. Edwards and Vishkin [131] also implement the Tarjan-Vishkin algorithm using the XMT platform, and show that their algorithm achieves good speedups. Slota and Madduri [331] present a BFS-based biconnectivity implementation which requires $O(mn)$ work in the worst-case, but behaves like a linear-work algorithm in practice. We ran the Slota and Madduri implementation on 36 hyper-threads allocated from the same socket, the configuration on which we observed the best performance for their code, and found that our implementation is between 1.4–2.1x faster than theirs. We used a DFS-ordered subgraph corresponding to the largest connected component to test their code, which produced the fastest times. Using the original order of the graph affects the running time

of their implementation, causing it to run between 2–3x slower as the amount of work performed by their algorithm depends on the order in which vertices are visited.

Our strongly connected components implementation achieves between a 13–43x speedup across all inputs. Our implementation takes a parameter β, which is the base of the exponential rate at which we grow the number of centers added. We set β between 1.1–2.0 for our experiments and note that using a larger value of β can improve the running time on smaller graphs by up to a factor of 2x. Our SCC implementation is between 1.6x faster to 4.8x slower than running connectivity on the graph. There are several existing SCC implementations that have been evaluated on real-world directed graphs [176, 332, 237]. The Hong et al. algorithm [176] is a modified version of the FWBW-Trim algorithm from McLendon et al. [237], but neither algorithm has any theoretical bounds on work or depth. Unfortunately [176] do not report running times, so we are unable to compare our performance with them. The Multistep algorithm [332] has a worst-case running time of $O(n^2)$, but the authors point-out that the algorithm behaves like a linear-time algorithm on real-world graphs. We ran our implementation on 16 cores configured similarly to their experiments and found that we are about 1.7x slower on LiveJournal, which easily fits in cache, and 1.2x faster on Twitter (scaled to account for a small difference in graph sizes). While the multistep algorithm is slightly faster on some graphs, our SCC implementation has the advantage of being theoretically-efficient and performs a predictable amount of work.

Our minimum spanning forest implementation achieves between 17–54x speedup over the implementation running on a single thread across all of our inputs. Obtaining practical parallel algorithms for MSF has been a longstanding goal in the field, and several existing implementations exist [34, 262, 106, 324, 388]. We compared our implementation with the union-find based MSF implementation from PBBS [324] and the implementation of Borůvka from [388], which is one of the fastest implementations we are aware of. Our MSF implementation is between 2.6–5.9x faster than the MSF implementation from PBBS. Compared to the edgelist based implementation of Borůvka from [388] our implementation is between 1.2–2.9x faster.

5.8.4 Covering Problems

Our MIS and maximal matching implementations achieve between 31–70x and 25–70x speedup across all inputs. The implementations by Blelloch et al. [64] are the fastest existing implementations of MIS and maximal matching that we are aware of, and are the basis for our maximal matching implementation. They report that their implementations are 3–8x faster than Luby's algorithm on 32 threads, and outperform a sequential greedy MIS implementation on more than 2 processors. We compared our rootset-based MIS implementation to the prefix-based implementation, and found that the rootset-based approach is between 1.1–3.5x faster. Our maximal matching implementation is between 3–

4.2x faster than the implementation from [64]. Our implementation of maximal matching can avoid a significant amount of work, as each of the filter steps can extract and permute just the $3n/2$ highest priority edges, whereas the edgelist-based version in PBBS must permute all edges. Our coloring implementation achieves between 11–56x speedup across all inputs. We note that our implementation appears to be between 1.2–1.6x slower than the asynchronous implementation of JP in [169], due to synchronizing on many rounds which contain few vertices.

Our approximate set cover implementation achieves between 5–57x speedup across all inputs. Our implementation is based on the implementation presented in Julienne [115]; the one major modification was to regenerate random priorities for sets that are active on the current round. We compared the running time of our implementation with the parallel implementation from [68] which is available in the PBBS library. We ran both implementations with $\epsilon = 0.01$. Our implementation is between 1.2x slower to 1.5x faster than the PBBS implementation on our graphs, with the exception of 3D-Torus. On 3D-Torus, the implementation from [68] runs 56x slower than our implementation as it does not regenerate priorities for active sets on each round causing worst-case behavior. Our performance is also slow on this graph, as nearly all of the vertices stay active (in the highest bucket) during each round, and using $\epsilon = 0.01$ causes a large number of rounds to be performed.

5.8.5 Substructure Problems

Our k-core implementation achieves between 5–46x speedup across all inputs, and 114x speedup on the 3D-Torus graph as there is only one round of peeling in which all vertices are removed. There are several recent papers that implement parallel algorithms for k-core [110, 115, 299, 192]. Both the ParK algorithm [110] and Kabir and Madduri algorithm [192] implement the peeling algorithm in $O(k_{\max}n + m)$ work, which is not work-efficient. Our implementation is between 3.8–4.6x faster than ParK on a similar machine configuration. Kabir and Madduri show that their implementation achieves an average speedup of 2.8x over ParK. Our implementation is between 1.3–1.6x faster than theirs on a similar machine configuration.

Our approximate densest subgraph implementation achieves between 44–77x speedup across all inputs. We ran our implementation with $\epsilon = 0.001$, which in our experiments produced subgraphs with density roughly equal to those produced by the 2-approximation algorithm based on degeneracy ordering, or setting ϵ to 0. To the best of our knowledge, there are no prior existing shared-memory parallel algorithms for this problem.

Our triangle counting (TC) implementation achieves between 39–81x speedup across all inputs. Unfortunately, we are unable to report speedup numbers for TC on our larger graphs as the single-threaded times took too long due to the algorithm performing $O(m^{3/2})$ work. There are a number of experimental papers that consider multicore triangle counting [308,

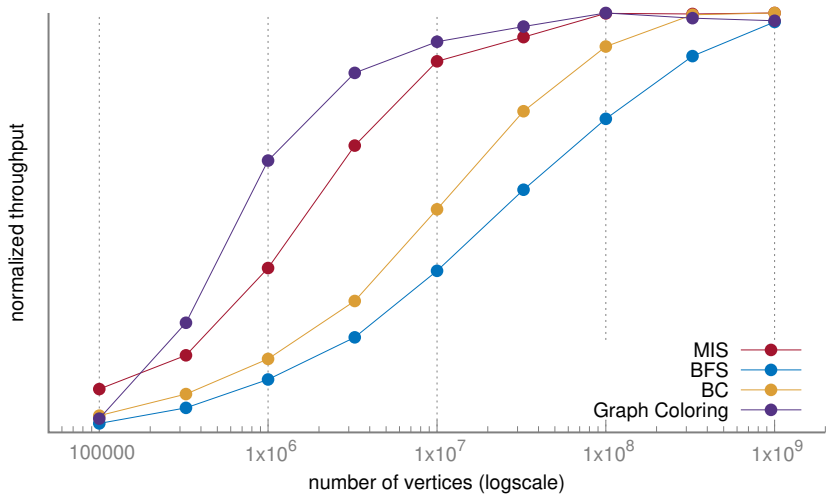


Figure 5.1: Log-linear plot of normalized throughput vs. vertices for MIS, BFS, BC, and coloring on the 3D-Torus graph family.

157, 202, 323, 3, 223]. We implement the algorithm from [323], and adapted it to work on compressed graphs. We note that in our experiments we intersect directed adjacency lists sequentially, as there was sufficient parallelism in the outer parallel-loop. There was no significant difference in running times between our implementation and the implementation from [323]. We ran our implementation on 48 threads on the Twitter graph to compare with the times reported by EmptyHeaded [3] and found that our times are about the same.

5.8.6 Eigenvector Problems

Our PageRank (PR) implementation achieves between 47–74x speedup across all inputs. We note that the running times we report are for a single iteration of PageRank. Our implementation is based on the implementation from Ligra [319], and uses a damping factor $\gamma = 0.85$. We note that the modification made to carry out dense iterations using a reduction over the in-neighbors of a vertex was important to decrease contention and improve parallelism, and provided between 2–3x speedup over the Ligra implementation in practice. Many graph processing systems implement PageRank. The optimizing compiler used by GraphIt generates a highly-optimized implementation that is currently the fastest shared-memory implementation known to us [383]. We note that our implementation is about 1.8x slower than the implementation in GraphIt for LiveJournal and Twitter when run on the same number of threads as in their experiments.

5.8.7 Performance on 3D-Torus

We ran experiments on a family of 3D-Torus graphs with different sizes to study how our diameter-bounded algorithms scale relative to algorithms with polylogarithmic depth. We were surprised to see that the running time of some of our polylogarithmic depth algorithms on this graph, like LDD and connectivity, are 17–40x more expensive than their running time on Twitter and Twitter-Sym, despite 3D-Torus only having 4x and 2.4x more edges than Twitter and Twitter-Sym. One reason for our slightly worse scaling on this graph (and the higher cost of algorithms relative to graphs with a similar number of edges) is the very low average-degree of this graph ($m/n = 6$) compared with the Twitter graph ($m/n = 57.8$). Many of the algorithms we study in this chapter process all edges incident to a vertex whenever a vertex is considered (e.g., when a vertex is part of a frontier in the LDD computation). Furthermore, each vertex is only processed a constant number of times. Thus, each time such an algorithm processes a vertex in the 3D-Torus graph, it only uses 24 bytes out of each 64-byte cache line (assuming each edge is stored in 4 bytes), but it will utilize the entire cache line in the Twitter graph, on average. Another possible reason is that we store the 3D-Torus graph ordered by dimension, instead of using a local ordering. However, we did not study reordering this graph, since it was not the main focus of this work.

In Figure 5.1 we show the normalized throughput of MIS, BFS, BC, and graph coloring for 3-dimensional tori of different sizes, where throughput is measured as the number of edges processed per second. The throughput for each application becomes saturated before our largest-scale graph for all applications except for BFS, which is saturated on a graph with 2 billion vertices. The throughput curves show that the theoretical bounds are useful in predicting how the half-lengths[6] are distributed. The half-lengths are ordered as follows: coloring, MIS, BFS, and BC. This is the same order as sorting these algorithms by their depth with respect to this graph.

5.8.8 Locality

While our algorithms are efficient on the TRAM, we do not analyze their cache complexity, and in general they may not be efficient in a model that takes caches into account. Despite this, we observed that our algorithms have good cache performance on the graphs we tested on. In this section we give some explanation for this fact by showing that our primitives make good use of the caches. Our algorithms are also aided by the fact that these graph datasets often come in highly local orders (e.g., see the *Natural* order in [120]). Table 5.6 shows metrics for our experiments measured using Open Performance Counter Monitor (PCM).

[6]The graph size when the system achieves half of its peak-performance.

Algorithm	Cycles Stalled	LLC Hit Rate	LLC Misses	BW	Time
k-core (histogram)	9	0.223	49	96	62.9
k-core (FETCHANDADD)	67	0.155	42	24	221
weighted BFS (blocked)	3.7	0.070	19	130	14.4
weighted BFS (unblocked)	5.6	0.047	29	152	25.2

Table 5.6: Cycles stalled while the memory subsystem has an outstanding load (trillions), LLC hit rate and misses (billions), bandwidth in GB/s (bytes read and written from memory, divided by running time), and running time in seconds. All experiments are run on the ClueWeb graph using 72 cores with hyper-threading.

We run our locality experiments on the ClueWeb graph. We found that using a work-efficient histogram is 3.5x faster than using FETCHANDADD in our k-core implementation, which suffers from high contention on this graph. Using a histogram reduces the number of cycles stalled due to memory by more than 7x. We also ran our wBFS implementation with and without the EDGEMAPBLOCKED optimization, which reduces the number of cache-lines read from and written to when performing a sparse EDGEMAP. The blocked implementation reads and writes 2.1x fewer bytes than the unoptimized version, which translates to a 1.7x faster runtime. We disabled the dense optimization for this experiment to directly compare the two implementations of a sparse EDGEMAP.

5.8.9 Processing Massive Web Graphs

In Tables 5.4 and 5.5, we show the running times of our implementations on the ClueWeb, Hyperlink2014, and Hyperlink2012 graphs. To put our performance in context, we compare our 72-core running times to running times reported by existing work. Table 5.7 summarizes state-of-the-art existing results in the literature. Most results process the *directed* versions of these graphs, which have about half as many edges as the symmetrized version. Unless otherwise mentioned, all results from the literature use the directed versions of these graphs. To make the comparison easier we show our running times for BFS, SSSP (weighted BFS), BC and SCC on the directed graphs, and running times for Connectivity, k-core and TC on the symmetrized graphs in Table 5.7.

FlashGraph [385] reports disk-based running times for the Hyperlink2012 graph on a 4-socket, 32-core machine with 512GB of memory and 15 SSDs. On 64 hyper-threads, they solve BFS in 208s, BC in 595s, connected components in 461s, and triangle counting in 7818s. Our BFS and BC implementations are 12x faster and 16x faster, and our triangle counting and connectivity implementations are 5.3x faster and 18x faster than their implementations, respectively. Mosaic [225] report in-memory running times on the Hyperlink2014 graph; we note that the system is optimized for external memory execution. They solve BFS in 6.5s, connected components in 700s, and SSSP (Bellman-Ford) in 8.6s on a machine with 24 hyper-threads and 4 Xeon-Phis (244 cores with 4 threads each) for a total of 1000 hyper-

Paper	Problem	Graph	Memory	Hyper-threads	Nodes	Time
Mosaic [225]	BFS*	2014	0.768	1000	1	6.55
	Connectivity*	2014	0.768	1000	1	708
	SSSP*	2014	0.768	1000	1	8.6
FlashGraph [385]	BFS*	2012	.512	64	1	208
	BC*	2012	.512	64	1	595
	Connectivity*	2012	.512	64	1	461
	TC*	2012	.512	64	1	7818
GraFBoost [191]	BFS*	2012	0.064	32	1	900
	BC*	2012	0.064	32	1	800
Slota et al. [330]	Largest-CC*	2012	16.3	8192	256	63
	Largest-SCC*	2012	16.3	8192	256	108
	Approx k-core*	2012	16.3	8192	256	363
Stergiou et al. [336]	Connectivity	2012	128	24000	1000	341
Gluon [111]	BFS	2012	24	69632	256	380
	Connectivity	2012	24	69632	256	75.3
	PageRank	2012	24	69632	256	158.2
	SSSP	2012	24	69632	256	574.9
This paper	BFS*	2014	1	144	1	5.71
	SSSP*	2014	1	144	1	9.08
	Connectivity	2014	1	144	1	11.2
	BFS*	2012	1	144	1	16.7
	BC*	2012	1	144	1	35.2
	Connectivity	2012	1	144	1	25.0
	SCC*	2012	1	144	1	185
	SSSP	2012	1	144	1	58.1
	k-core	2012	1	144	1	184
	PageRank	2012	1	144	1	462
	TC	2012	1	144	1	1168

Table 5.7: System configurations (memory in terabytes, hyper-threads, and nodes) and running times (seconds) of existing results on the Hyperlink graphs. The last section shows our running times. *These problems are run on directed versions of the graph.

threads, 768GB of RAM, and 6 NVMes. Our BFS and connectivity implementations are 1.1x and 62x faster respectively, and our SSSP implementation is 1.05x slower. Both FlashGraph and Mosaic compute weakly connected components, which is equivalent to connectivity. GraFBoost [191] report disk-based running times for BFS and BC on the Hyperlink2012 graph on a 32-core machine. They solve BFS in 900s and BC in 800s. Our BFS and BC implementations are 53x and 22x faster than their implementations, respectively.

Slota et al. [330] report running times for the Hyperlink2012 graph on 256 nodes on the Blue Waters supercomputer. Each node contains two 16-core processors with one thread

each, for a total of 8192 hyper-threads. They report they can find the *largest* connected component and SCC from the graph in 63s and 108s respectively. Our implementations find *all* connected components 2.5x faster than their largest connected component implementation, and find *all* strongly connected components 1.6x slower than their largest-SCC implementation. Their largest-SCC implementation computes two BFSs from a randomly chosen vertex—one on the in-edges and the other on the out-edges—and intersects the reachable sets. We perform the same operation as one of the first steps of our SCC algorithm and note that it requires about 30 seconds on our machine. They solve approximate k-cores in 363s, where the approximate k-core of a vertex is the coreness of the vertex rounded up to the nearest powers of 2. Our implementation computes the *exact* coreness of each vertex in 184s, which is 1.9x faster than the approximate implementation while using 113x fewer cores.

Recently, Dathathri et al. [111] have reported running times for the Hyperlink2012 graph using Gluon, a distributed graph processing system based on Galois. They process this graph on a 256 node system, where each node is equipped with 68 4-way hyper-threaded cores, and the hosts are connected by an Intel Omni-Path network with 100Gbps peak bandwidth. They report times for BFS, connectivity, PageRank, and SSSP. Other than their connectivity implementation, which uses pointer-jumping, their implementations are based on data-driven asynchronous label-propagation. We are not aware of any theoretical bounds on the work and depth of these implementations. Compared to their reported times, our implementation of BFS is 22.7x faster, our implementation of connectivity is 3x faster, and our implementation of SSSP is 9.8x faster. Our PageRank implementation is 2.9x slower (we ran it with ϵ, the variable that controls the convergence rate of PageRank, set to $1e-6$). However, we note that the PageRank numbers they report are not for true PageRank, but PageRank-Delta, and are thus in some sense incomparable.

Stergiou et al. [336] describe a connectivity algorithm that runs in $O(\lg n)$ rounds in the BSP model and report running times for the symmetrized Hyperlink2012 graph. They implement their algorithm using a proprietary in-memory/secondary-storage graph processing system used at Yahoo!, and run experiments on a 1000 node cluster. Each node contains two 6-core processors that are 2-way hyper-threaded and 128GB of RAM, for a total of 24000 hyper-threads and 128TB of RAM. Their fastest running time on the Hyperlink2012 graph is 341s on their 1000 node system. Our implementation solves connectivity on this graph in 25s–13.6x faster on a system with 128x less memory and 166x fewer cores. They also report running times for solving connectivity on a private Yahoo! webgraph with 272 billion vertices and 5.9 trillion edges, over 26 times the size of our largest graph. While such a graph seems to currently be out of reach of our machine, we are hopeful that techniques from theoretically-efficient parallel algorithms can help solve problems on graphs at this scale and beyond.

5.9 Related Work

Parallel Graph Algorithms. Parallel graph algorithms have received significant attention since the start of parallel computing, and many elegant algorithms with good theoretical bounds have been developed over the decades (e.g., [316, 196, 224, 17, 346, 247, 284, 188, 104, 277, 246, 140, 57, 242]). A major goal in parallel graph algorithm design is to find *work-efficient* algorithms with polylogarithmic depth. While many suspect that work-efficient algorithms may not exist for all parallelizable graph problems, as inefficiency may be inevitable for problems that depend on transitive closure, many problems that are of practical importance do admit work-efficient algorithms [195]. For these problems, which include connectivity, biconnectivity, minimum spanning forest, maximal independent set, maximal matching, and triangle counting, giving theoretically-efficient implementations that are simple and practical is important, as the amount of parallelism available on modern systems is still modest enough that reducing the amount of work done is critical for achieving good performance. Aside from intellectual curiosity, investigating whether theoretically-efficient graph algorithms also perform well in practice is important, as theoretically-efficient algorithms are less vulnerable to adversarial inputs than ad-hoc algorithms that happen to work well in practice.

Unfortunately, some problems that are not known to admit work-efficient parallel algorithms due to the transitive-closure bottleneck [195], such as strongly connected components (SCC) and single-source shortest paths (SSSP) are still important in practice. One method for circumventing the bottleneck is to give work-efficient algorithms for these problems that run in depth proportional to the diameter of the graph—as real-world graphs have low diameter, and theoretical models of real-world graphs predict a logarithmic diameter, these algorithms offer theoretical guarantees in practice [303, 75]. Other problems, like k-core are P-complete [23], which rules out polylogarithmic-depth algorithms for them unless P = NC [158]. However, even k-core admits an algorithm with strong theoretical guarantees that is efficient in practice [115].

Parallel Graph Processing Frameworks. Motivated by the need to process very large graphs, there have been many graph processing frameworks developed in the literature (e.g., [229, 154, 223, 259, 319] among many others). We refer the reader to [235, 377] for surveys of existing frameworks. Several recent graph processing systems evaluate the scalability of their implementations by solving problems on massive graphs [333, 385, 225, 115, 191, 336]. All of these systems report running times either on the Hyperlink 2012 graph or Hyperlink 2014 graphs, two web crawls released by the WebDataCommons that are the largest and second largest publicly-available graphs respectively. We describe these recent systems and give a detailed comparison of how our implementations perform compare to their codes in Section 5.8.

Benchmarking Parallel Graph Algorithms. There are a surprising number of existing

benchmarks of parallel graph algorithms. SSCA [35], an early graph processing benchmark, specifies four graph kernels, which include generating graphs in adjacency list format, subgraph extraction, and graph clustering. The Problem Based Benchmark Suite (PBBS) [324] is a general benchmark of parallel algorithms that includes 6 problems on graphs: BFS, spanning forest, minimum spanning forest, maximal independent set, maximal matching, and graph separators. The PBBS benchmarks are problem-based in that they are defined only in terms of the input and output without any specification of the algorithm used to solve the problem. We follow the style of PBBS in this chapter of defining the input and output requirements for each problem. The Graph Algorithm Platform (GAP) Benchmark Suite [45] specifies 6 kernels: BFS, SSSP, PageRank, connectivity, betweenness centrality, and triangle counting.

Several recent benchmarks characterize the architectural properties of parallel graph algorithms. GraphBIG [255] describes 12 applications, including several problems that we consider, like k-core and graph coloring (using the Jones-Plassmann algorithm), but also problems like depth-first search, which are difficult to parallelize, as well as dynamic graph operations. CRONO [13] implements 10 graph algorithms, including all-pairs shortest paths, exact betweenness centrality, traveling salesman, and depth-first search. LDBC [181] is an industry-driven benchmark that selects 6 algorithms that are considered representative of graph processing including BFS, and several algorithms based on label propagation.

Unfortunately, all of the existing graph algorithm benchmarks we are aware of restrict their evaluation to small graphs, often on the order of tens or hundreds of millions of edges, with the largest graphs in the benchmarks having about two billion edges. As real-world graphs are frequently several orders of magnitude larger than this, evaluation on such small graphs makes it hard to judge whether the algorithms or results from a benchmark scale to terabyte-scale graphs.

5.10 Discussion

In this chapter, we showed that we can process the largest publicly-available real-world graph on a single shared-memory server with 1TB of memory using theoretically-efficient parallel algorithms. We outperform existing implementations on the largest real-world graphs, and use many fewer resources than the distributed-memory solutions. On a per-core basis, our numbers are significantly better. Our results provide evidence that theoretically-efficient shared-memory graph algorithms can be efficient and scalable in practice.

There are many directions for future work stemming from this work. One is to continue to extend GBBS with other graph problems that were not considered in this chapter. For example, in our recent work we have extended GBBS with work-efficient clique-counting

algorithms and work-efficient algorithms for low out-degree orientation [313]. It would be interesting to study and implement parallel algorithms for other classic graph problems, such as planarity testing and embedding, planar separator, higher connectivity, amongst many others.

Another direction is to extend GBBS to fundamental application domains of graph algorithms, such as graph clustering. Although clustering is quite different from the problems studied in this chapter since there is usually no single "correct" way to cluster a graph or point set, we believe that our approach will be useful for building theoretically-efficient and scalable single-machine clustering algorithms, including density-based clustering [137], single-linkage methods such as affinity clustering [42], and agglomerative graph clustering [230].

6 Semi-Asymmetric Graph Algorithms

6.1 Introduction

Over the past decade, there has been a steady increase in the main-memory sizes of commodity multicore machines, which has led to the development of fast single-machine shared-memory graph algorithms for processing massive graphs with hundreds of billions of edges on a single machine (see for example [319, 260, 322], and the work covered in this thesis in Chapter 4 and Chapter 5). As shown in Chapters 4 and 5, single-machine analytics by-and-large outperform their distributed memory counterparts, running up to *orders of magnitude faster* using much fewer resources (see also [238, 319, 322]). These analytics have become increasingly relevant due to a longterm trend of increasing memory sizes, which continues today in the form of new non-volatile memory technologies that are now emerging on the market (e.g., Intel's *Optane DC Persistent Memory*). These devices are significantly cheaper on a per-gigabyte basis, provide an order of magnitude greater memory capacity per DIMM than traditional DRAM, and offer byte-addressability and low idle power, thereby providing a realistic and cost-efficient way to equip a commodity multicore machine with multiple terabytes of non-volatile RAM (NVRAM).

Due to these advantages, NVRAMs are likely to be a key component of many future memory hierarchies, likely in conjunction with a smaller amount of traditional DRAM. However, a challenge of these technologies is to overcome an *asymmetry* between reads and writes—write operations are more expensive than reads in terms of energy and throughput. This property requires rethinking algorithm design and implementations to minimize the number of writes to NVRAM [53, 76, 69, 94, 358]. As an example of the memory technology and its tradeoffs, in this chapter we use a 48-core machine that has 8x as much NVRAM as DRAM (we are aware of machines with 16x as much NVRAM as DRAM [152]), where the combined read throughput for all cores from the NVRAM is about 3x slower than reads from the DRAM, and writes on the NVRAM are a further factor of about 4x slower [186, 293] (a factor of 12 total). Under this asymmetric setting, algorithms performing a large number of writes could see a significant performance penalty if care is not taken to avoid or eliminate writes.

An important property of most graphs used in practice is that they are sparse, but still tend to have many more edges than vertices, often from one to two orders of magnitude more. This is true for almost all social network graphs [215], but also for many graphs that are derived from various simulations [113]. In Figure 6.1 we show that over 90% of the large graphs (more than 1 million vertices) from the SNAP [215] and LAW [81] datasets have at least 10 times as many edges as vertices. Given that very large graphs

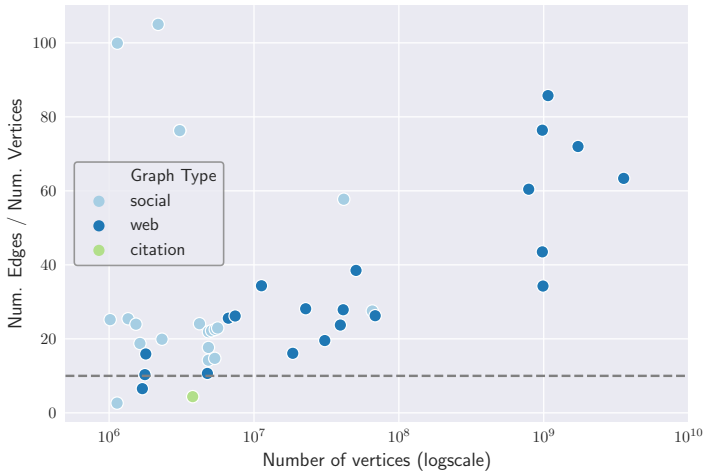


Figure 6.1: Number of vertices (logscale) vs. average degree (m/n) on 42 real-world graphs with $n > 10^6$ from the SNAP [215] and LAW [81] datasets. Over 90% of the graphs have average degree larger than 10 (corresponding to the gray dashed line).

today can have over 100 billion edges (requiring around a terabyte of storage), but only a few billion vertices, a popular and reasonable assumption both in theory and in practice is that vertices, but not edges, fit in DRAM [1, 138, 204, 236, 239, 254, 274, 339, 385, 386].

With these characteristics of NVRAM and real-world graphs in mind, we propose a *semi-asymmetric* approach to parallel graph analytics, in which (i) the full graph is stored in NVRAM and is accessed in *read-only mode* and (ii) the amount of DRAM is proportional to the number of vertices. Although completely avoiding writes to the NVRAM may seem overly restrictive, the approach has the following benefits: (i) algorithms avoid the high cost of NVRAM writes, (ii) the algorithms do not contribute to NVRAM wear-out or wear-leveling overheads, and (iii) algorithm design is independent of the actual cost of NVRAM writes, which has been shown to vary based on access pattern and number of cores [186, 293] and will likely change with innovations in NVRAM technology and controllers. Moreover, it enables an important NUMA optimization in which a copy of the graph is stored on each socket (Section 6.8), for fast read-only access without any cross-socket coordination. Finally, with no graph mutations, there is no need to re-compress the graph on-the-fly when processing compressed graphs [115, 117].

The key question, then, is the following:

Is the (restrictive) semi-asymmetric approach effective for designing fast graph algorithms?

In this chapter, we provide strong theoretical and experimental evidence of the approach's effectiveness.

Our Contributions. Our main contribution in this chapter is *Sage*, a parallel semi-asymmetric graph engine with which we implement provably-efficient (and often work-

optimal) algorithms for over a dozen fundamental graph problems (see Table 6.1). The key innovations are in ensuring that the updated state is associated with vertices and not edges, which is particularly challenging (i) for certain edge-based parallel graph traversals and (ii) for algorithms that "delete" edges as they go along in order to avoid revisiting them once they are no longer needed. We provide general techniques (Sections 6.4 and 6.5) to solve these two problems. For the latter, used by four of our algorithms, we require relaxing the prescribed amount of DRAM to be on the order of one bit per edge. Details of our algorithms are given in Section 6.7. Our codes extend the current state-of-the-art DRAM-only codes from GBBS (described in Chapter 5), and can be found at https://github.com/ParAlg/gbbs/tree/master/sage.

From a theoretical perspective, we propose a model for analyzing algorithms in the semi-asymmetric setting (Section 6.2). The model, called the Parallel Semi-Asymmetric Model (PSAM), consists of a shared asymmetric large-memory with unbounded size that can hold the entire graph, and a shared symmetric small-memory with $O(n)$ words of memory, where n is the number of vertices in the graph. In a relaxed version of the model, we allow small-memory size of $O(n + m/\lg n)$ words, where m is the number of edges in the graph. Although we do not use writes to the large-memory in our algorithms, the PSAM model permits writes to the large-memory, which are $\omega > 1$ times more costly than reads. We prove strong theoretical bounds in terms of PSAM work and depth for all of our parallel algorithms in Sage, as shown in Table 6.1. Most of the algorithms are work-efficient (performing asymptotically the same work as the best sequential algorithm for the problem) and have polylogarithmic depth (parallel time). These provable guarantees ensure that our algorithms perform reasonably well across graphs with different characteristics, machines with different core counts, and NVRAMs with different read-write asymmetries.

We summarize the main contributions of this chapter below.

(**1**) A semi-asymmetric approach to parallel graph analytics that avoids writing to the NVRAM and uses DRAM proportional to the number of vertices.

(**2**) Sage: a parallel semi-asymmetric graph engine with implementations of 18 fundamental graph problems, and general techniques for semi-asymmetric parallel graph algorithms. We have made all of our codes publicly-available.[1]

(**3**) A new theoretical model called the Parallel Semi-Asymmetric Model, and techniques for designing efficient, and often work-optimal parallel graph algorithms in the model.

(**4**) A comprehensive experimental evaluation of Sage on an NVRAM system showing that Sage significantly outperforms prior work and nearly matches state-of-the-art DRAM-only performance.

Preliminaries. We refer to Chapter 2 for general preliminaries on notation and parallel primitives used in this chapter. The work in this chapter builds on Ligra and Ligra+ (covered

[1]Our code can be found at https://github.com/ParAlg/gbbs/tree/master/sage, and an accompanying website at https://paralg.github.io/gbbs/sage.

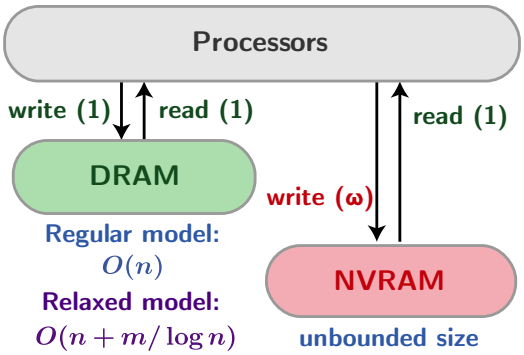


Figure 6.2: The Parallel Semi-Asymmetric Model. Algorithms in the model perform accesses to a symmetric small-memory (DRAM) and an asymmetric large-memory (NVRAM) at a word-granularity. Reads from both memories are charged unit-cost, whereas writes to the asymmetric memory are charged ω. In the regular model, algorithms have access to $O(n)$ words of symmetric memory, and in a relaxed variant have access to $O(n + m/\lg n)$ words of symmetric memory. Compared to existing two-level models, the main advantages of the PSAM are that it explicitly models *NVRAM read-write asymmetry* and it provides *sufficient symmetric memory* to design provably-efficient and practical parallel graph algorithms.

in Chapter 2), Julienne (Chapter 4) and the GBBS algorithms described in Chapter 5. Our algorithms work in a parallel model called the Parallel Semi-Asymmetric Model that we introduce in Section 6.2. Section 6.2 also specifies how the work and depth of an algorithm are calculated in this model.

6.2 Parallel Semi-Asymmetric Model

6.2.1 Model Definition

The ***Parallel Semi-Asymmetric Model (PSAM)*** consists of an *asymmetric* large-memory (NVRAM) with unbounded size, and a *symmetric* small-memory (DRAM) with $O(n)$ words of memory. In a relaxed version of the model, we allow small-memory size of $O(n+m/\lg n)$ words. The relaxed version is intended to model a system where the ratio of NVRAM to DRAM is close to the average degree of real-world graphs (see Figure 6.1 and Table 2.1).

The PSAM has a set of threads that share both the large-memory and small-memory. The underlying mechanisms for parallelism are identical to the T-RAM or binary forking model, which is discussed in detail in [62, 72, 117]. In the model, each thread acts like a sequential RAM that also has a `fork` instruction. When a thread performs a `fork`, two newly created child threads run starting at the next instruction, and the original thread is suspended until all the children terminate. A computation starts with a single *root* thread and finishes when that root thread finishes.

Algorithm Cost. We analyze algorithms on the PSAM using the ***work-depth measure*** [188]. The work-depth measure is a fundamental tool in analyzing parallel algorithms, e.g., see [63, 117, 161, 321, 342, 341, 366] for a sample of recent practical uses of this model. Like other multi-level models (e.g., the ANP model [53]), we assume unit cost reads and writes to the small-memory, and reads from the large-memory, all in the unit of a word. A write to the large-memory has a cost of $\omega > 1$, which is the cost of a write relative to a read on NVRAMs. The overall ***work*** W of an algorithm is the sum of the costs for all memory accesses by all threads. The ***depth*** D is the cost of the highest cost sequence of dependent instructions in the computation. A work-stealing scheduler can execute a computation in $W/p + O(D)$ time with high probability on p processors [53, 78]. Figure 6.2 illustrates the PSAM model.

6.2.2 Discussion

It is helpful to first clarify why we chose to keep the modeling parameters simple, focusing on *NVRAM read-write asymmetry*, when several other parameters are also available (as discussed below). Our goal was to design a theoretical model that helps guide algorithm design by capturing the most salient features of the new hardware.

Modeling Read and Write Costs. Although NVRAM reads are about 3x more costly than accesses to DRAM [293], we charge both unit cost in the PSAM. When this cost gap needs to be studied (especially for showing lower bounds), we can use an approach similar to the asymmetric RAM (ARAM) model [69], and define the *I/O cost* Q of an algorithm without charging for instructions or DRAM accesses. All algorithms in this chapter have asymptotically as many instructions as NVRAM reads, and therefore have the same I/O cost Q as work W up to constant factors.

Writes to Large-memory. Although in the approach used in this chapter we do not perform writes to the large-memory, the PSAM is designed to allow for analyzing alternate approaches that do perform writes to large-memory. Furthermore, permitting writes to the large-memory enables us to consider the cost of algorithms from previous work such as GBBS [117] and observe that many prior algorithms with W work in the standard work-depth model are $\Theta(\omega W)$ work in the PSAM. We emphasize that the objective of this work is to evaluate whether the restrictive approach used in our algorithms—i.e., completely avoiding writes to the large-memory, thereby gaining the benefits discussed in Section 6.1—is effective compared to existing approaches for programming NVRAM graph algorithms. We note that algorithms designed with a small number of large-memory writes could possibly be quite efficient in practice.

Applicability. In this chapter, we provide evidence that the PSAM is broadly applicable for many (18) fundamental graph problems. We believe that many other problems will also fit in the PSAM. For example, counting and enumerating k-cliques, which were very

recently studied in the in-memory setting [313], can be adapted to the PSAM using the filtering technique proposed in this chapter. Other fundamental subgraph problems, such as subgraph matching [165, 343] and frequent subgraph mining [132, 364] could be solved in the PSAM using a similar approach, but mining many large subgraphs may require performing some writes to the NVRAM (as discussed below). Other problems, such as local search problems including CoSimRank [294], personalized PageRank, and other local clustering problems [325], naturally fit in the regular PSAM model.

We note that certain problems seem to require performing writes in the PSAM. For example, in the k-truss problem, the output requires emitting the trussness value for each edge, and thus storing the output requires $\Theta(m)$ words of memory, which requires $\Theta(\omega m)$ cost due to writes. Generalizations of k-truss, such as the (r, s)-nucleii problem appear to have the same requirement for $r \geq 2$ [300].

6.2.3 Relationship to Other Models

Asymmetric Models. The model considered in this chapter is related to the ARAM model [69] and the asymmetric nested-parallel (ANP) model [53]. Compared to these more general models, the PSAM is specially designed for graphs, with its small-memory being either $O(n)$ or $O(n + m/\lg n)$ words (for n vertices and m edges).

External and Semi-External Memory Models. The External Memory model (also known as the I/O or disk-access model) [12] is a classic two-level memory model containing a bounded internal memory of size M and an unbounded external memory. I/Os to the external memory are done in blocks of size B. The Semi-External Memory model [1] is a relaxation of the External Memory model where there is a small-memory that can hold the vertices but not the edges.

There are three major differences between the PSAM and the External Memory and Semi-External Memory models. First, unlike the PSAM, neither the External Memory nor the Semi-External Memory model *account for accessing the small-memory (DRAM)*, because the objective of these models is to focus on the cost of expensive I/Os to the external memory. We believe that for existing systems with NVRAMs, the cost of DRAM accesses is not negligible. Second, both the External Memory and Semi-External Memory have a parameter B to model data movement in large chunks. NVRAMs support random access, so for the ease of design and analysis we omit this parameter B. Third, the PSAM explicitly models the asymmetry of writing to the large memory, whereas the External Memory and Semi-External Memory models treat both reads and writes to the external memory indistinguishably (both cost B). The asymmetry of these devices is significant for current devices (writes to NVRAM are 4x slower than reads from NVRAM, and 12x slower than reads from DRAM [186, 293]), and could be even larger in future generations of energy-efficient NVRAMs. Explicitly modeling asymmetry is an important aspect of our approach in the PSAM.

Semi-Streaming Model. In the semi-streaming model [138, 254], there is a memory size of $O(n \cdot \text{polylog}(n))$ bits and algorithms can only read the graph in a sequential streaming order (with possibly multiple passes). In contrast, the PSAM allows random access to the input graph because NVRAMs intrinsically support random access. Furthermore the PSAM allows expensive writes to the large-memory, which is read-only in the semi-streaming model.

Table 6.1: Work and depth bounds of Sage algorithms in the PSAM. The GBBS Work column shows the work of GBBS algorithms converted to use NVRAM without taking advantage of the small-memory, and corresponds to the GBBS-NVRAM using libvmmalloc experiment (pink dashed-bars in Figure 7). The theoretical performance for the GBBS-MemMode experiment (green dashed-bars in Figure 1) lies in-between the GBBS Work and Sage Work. The vertical text in the first column indicates the technique used to obtain the result in the PSAM: EDGEMAPCHUNKED is the semi-asymmetric traversal in Section 6.4 and Filter is the Graph Filtering method in Section 6.5. † denotes that our algorithm uses $O(n + m/\lg n)$ words of memory. ¶ denotes that our algorithm uses $O(n + m/\lg n)$ words of memory in practice, but requires only $O(n)$ words of memory theoretically. * denotes that a bound holds in expectation and ‡ denotes that a bound holds with high probability or *whp* ($O(kf(n))$ cost with probability at least $1 - 1/n^k$). d_G is the diameter of the graph, Δ is the maximum degree, $L = \min(\sqrt{m}, \Delta) + \lg^2 \Delta \lg n / \lg\lg n$, and P_{it} is the number of iterations of PageRank until convergence. We assume that $m = \Omega(n)$.

	Problem	GBBS Work	Sage Work	Sage Depth
EDGEMAPCHUNKED	**Breadth-First Search**	$O(\omega m)$	$O(m)$	$O(\mathrm{d}_G \lg n)$
	Weighted BFS	$O(\omega m)^*$	$O(m)^*$	$O(\mathrm{d}_G \lg n)^\ddagger$
	Bellman-Ford	$O(\omega \mathrm{d}_G m)$	$O(\mathrm{d}_G m)$	$O(\mathrm{d}_G \lg n)$
	Single-Source Widest Path	$O(\omega \mathrm{d}_G m)$	$O(\mathrm{d}_G m)$	$O(\mathrm{d}_G \lg n)$
	Single-Source Betweenness	$O(\omega m)$	$O(m)$	$O(\mathrm{d}_G \lg n)$
	$O(k)$**-Spanner**	$O(\omega m)^*$	$O(m)^*$	$O(k \lg n)^\ddagger$
	LDD	$O(\omega m)^*$	$O(m)^*$	$O(\lg^2 n)^\ddagger$
	Connectivity	$O(\omega m)^*$	$O(m)^*$	$O(\lg^3 n)^\ddagger$
	Spanning Forest	$O(\omega m)^*$	$O(m)^*$	$O(\lg^3 n)^\ddagger$
	Graph Coloring	$O(\omega m)^*$	$O(m)^*$	$O(\lg n + L \lg \Delta)^*$
	Maxmial Independent Set	$O(\omega m)^*$	$O(m)^*$	$O(\lg^2 n)^\ddagger$
Both	**Biconnectivity**¶	$O(\omega m)^*$	$O(m)^*$	$O(\mathrm{d}_G \lg n + \lg^3 n)^\ddagger$
	Approximate Set Cover†	$O(\omega m)^*$	$O(m)^*$	$O(\lg^3 n)^\ddagger$
Filter	**Triangle Counting**†	$O(\omega(m+n) + m^{3/2})$	$O(m^{3/2})$	$O(\lg n)$
	Maximal Matching†	$O(\omega m)^*$	$O(m)^*$	$O(\lg^3 m)^\ddagger$
	PageRank Iteration	$O(m + \omega n)$	$O(m)$	$O(\lg n)$
	PageRank	$O(P_{it}(m + \omega n))$	$O(P_{it} m)$	$O(P_{it} \lg n)$
	k**-core**	$O(\omega m)^*$	$O(m)^*$	$O(\rho \lg n)^\ddagger$
	Approximate Densest Subgraph	$O(\omega m)$	$O(m)$	$O(\lg^2 n)$

6.3 Sage: A Semi-Asymmetric Graph Engine

Our main approach in Sage is to develop PSAM techniques that perform *no writes to the large-memory*. Using these primitives lets us derive efficient parallel algorithms (i) whose cost is independent of ω, the asymmetry of the underlying NVRAM technology, (ii) that do not contribute to NVRAM wearout or wear-leveling overheads, and (iii) that do not require on-the-fly recompression for compressed graphs. The surprising result of our experimental study is that this strict discipline—to entirely avoid writes to the large-memory—achieves state-of-the-art results in practice. This discipline also enables storage optimizations (discussed in Section 6.8), and perhaps most importantly lends itself to designing provably-efficient parallel algorithms that interact with the graph through high-level primitives.

Semi-Asymmetric EDGEMAP. Our first contribution in Sage is a version of EDGEMAP (defined in Chapter 2) that achieves improved efficiency in the PSAM. The issue with the implementation of EDGEMAP used in Ligra, and subsequent systems (Ligra+ and Julienne) based on Ligra is that although it is work-efficient, it may use significantly more than $O(n)$ space, violating the PSAM model. In this chapter, we design an improved implementation of EDGEMAP which achieves superior performance in the PSAM model (described in Section 6.4). Our result is summarized by the following theorem:

Theorem 4. *There is a PSAM algorithm for EDGEMAP given a* vertexSubset U *that runs in* $O(\sum_{u \in U} d(u))$ *work,* $O(\lg n)$ *depth, and uses at most* $O(n)$ *words of memory in the worst case.*

Semi-Asymmetric Graph Filtering. An important primitive used by many parallel graph algorithms performs *batch-deletions* of edges incident to vertices over the course of the algorithm. A batch-deletion operation is just a bulk remove operation that logically deletes these edges from the graph. These deletions are done to reduce the number of edges that must be examined in the future. For example, four of the algorithms studied in this chapter—biconnectivity, approximate set cover, triangle counting, and maximal matching—utilize this primitive.

In prior work in the shared-memory setting, deleted edges are handled by actually removing them from the adjacency lists in the graph. In these algorithms, deleting edges is important for two reasons. First, it reduces the amount of work done when edges incident to the vertex are examined again, and second, removing the edges is important to bound the theoretical efficiency of the resulting implementations [115, 117]. In the PSAM, however, deleting edges is expensive because it requires writes to the large-memory.

In our Sage algorithms, instead of directly modifying the underlying graph, we build an auxiliary data structure, which we refer to as a ***graphFilter***, that efficiently supports updating a graph with a sequence of deletions. The graphFilter data structure can be viewed as a bit-packed representation of the original graph that supports mutation. Importantly,

this data structure fits into the small-memory of the relaxed version of the PSAM. We formally define our data structure and state our theoretical results in Section 6.5.

Efficient Semi-Asymmetric Graph Algorithms. We use our new semi-asymmetric techniques to design efficient semi-asymmetric graph algorithms for 18 fundamental graph problems. In all but a few cases, the bounds are obtained by applying our new semi-asymmetric techniques in conjunction with existing efficient DRAM-only graph algorithms from Dhulipala et al. [117]. We summarize the PSAM work and depth of the new algorithms designed in this chapter in Table 6.1, and present the detailed results in Section 6.7.

6.4 *Semi-Asymmetric Graph Traversal*

Our first technique is a cache-friendly and memory-efficient sparse EDGEMAP primitive designed for the PSAM. This technique is useful for obtaining PSAM algorithms for many of the problems studied in this thesis. Graph traversals are a basic graph primitive, used throughout many graph algorithms [108, 319]. A graph traversal starts with a frontier (subset) of seed vertices. It then runs a number of iterations, where in each iteration, the edges incident to the current frontier are explored, and vertices in this neighborhood are added to the next frontier based on some user-defined conditions.

Existing Memory-Inefficient Graph Traversal

Ligra implements the direction-optimization proposed by Beamer [44], which runs either a *sparse* (push-based) or *dense* (pull-based) traversal, based on the number of edges incident to the current frontier. The sparse traversal processes the out-edges of the current frontier to generate the next frontier. The dense traversal processes the in-edges of all vertices, and checks whether they have a neighbor in the current frontier. Ligra uses a threshold to select a method, which by default is a constant fraction of m to ensure work-efficiency.

The dense method is memory-efficient—theoretically, it only requires $O(n)$ bits to store whether each output vertex is on the next frontier. However, the sparse method can be memory-inefficient because it allocates an array with size proportional to the number of edges incident to the current frontier, which can be up to $O(m)$. In the PSAM, an array of this size can only be allocated in the large-memory, so the traversal is inefficient. This is also true for the real graphs and machines that we tested in this chapter.

The GBBS algorithms [117] use a *blocked* sparse traversal, referred to as EDGEMAP-BLOCKED, that improves the cache-efficiency of parallel graph traversals by only writing to as many cache lines as the size of the newly generated frontier. This technique is not memory-efficient, as it allocates an intermediate array with size proportional to the number of edges incident to the current frontier, which can be up to $O(m)$ in the worst-case.

Memory-Efficient Traversal: EDGEMAPCHUNKED

In this chapter, we present a chunk-based approach that improves the memory-efficiency of the sparse (push-based) EDGEMAP. Our approach, which we refer to as EDGEMAPCHUNKED, achieves the same cache performance as the EDGEMAPBLOCKED implementation used in GBBS [117], but significantly improves the intermediate memory usage of the approach.

We provide pseudocode for our algorithm in Algorithm 23. The algorithm is based on three types of chunking. First, the algorithm breaks up the edges incident to each vertex into *blocks* based on the underlying graph's block size. Second, the algorithm chunks the outgoing edges to traverse, which it breaks up into *groups*. Each group consists of some number of blocks. The blocks within a group will be processed sequentially, but different groups can be processed in parallel. Third, the algorithm performs *chunking* of the output that is generated, writing out the neighbors that must be emitted in the output vertexSubset into fixed-size *chunks*.

The algorithm first breaks each vertex up into units of work called blocks based on a block size parameter, G_{b_size} (Line 11). This block size can be tuned arbitrarily for uncompressed graphs, but must be set equal to the compression block size for compressed graphs. Different settings of the block size result in a tradeoff between the depth of the algorithm and the amount of small-memory used in the PSAM model. We discuss this tradeoff, and how to set this block size below. Next, the algorithm decides the number of groups to create (Lines 13–18), where each group consists of a set of blocks. It then processes the groups in parallel. For each group, it processes the blocks within the group one at a time. When starting the next block, it calls the FETCHCHUNK procedure (Lines 4–9), which returns an output chunk for the current group, allocating a fresh chunk if the current chunk is too full (Line 22). Each group stores the chunks allocated for it in a per-group vector of output chunks (V from Line 19), which can be accessed safely without any atomics, since each group is processed by a single thread. The chunk allocations are done in our implementation using a pool-based thread-local allocator (Line 3). Next, the algorithm processes the block and writes all neighbors that should be emitted in the next vertexSubset into the chunk, and updates the block size (Line 23). Note that the FETCHCHUNK procedure ensures that the returned chunk has sufficient space to store all neighbors in the block being processed (Line 6). The remaining steps aggregate the chunks from the per-group vectors (Line 24), perform prefix sum on the chunk sizes (Line 26), and copy the data within the chunks into an array with size proportional to the number of returned neighbors (Lines 28–30). After copying the data within a chunk, the algorithm frees the chunk (Line 30). Finally, the algorithm returns the output vertexSubset (Line 31).

Memory Usage, Work, and Depth. First note that in the degenerate case where *all* edges are processed using our implementation, the code can create up to m/G_{b_size} many blocks, which can be $\Omega(n)$. Instead, we ensure that $G_{b\_size} = d_{avg} = \lceil m/n \rceil$, or the average degree. In this case, the maximum number of blocks used is $m/G_{b\_size} = m/d_{avg} = O(n)$. It

Algorithm 23 EDGEMAPCHUNKED

```
1: chunk_size = max(4096, d_avg)
2: min_group_size = max(4096, d_avg)
3: chunk_alloc := Initialize thread-local allocator with chunk_size
4: procedure FETCHCHUNK(b_size, V)
5:    chunk := Last chunk from V                ▷ V : vector of output chunks for this group
6:    if chunk = null or chunk.SIZE + b_size > chunk_size then
7:       chunk := chunk_alloc.ALLOCATE()
8:       Insert chunk into V
9:    return chunk
10: procedure EDGEMAPCHUNKED(G, U, F)
11:    G_{b_size} := d_avg                       ▷ underlying block size used in G
12:    B := Output blocks corresponding to u ∈ U
13:    O := Prefix sums of block-degrees for blocks in B
14:    d_U := Σ_{u∈U} deg(u)
15:    group_size := max(⌈d_U/8p⌉, min_group_size)
16:    num_groups := ⌈d_U/group_size⌉
17:    idxs := {i · group_size | i ∈ [num_groups]}
18:    Offs := Offsets into O resulting from a parallel merge of idxs and O
19:    V := Array of vectors storing chunks of size num_groups
20:    parfor i in [0, |Offs|) do                ▷ In parallel over the groups
21:       for j in [Offs[i], Offs[i + 1]) do     ▷ ≤ G_{b_size} edges in block j
22:          chunk := FETCHCHUNK(G_{b_size}, V_i)
23:          Process vertices in block j by applying F, and write vertices where F returns true into
     chunk
24:    C := All chunks extracted from V
25:    output_offsets := Array of length |C| where the i'th entry contains the size of the i'th chunk,
     C[i]
26:    output_size := Scan(output_offsets, +)
27:    output := Array of size output_size
28:    parfor c ∈ C do
29:       Copy elements in c to output starting at offset in output_offsets corresponding to c
30:       chunk_alloc.RELEASE(c)
31:    return output
```

is simple to check the remainder of the code and observe that the amount of intermediate memory and the output size are bounded by $O(n)$ words. The overall small-memory usage of the procedure is therefore $O(n)$ words. Note that for compressed graphs, the block size is equal to the compression block size, and thus the compression block size must be set to d_{avg} to ensure theoretical efficiency in the PSAM. For uncompressed graphs the block size

can be set arbitrarily.

To ensure that we do not create an unnecessarily large number of groups, we set the number of groups to $O(\min(8p, \sum_{u \in U} deg(u)/min\_group\_size))$ on p processors (Lines 14–16 and Line 2). These parameters balance between providing enough parallel slackness for work-stealing when there is a large amount of work to be done (the $8p$ term in the min) while also ensuring that we do not over-provision parallelism in the case when the number of edges incident to the frontier is small (the second term in the min).

The overall work of the procedure is $O(\sum_{u \in U} deg(u))$, and the overall depth is $O(\lg n + group\_size)$, since each group is processed sequentially by a single thread, and each call to the procedure requires aggregating the per-group chunks, which can be done in parallel. Note that theoretically, $group\_size = O(d_{avg})$ since the max on Line 15 includes the $\lceil d_U/8p \rceil$ only to avoid creating excess groups when parallelism is abundant, and $min\_group\_size = O(d_{avg})$ (Line 2). The work done for the chunk-aggregation is proportional to the number of groups (and number of output chunks), both of which are upper-bounded by $O(\sum_{u \in U} deg(u))$. d_{avg} is usually a small constant in real-world graphs (see Table 6.2).

6.4.1 Case Study: Breadth-First Search

Algorithm. Figure 6.3 provides the full Sage code used for our implementation of BFS. The algorithm outputs a BFS-tree, but can trivially be modified to output shortest-path distances from the source to all reachable vertices. The user first imports the Sage library (Line 1). The definition of BFSFunc defines the user-defined function supplied to edgeMap (Lines 3–20). The main algorithm, BFS, is templatized over a graph type (Line 21). The BFS code first initializes the parent array P (Line 25), sets the parent of the source vertex to itself (Line 26), and initializes the first frontier to contain just the parent (Line 27). It then loops while the frontier is non-empty (Lines 28–31), and calls edgeMapChunked in each iteration of the while loop (Line 30).

The function supplied to edgeMapChunked is BFSFunc (Lines 3–20), which contains two implementations of update based on whether a sparse or dense traversal is applied (update and updateAtomic respectively), and the function cond indicating whether a neighbor should be visited. This logic is identical to the update function used in BFS in Ligra, and we refer the interested reader to Shun and Blelloch [319] for a detailed explanation.

PSAM: Work-Depth Analysis. The work is calculated as follows. First, the work of initializing the parent array, and constructing the initial frontier is just $O(n)$. The remaining work is to apply edgeMap across all rounds. To bound this quantity, first observe that each vertex, v, processes its out-edges at most once, in the round where it is contained in *Frontier* (if other vertices try to visit v in subsequent rounds notice that the cond function will return *false*). Let R be the set of all rounds run by the algorithm, $W_{\text{edgeMapChunked}}(r)$ be

```
#include "sage.h"
#include <limits>
template <class W, class Int>
struct BFSFunc {
  sequence<Int>& P;
  Int max_int;
  BFSFunc(sequence<Int>& P) : P(P) {
    max_int = std::numeric_limits<Int>::max();}
  bool update(Int s, Int d, W w) {
    if (P[d] == max_int) {
      P[d] = s;
      return 1;
    }
    return 0;
  }
  bool updateAtomic(Int s, Int d, W w) {
    return (CAS(&P[d], max_int, s));
  }
  bool cond(Int d) { return (P[d] == max_int); }
};
template <class Graph, class Int>
sequence<Int> BFS(Graph& G, Int src) {
  using W = typename Graph::weight_type;
  Int max_int = std::numeric_limits<Int>::max();
  auto P = sequence<Int>(G.n, max_int);
  P[src] = src;
  auto frontier = vertexSubset(G.n, src);
  while (!frontier.isEmpty()) {
    auto F = BFSFunc<W, Int>(P);
    frontier = edgeMapChunked(G, frontier, F);
  }
  return P;
}
```

Figure 6.3: Code for Breadth-First Search in Sage.

the work of EDGEMAPCHUNKED on the r-th round, and U_r be the set of vertices in *Frontier* in the r-th round. Then, the work is $\sum_{r \in R} W_{\text{EDGEMAPCHUNKED}}(r) = \sum_{r \in R} \sum_{u \in U_r} d(u) = O(m)$.

The depth to initialize the parents array is $O(\lg n)$, and the depth of each of the r applications of EDGEMAPCHUNKED is $O(\lg n)$ by Theorem 4. Thus, the overall depth is $O(r \lg n) = O(\mathrm{d}_G \lg n)$. The small-memory space used for the parent array is $O(n)$ words, and the maximum space used over all EDGEMAPCHUNKED calls is $O(n)$ words by Theorem 4.

This proves the following theorem:

Theorem 5. *There is a PSAM algorithm for breadth-first search that runs in $O(m)$ work, $O(\mathsf{d}_G \lg n)$ depth, and uses only $O(n)$ words of small-memory.*

6.5 Semi-Asymmetric Graph Filtering

Sage provides a high-level filtering interface that captures both the current implementation of filtering in GBBS, as well as the new mutation-avoiding implementation described in this chapter. The interface provides functions for creating a new graphFilter, filtering edges from a graph based on a user-defined predicate, and a function similar to EDGEMAP which filters edges incident to a subset of vertices based on a user-defined predicate. Since edges incident to a vertex can be deleted over the course of the algorithm by using a graphFilter, we call edges that are currently part of the graph represented by the graphFilter as ***active*** edges.

We first discuss a semantic issue that arises when filtering graphs. Suppose the user builds a filter G_f over a symmetric graph G. If the filtering predicate takes into account the directionality of the edge, then the resulting graph filter can become directed, which is unlikely to be what the user intends. Therefore, we designed the constructor to have the user explicitly specify this decision by indicating whether the user-defined predicate is symmetric or asymmetric, which results in either a symmetric or asymmetric graph filter.

The filtering interface is defined as follows:

- **MAKEFILTER**(G : Graph,
  P : edge $\mapsto$ bool, S : bool) : graphFilter
  Creates a graphFilter G_f for the immutable graph G with respect to the user-defined predicate P, and S, which indicates whether the filter is symmetric or asymmetric.
- **FILTEREDGES**(G_f : graphFilter) : int
  Filters all active edges in G_f that do not satisfy the predicate P from G_f. The function mutates the supplied graphFilter, and returns the number of edges remaining in the graphFilter.
- **EDGEMAPPACK**(G_f : graphFilter,
  S : vertexSubset) : vertexSubset
  Filters edges incident to $v \in S$ that do not satisfy the predicate P from G_f. Returns a vertexSubset on the same vertex set as S, where each vertex is augmented with its new degree in G_f.

Graph Filter Data Structure

For simplicity, we describe the symmetric version of the graph filter data structure. The asymmetric filter follows naturally by using two copies of the data structure described below, one for the in-edges and one for the out-edges.

We first review how edges are represented in Sage. In the (uncompressed) CSR format, the neighbors of a vertex are stored contiguously in an array. If the graph is compressed using one of the parallel compression methods from Ligra+ [322], the incident edges are divided into a number of *compression blocks*, where each block is sequentially encoded using a difference-encoding scheme with variable-length codes. Each block must be sequentially decoded to retrieve the neighbor IDs within the block, but by choosing an appropriate block size, the edges incident to a high-degree vertex can be traversed in parallel across the blocks.

The graph filter's design *mirrors the CSR representation* described above. The design of our structure is inspired by similar bit-packed structures, most notably the cuckoo-filter by Eppstein et al. [134]. Figure 6.4 illustrates the graph filter for the following description.

Definition. The graph data is stored in the compressed sparse row (CSR) format on NVRAM, and is read-only. Each vertex's incident edges are logically divided into blocks of size $\mathcal{F}_B$, the ***filter block size***, which is the provided block size rounded up to the next multiple of the number of bits in a machine word, inclusive (64 bits on modern architectures and $\lg n$ bits in theory). In Figure 6.4, $\mathcal{F}_B = 2$. For compressed graphs, this block size is always equal to the compression block size (and thus, both must be tuned together).

The filter consists of blocks corresponding to a subset of the logical blocks in the edges array, and is stored in DRAM. Each vertex stores a pointer to the start of its blocks, which are stored contiguously. For each block, the filter stores $\mathcal{F}_B$ many bits, where the bits correspond one-to-one to the edges in the block. Each block also stores two words of metadata: (i) the *original block-ID* in the adjacency list that the block corresponds to, and (ii) the *offset*, which stores the number of active edges before this block. The original block-IDs are necessary because over the course of the algorithm, only a subset of the original blocks used for a vertex may be currently present in the graph filter, and the data structure must remember the original position of each block. The offset is needed for graph primitives which copy all active edges incident to a vertex into an array with size proportional to the degree of the vertex.

The overall graph filter structure thus consists of blocks of bitsets per vertex. It stores the per-vertex blocks contiguously, and stores an offset to the start of each vertex's blocks. It also stores each vertex's current degree, as well as the number of blocks in the vertex structure. Finally, the structure stores an additional n bits of memory which are used to mark vertices as dirty.

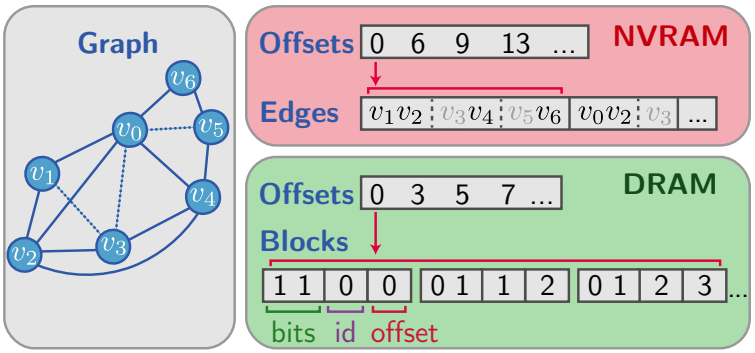


Figure 6.4: The graph data is stored in the compressed sparse row (CSR) format on NVRAM, and is read-only. Each vertex's incident edges are logically divided into blocks of size $\mathcal{F}_B$ each ($\mathcal{F}_B$ = 2 here). The filter is structured similarly to the CSR format and is stored in DRAM. It consists of blocks corresponding to a subset of the logical blocks in the edges array. Each vertex stores a pointer to the start of its blocks, which are stored contiguously. Each filter block stores bits representing the status of the edges in the logical block it corresponds to. In the figure, deleted edges are marked with dotted lines, and correspond to bits set to 0 in the filter. Each block also stores an offset storing the number of active edges (edges with bits set to 1) in all preceding blocks for this vertex, as well as its original block-id, which are both used by our algorithms. When an edge is deleted, its corresponding bit is set to 0, and the offsets of the blocks for the vertex are updated accordingly. The original graph data in stored in the CSR format and is stored on NVRAM and is read-only. On DRAM, we maintain the filter structure that consists of blocks. Each block corresponds to $\mathcal{F}_B$ many edges in a consecutive range, and stores $\mathcal{F}_B$ many bits for these edges. In addition it stores an offset and the original block-id for each block, which are used by our algorithms. When an edge is deleted, its corresponding bit is set to 0, and the offsets of the blocks for the vertex are updated accordingly. Once a constant fraction of blocks are empty (e.g., the gray block), we physically delete all the empty blocks from the filter structure to guarantee work-efficiency. All writes are on DRAM.

Algorithms

MakeFilter. To create a graph filter, the algorithm first computes the number of blocks that each vertex requires, based on $\mathcal{F}_B$, and writes the space required per vertex into an array. Next, it prefix sums the array, and allocates the required $O(m)$ bits of memory contiguously. It then initializes the per-vertex blocks in parallel, setting all edges as active (their corresponding bit is set to 1). Finally, it allocates an array of n per-vertex structures storing the degree, offset into the bitset structure corresponding to the start of the vertex's blocks, and the number of blocks for that vertex. Lastly, it initializes per-vertex dirty bits to false (not dirty) in parallel.

The overall work to create the filter is $O(m)$ and the overall depth is $O(\lg n + \mathcal{F}_B)$, because a block is processed sequentially. If the user specifies that the initially supplied

predicate returns false for some edges, the implementation calls FILTEREDGES (described below), which runs within the same work and depth bounds.

PACKVERTEX. Next, we describe an algorithm to pack out the edges incident to a vertex given a predicate P. This algorithm is used to implement both FILTEREDGES and EDGEMAPPACK. Note that the framework does not expose this primitive directly to users, since the number of edges in the graph is not updated by this primitive.

The algorithm first maps over all blocks incident to the vertex in parallel. For each block, it finds all active bits in the block, reads the edge corresponding to the active bit and applies the predicate P, unsetting the bit iff the predicate returns false. If the bit for an edge (u, v) is unset, the algorithm marks the dirty bit for v to true if needed. Note that for uncompressed graphs, the edge corresponding to an active bit can be directly read, whereas for a compressed graph, the entire block may have to be decoded to retrieve the value of a particular edge.

The algorithm maintains a count of how many bits are still active while processing the block, and stores the per-block counts in an array of size equal to the number of blocks. Next, it performs a reduction over this array to compute the number of blocks with at least one active edge. If this value is less than a constant fraction of the current number of blocks incident to the vertex, the algorithm filters out all of these blocks with no active elements, and packs the remaining blocks contiguously in the same memory, using a parallel filter over the blocks. The algorithm then updates the offsets for all blocks using a prefix sum. Finally, the algorithm updates the vertex degree and number of currently active blocks incident to the vertex.

The overall work is $O(A \cdot (\mathcal{F}_B/\lg n) + d_{active}(v))$ and the depth is $O(\lg n + \mathcal{F}_B)$, where A is the number of non-empty blocks corresponding to v and $d_{active}(v)$ is the number of active edges incident to vertex v.

EDGEMAPPACK. The EDGEMAPPACK primitive is implemented by applying PACKVERTEX to each vertex in the vertexSubset in parallel. It then updates the number of active edges by performing a reduction over the new vertex degrees in the vertexSubset. The overall work is the sum of the work for packing out each vertex in the vertexSubset, S, which is $O(A \cdot (\mathcal{F}_B/\lg n) + \sum_{v \in S}(1 + d_{active}(v)))$, and the depth is $O(\lg n + \mathcal{F}_B)$, where A is the number of non-empty blocks corresponding to all $v \in S$.

FILTEREDGES. The FILTEREDGES primitive uses the EDGEMAPPACK, providing a vertexSubset containing all vertices. The work is $O(n + A \cdot (\mathcal{F}_B/\lg n) + |E_{active}|)$, and the depth is $O(\lg n + \mathcal{F}_B)$, where A is the number of non-empty blocks in the graph and E_{active} is the set of active edges represented by the graph filter.

Vertex Primitives. When calling vertex primitives on a graph filter, such as accessing the degree of the vertex, or mapping over its incident active edges, we first check whether the vertex is marked dirty. If so, we pack it out using the algorithm described above before performing the operation.

Implementation

Optimizations. We use the widely available TZCNT and BLSR x86 intrinsics to accelerate block processing. Each block is logically divided into a number of machine words, so we consider processing a single machine word. If the word is non-zero, we create a temporary copy of the word, and loop while this copy is non-zero. In each iteration, we use TZCNT to find the index of the next lowest bit, and clear the lowest bit using BLSR. Doing so allows us to process a block with q words and k non-zero bits in $O(q + k)$ instructions.

We also implemented intersection primitives, which are used in our triangle counting algorithm based on the decoding implementation described above. For compressed graphs, since we may have to decode an entire compressed block to fetch a single active edge, we immediately decompress the entire block and store it locally in the iterator's memory. We then process the graph filter's bits word-by-word using the intrinsic-based algorithm described above.

Finally, we avoid the use of dirty bits in the algorithms using the graph filtering structure studied in this chapter by supplying the framework's primitives with a flag which informs the framework that the algorithm will not operate on vertices which are not packed. By avoiding using the dirty bits, our algorithms avoid an extra random-write of a neighbor's dirty-bit when performing pack operations, an a random-read to check a dirty-bit when performing operations on vertices.

Memory Usage. The overall memory requirement of a graphFilter is $3n$ words to store the degrees, offsets, and number of blocks, plus $O(m)$ bits to store the bitset data and the metadata. The metadata increases the memory usage by a constant factor, since $\mathcal{F}_B$ is at least the size of a machine word, and so the metadata stored per block can be amortized against the bits stored in the block. The overall memory usage is therefore $O(n + m/\lg n)$ words of memory. For our uncompressed inputs, the size of the graph filter is 4.6–8.1x smaller than the size of the uncompressed graph. For our compressed inputs, the size of the filter is 2.7–2.9x smaller than the size of the compressed graph.

6.6 Semi-Asymmetric Bucketing

We now briefly describe how to adapt the work-efficient bucketing structure from Julienne [115] to the PSAM. A bucketing structure maintains a dynamic mapping between a set of *elements* and *buckets*, and is used in several important graph algorithms for work-efficiency: weighted breadth-first search, k-core, approximate densest subgraph, and approximate set cover. The bucketing strategy in Julienne is based on *lazy bucketing*, which avoids deleting the bucketed elements from buckets that they are moved out of. If the elements that are bucketed are the vertices, and the total number of bucket updates is

$O(m)$, then the use of lazy bucketing will require the bucket structure to use $O(m)$ words of small-memory, violating the PSAM requirements.

We can address this issue by using *semi-eager bucketing*. In the semi-eager version, each bucket maintains two counters, storing the number of live (not logically deleted) elements currently in the bucket, and the number of dead (logically deleted) elements. When moving a vertex out of a bucket, we increment the dead element count in that bucket. When a bucket contains more than a constant factor of dead elements, we physically pack them out. Since each vertex is contained in a single bucket, this approach only uses $O(n)$ words of small-memory for bucketing vertices.

In practice, we use the practical variant of the bucketing structure proposed in Julienne [115], which is based on maintaining a constant number of active buckets with the highest priority, and an overflow bucket for all remaining vertices. This approach also uses only $O(n)$ words of small-memory for bucketing vertices.

6.7 Semi-Asymmetric Graph Algorithms

We now describe Sage's efficient parallel graph algorithms in the PSAM model. Our results and theoretical bounds are summarized in Table 6.1. The bounds are obtained by combining efficient in-memory algorithms in our prior work from Chapter 5 with the new semi-asymmetric techniques designed in Sections 6.4 and 6.5. We elide some of the details of the theoretical results by describing how the results are obtained based on the results presented in Chapter 5. Specifications of the graph problems can be found in Chapter 2.

Shortest Path Problems

Algorithms. We consider six shortest-path problems in this chapter: ***breadth-first search (BFS)***, ***integral-weight SSSP (wBFS)***, ***general-weight SSSP (Bellman-Ford)***, ***single-source betweenness centrality***, ***single-source widest path***, and ***$O(k)$-spanner***. Our BFS, Bellman-Ford, and betweenness centrality implementations are based on those in Ligra [319], and our wBFS implementation is based on the one in Julienne (Chapter 4). We provide two implementations of the single-source widest path algorithm, one based on Bellman-Ford, and another based on the wBFS implementation from Julienne (Chapter 4). An $O(k)$-spanner is a subgraph that preserves shortest-path distances within a factor of $O(k)$. Our $O(k)$-spanner implementation is based on an algorithm by Miller et al. [248].

Efficiency in the PSAM. Our theoretical bounds for these problems in the PSAM are obtained by using the edgeMapChunked primitive (Section 6.4) for performing sparse graph traversals, because all of these algorithms can be expressed as iteratively performing edgeMapChunked over subsets of vertices. The proofs are similar to the proof we provide for BFS in Section 6.4.1 and rely on Theorem 4. Note that the bucketing data structure

used in Julienne (Chapter 4) requires only $O(n)$ words of space to bucket vertices, and thus automatically fits in the PSAM model. The Miller et al. construction builds an $O(k)$-spanner with size $O(n^{1+1/k})$, and runs in $O(m)$ expected work and $O(k \lg n)$ depth *whp*. Our implementation in Sage runs our low-diameter decomposition algorithm, which is efficient in the PSAM as we describe below. We set k to be $\Theta(\lg n)$, which results in a spanner with size $O(n)$.

Connectivity Problems

Algorithms. We consider four connectivity problems in this chapter: ***low-diameter decomposition (LDD)***, ***connectivity***, ***spanning forest***, and ***biconnectivity***. Our implementations are extensions of the implementations provided in GBBS [117].

Efficiency in the PSAM. First, we replace the calls to edgeMapBlocked in each algorithm with calls to edgeMapChunked, which ensures that the graph traversal step uses $O(n)$ words of small-memory using Theorem 4. This modification results in PSAM algorithms for LDD. For the other connectivity-like algorithms that use LDD, namely connectivity, spanning forest, biconnectivity, we use the improved analysis of LDD provided in [248] to argue that the number of inter-cluster edges after applying LDD with $\beta = O(1)$ is $O(n)$ in expectation. Specifically, we use Corollary 3.1 of [248], which we reproduce here for completeness:

Lemma 3 (Corollary 3.1 of [248])**.** *In an exponential start time decomposition with parameter $\beta = \frac{\lg n}{2k}$, for any vertex $v \in V$ the ball $B(v, 1) = \{u \in V \mid d(u, v) \leq 1\}$ intersects $O(n^{1/k})$ clusters in expectation.*

The number of clusters the radius-1 ball around a vertex intersects is exactly the number of inter-cluster edges incident to the vertex. Applying Lemma 3 with $k = c \lg n$, we have that $\beta = \frac{1}{2c}$, for a suitable constant c, and that the number of inter-cluster edges incident to v is $O(n^{1/c \lg n}) = O(1)$ in expectation.

Thus, applying this lemma across all vertices, by linearity the number of inter-cluster edges in G drops to $O(n)$ in expectation after applying the LDD once, when $\beta = \frac{1}{2c}$ for some constant c. At this point, the entire graph on n vertices, and all inter-cluster edges can be built in the fast memory, and the previous algorithm of Shun et al. [321] can be run in the fast memory on this new graph. By the analysis above, and the fact that the Shun et al. [321] algorithm runs in $O(m)$ expected work and $O(\lg^3 n)$ depth *whp*, we have the following result for connectivity in the PSAM.

Theorem 6. *There is a parallel connectivity algorithm that performs $O(m)$ expected work and $O(\lg^3 n)$ depth* whp *in the PSAM model with $O(n)$ words of memory.*

An identical argument holds for spanning forest, and biconnectivity (which uses the connectivity algorithm as a subroutine).

Corollary 1. *There are algorithms for computing spanning forest and biconnectivity that perform $O(m)$ expected work each, and $O(\lg^3 n)$ depth and $O(d_G \lg n + \lg^3 n)$ depth* whp *in the PSAM model with $O(n)$ words of memory.*

We note that our biconnectivity implementation utilizes the filtering structure (Section 6.5) to accelerate a call to connectivity that operates on a graph with most of the edges removed, and thus our implementation runs in the relaxed PSAM model for practical efficiency.

Handling Restarts in Low-Diameter Decomposition-Based Algorithms. One issue that arises when running in the PSAM is that algorithms which use $O(n)$ words of small-memory *in expectation* may need to be restarted if the space bound is violated. This situation can occur for several of our algorithms which are based on running a low-diameter decomposition, and contracting, or selecting inter-cluster edges based on the decomposition. In what follows, we focus on the connectivity algorithm, since our spanning forest, biconnectivity, and $O(k)$-spanner algorithm are also handled identically.

In our connectivity algorithm, we can ensure that the algorithm runs in $O(n)$ words of space by re-running the LDD algorithm until it succeeds. Checking whether an LDD succeeds can be done in $O(n)$ space and $O(n + m)$ work by simply counting the number of inter-cluster edges formed by the partition of vertices. Once the LDD succeeds, the rest of the algorithm does not require restarts, since the recursive calls in the connectivity algorithm always fit within $O(n)$ words of space [321]. We note that since the LDD succeeds with constant probability, an expected constant number of iterations are needed for the connectivity algorithm to succeed. Therefore, the work bounds obtained using restarting are still $O(m)$ in expectation. Furthermore, the depth bound is not affected by the restarts since we only need to perform restarts at the first level of recursion in the algorithm. Since we must perform at most $O(\lg n)$ restarts at this level to guarantee success *whp*, the overall contribution to the depth of re-running an LDD at this step is $O(\lg^2 n)$ *whp*, which is subsumed by the algorithm's overall depth.

Covering Problems

Algorithms. We consider four covering problems in this chapter: ***maximal independent set (MIS)***, ***maximal matching***, ***graph coloring***, and ***approximate set cover***. All of our implementations are extensions of our previous work in GBBS [117].

Efficiency in the PSAM. For MIS and graph coloring, we derive PSAM algorithms by applying our EDGEMAPCHUNKED optimization because other than graph traversals, both algorithms already use $O(n)$ words of small-memory.

Our maximal matching algorithm runs phases of the filtering-based maximal matching algorithm described in [117] on subsets of the edges such that they fit in small-memory. The algorithm has access to $O(n + m/\lg n)$ words of memory, so we can solve the problem by extracting the next $O(m/\lg n)$ unprocessed edges on each phase, and running the random-priority based maximal matching algorithm on the subset of edges. The algorithm will finish after performing $O(\lg n)$ phases. This does not affect the overall work and increases the depth by an $O(\lg n)$ factor. The depth of applying maximal matching within a phase is $O(\lg^2 n)$ *whp* using the analysis of Fischer and Noever [141], combined with the implementation from Blelloch et al. [64]. Thus, the overall depth of the algorithm is $O(\lg^3 n)$ *whp*. In practice, we use a similar approach that is theoretically motivated and makes use of our graph-filtering structure. In each phase, we extract $O(n)$ unmatched edges, and process them using the random-priority based algorithm [64]. All unmatched edges from this set are discarded, and the graph is filtered using FILTEREDGES to pack out edges incident to matched vertices. Theoretically, we can switch to the previously described version after a constant number of such phases. However, we observed that in practice, a constant number of iterations of the filtering procedure suffices for all graphs that we tested on.

Our approximate set cover implementation is similar to the implementation from [117], with the exception that the underlying filtering is done using a graph filter, instead of mutating the original graph. The bounds obtained for the graph filter structure match the bounds on filtering used in the GBBS code, which mutates the underlying graph, and so our implementation also computes a $(1+\epsilon)$-approximate set cover in $O(m)$ expected work and $O(\lg^3 n)$ depth *whp*.

Substructure Problems

Algorithms. We consider three substructure-based problems in this chapter: ***k-core***, ***approximate densest subgraph*** and ***triangle counting***. Substructure problems are fundamental building blocks for community detection and network analysis (e.g., [82, 180, 217, 298, 300, 362]). Our k-core and triangle-counting implementations are based on the implementation from GBBS [117].

Efficiency in the PSAM. For the k-core algorithm to use $O(n)$ words of small-memory, it should use the fetch-and-add based implementation of k-core, which performs atomic accumulation in an array in order to update the degrees. However, the fetch-and-add based implementation performs poorly in practice, where it incurs high contention to update the degrees of vertices incident to many removed vertices [117]. Therefore, in practice we use a *histogram-based* implementation, which always runs faster than the fetch-and-add based implementation (the histogram primitive is fully described in [117]). In this chapter, we implemented a *dense* version of the histogram routine, which performs reads for all vertices in the case where the number of neighbors of the current frontier is higher than

a threshold t. The work of the dense version is $O(m)$. Using $t = m/c$ for some constant c ensures work-efficiency, and results in low memory usage for sparse calls in practice. Our approximate densest subgraph algorithm is similar to our k-core algorithm, and uses a histogram to accelerate processing the removal of vertices. The code uses the dense histogram optimization described above.

Our triangle counting code is based on the GBBS implementation, which is an implementation of the parallel triangle counting algorithm from Shun and Tangwongsan [323]. Our implementation uses the graph filter structure to orient edges in the graph from lower degree to higher degree. Since we only require outgoing edges, we supply a flag to the framework which permits it to only represent the outgoing edges of the directed graph, halving the amount of internal memory required. Our implementation uses the new iterator implemented over the graph filter structure to perform intersections between the outgoing edges of two vertices. We note that in our implementation, we perform intersection sequentially, which theoretically guarantees a depth of $O(\sqrt{m})$. However, the parallelism of this algorithm is $O(m^{3/2}/m^{1/2}) = O(m)$ which in practice is sufficiently high that reducing the depth using a parallel intersection routine is unnecessary, and can hurt performance due to using a more complicated intersection algorithm.

Theoretically, the bounds for Triangle Counting in Table 6.1 can be obtained by using a parallel intersection method on the filter structure. The implementation of this idea for the blocked adjacency lists in the filter structure is identical to the $O(\lg n)$ depth intersection method defined on compressed graphs in [117]. The idea is similar to the classic parallel intersection, or merge algorithms on two sorted arrays [188, 62].

Eigenvector Problems

Algorithms. We consider the ***PageRank*** algorithm, designed to rank the importance of vertices in a graph [85]. Our PageRank implementation is based on the implementation from Ligra.

Efficiency in the PSAM. We optimized the Ligra implementation to improve the depth of the algorithm. The implementation from Ligra runs dense iterations, where the aggregation step for each vertex (reading its neighbor's PageRank contributions) is done sequentially. In Sage, we implemented a reduction-based method that reduces over these neighbors using a parallel reduce. Therefore, each iteration of our implementation requires $O(m)$ work and $O(\lg n)$ depth. The overall work is $O(P_{it} \cdot m)$ and depth is $O(P_{it} \lg n)$, where P_{it} is the number of iterations required to run PageRank to convergence with a convergence threshold of $\epsilon = 10^{-6}$.

6.8 Experiments

Overview of Results. After describing the experimental setup (Section 6.8.1), we show the following main experimental results:

- **Section 6.8.2:** Our NUMA-optimized graph storage approach outperforms naive (and natural) approaches by 6.2x.
- **Section 6.8.3:** Sage achieves between 31–51x speedup for shortest path problems, 28–53x speedup for connectivity problems, 16–49x speedup for covering problems, 9–63x speedup for substructure problems and 42–56x speedup for eigenvector problems.
- **Section 6.8.4:** Compared to existing state-of-the-art DRAM-only graph analytics, Sage run on NVRAM is 1.17x faster on average on our largest graph that fits in DRAM. Sage run on NVRAM is only 5% slower on average than when run entirely in DRAM.
- **Section 6.8.5:** We study how Sage compares to other NVRAM approaches for graphs that are larger than DRAM. We find that Sage on NVRAM using App-Direct Mode is 1.94x faster on average than the recent state-of-the-art Galois codes [152] run using Memory Mode. Compared to GBBS codes run using Memory Mode, Sage is 1.87x faster on average across all 18 problems.
- **Section 6.8.6:** We compare Sage with existing state-of-the-art semi-external memory graph processing systems, including FlashGraph, Mosaic, and GridGraph and find that our times are 9.3x, 12x, and 8024x faster on average, respectively.

6.8.1 Experimental Setup

Machine Configuration

We run our experiments on a 48-core, 2-socket machine (with two-way hyper-threading) with 2 × 2.2Ghz Intel 24-core Cascade Lake processors (with 33MB L3 cache) and 375GB of DRAM. The machine has 3.024TB of NVRAM spread across 12 252GB DIMMs (6 per socket). All of our *speedup* numbers report running times on a single thread (T1) divided by running times on *48-cores with hyper-threading* (T96). Our programs are compiled with the `g++` compiler (version 7.3.0) with the `-O3` flag. We use the command `numactl -i all` for our parallel experiments. Our programs use a work-stealing scheduler that we implemented, implemented similarly to Cilk [79].

NVRAM Configuration

NVRAM Modes. The NVRAM we use (Optane DC Persistent Memory) can be configured in two distinct modes. In ***Memory Mode***, the DRAM acts like a direct-mapped cache between L3 and the NVRAM for each socket. Memory Mode transparently provides access to higher memory capacity without software modification. In this mode, the read-write

Table 6.2: Graph inputs, number of vertices, edges, and average degree (d_{avg}).

Graph Dataset	Num. Vertices	Num. Edges	d_{avg}
LiveJournal [81]	4,847,571	85,702,474	17.6
com-Orkut [378]	3,072,627	234,370,166	76.2
Twitter [208]	41,652,231	2,405,026,092	57.7
ClueWeb [81]	978,408,098	74,744,358,622	76.3
Hyperlink2014 [241]	1,724,573,718	124,141,874,032	72.0
Hyperlink2012 [241]	3,563,602,789	225,840,663,232	63.3

asymmetry of NVRAM is obscured by the DRAM cache, and causes the DRAM hit rate to dominate memory performance. In ***App-Direct Mode***, NVRAM acts as byte-addressable storage independent of DRAM, providing developers with direct access to the NVRAM.

Sage Configuration. In Sage, we configure the NVRAM to use App-Direct Mode. The devices are configured using the FSDAX mode, which removes the page cache from the I/O path for the device and allows MMAP to directly map to the underlying memory.

Graph Storage. The approach we use in Sage is to store two separate copies of the graph, one copy on the local NVRAM of each socket. Threads can determine which socket they are running on by reading a thread-local variable, and access the socket-local copy of the graph. We discuss the approach in detail in Section 6.8.2

Graph Data

To show how our algorithms perform on graphs at different scales, we selected a representative set of real-world graphs of varying sizes. These graphs are Web graphs and social networks, which are low-diameter graphs that are frequently used in practice. We list the graphs used in our experiments in Table 6.2, which we symmetrized to obtain larger graphs and so that all of the algorithms would work on them. Hyperlink 2012 is the largest publicly-available real-world graph. We create weighted graphs for evaluating weighted BFS, Bellman-Ford, and Widest Path by selecting edge weights in the range $[1, \lg n)$ uniformly at random. We process the ClueWeb, Hyperlink2014, and Hyperlink2012 graphs in the parallel byte-encoded compression format from Ligra+ [322], and process LiveJournal, com-Orkut, and Twitter in the uncompressed (CSR) format.

6.8.2 Graph Layout in NVRAM

While building Sage, we observed startingly poor performance of cross-socket reads to graph data stored on NVRAM. We designed a simple micro-benchmark that illustrates this behavior. The benchmark runs over all vertices in parallel. For the i-th vertex, it counts the number of neighbors incident to it by reducing over all of its incident edges. It then writes this value to an array location corresponding to the i-th vertex. The graph is stored

in CSR format, and so the benchmark reads each vertex offset exactly once, and reads the edges incident to each vertex exactly once. Therefore the total number of reads from the NVRAM is proportional to $n + m$, and the number of (in-memory) writes is proportional to n.

For the ClueWeb graph, we observed that running the benchmark with the graph on one socket using all 48 hyper-threads on the *same socket* results in a running time of 7.1 seconds. However, using `numactl -i all`, and running the benchmark on all threads across both sockets results in a running time of 26.7 seconds, which is *3.7x worse*, despite using twice as many hyper-threads. While we are uncertain as to the underlying reason for this slowdown, one possible reason could be the granularity size for the current generation of NVRAM DIMMs, which have a larger effective cache line size of 256 bytes [186], and a relatively small cache within the physical NVM device. Using too many threads could cause thrashing, which is a possible explanation of the slowdowns we observed when scaling up reads to a single NVRAM device by increasing the number of threads. To the best of our knowledge, this significant slowdown has not been observed before, and understanding how to mitigate it is an interesting question for future work.

As described earlier, our approach in Sage is to store two separate copies of the graph, one on the local NVRAM of each socket. Using this configuration, our micro-benchmark runs in 4.3 seconds using all 96 hyper-threads, which is 1.6x faster than the single-socket experiment and *6.2x faster* than using threads across both sockets to the graph stored locally within a single socket.

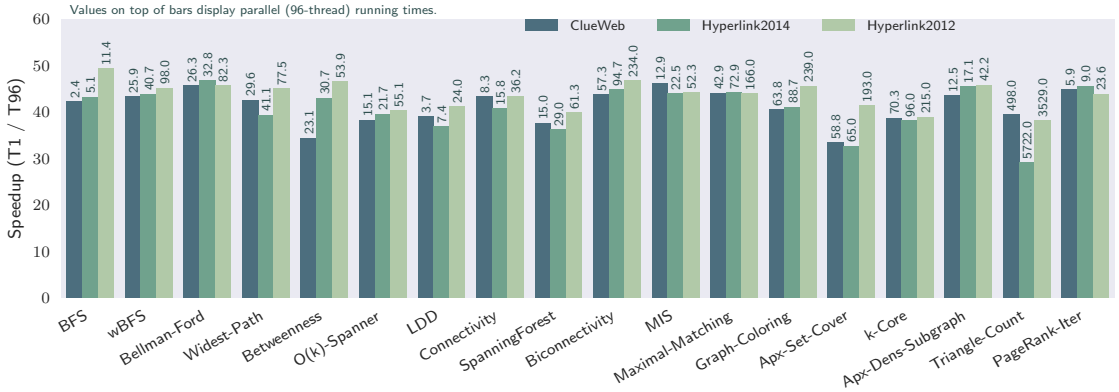


Figure 6.5: Speedup of Sage algorithms on large graph inputs on a 48-core machine (with 2-way hyper-threading), measured relative to the algorithm's single-thread time. All algorithms are run using NVRAM in App-Direct Mode. Each bar is annotated with the parallel running time on top of the bar.

6.8.3 Scalability

Figure 6.5 shows the speedup obtained on our machine for Sage implementations on our large graphs, annotating each bar with the parallel running time. In all of these experiments, we store all of the graph data in NVRAM and use DRAM for all temporary data.

Shortest Path Problems. Our BFS, weighted BFS, Bellman-Ford, and betweenness centrality implementations achieve between parallel speedups of 31–51x across all inputs. For $O(k)$-Spanner, we achieve 39–51x speedups across all inputs. All Sage codes use the memory-efficient sparse traversal (i.e., EDGEMAPCHUNKED) designed in this chapter. We note that the new weighted-SSSP implementations using EDGEMAPCHUNKED are up to 2x more memory-efficient than the implementations from [117]. We ran our $O(k)$-Spanner implementation with k set to $\lceil \lg_2 n \rceil$ by default.

Connectivity Problems. Our low-diameter decomposition implementation achieves a speedup of 28–42x across all inputs. Our connectivity and spanning forest implementations, which use the new filtering structure from Section 6.5, achieve speedups of 37–53x across all inputs. Our biconnectivity implementation achieves a speed up of 38–46x across all inputs. We found that setting $\beta = 0.2$ in the LDD-based algorithms (connectivity, spanning forest, and biconnectivity) performs best in practice, and creates significantly fewer than $m\beta = m/5$ inter-cluster edges predicted by the theoretical bound [246], due to many duplicate edges that get removed.

Covering Problems. Our MIS, maximal matching, and graph coloring implementations achieve speedups of 43–49x, 33–44x, and 16–39x, respectively. Our MIS implementation is similar to the implementation from GBBS. Our maximal matching implementation implements several new optimizations over the implementation from GBBS, such as using a parallel hash table to aggregate edges that will be processed in a given round. These optimizations result in our code (using the graph filter) running faster than the original code when run in DRAM-only, outperforming the 72-core DRAM-only times reported in [117] for some graphs (we discuss the speedup of Sage over GBBS in Section 6.8.4).

Substructure Problems. Our k-core, approximate densest subgraph, and triangle counting implementations achieve speedups of 9–38x, 43–48x, and 29–63x, respectively. Our code achieves similar speedups and running times on NVRAM compared to the previous times reported in [117]. We ran the approximate densest subgraph implementation with $\epsilon = 0.001$, which produces subgraphs of similar density to the 2-approximation of Charikar [96]. Lastly, the Sage triangle counting algorithm uses the iterator defined over graph filters to perform parallel intersection. The performance of our implementation is affected by the number of edges that must be decoded for compressed graph inputs, and we discuss this in detail in Section 6.8.

Eigenvector Problems. Our PageRank implementation achieves a parallel speedup of

42–56x. Our implementation is based on the PageRank implementation from Ligra, and improves the parallel scalability of the Ligra-based code by aggregating the neighbor's contributions for a given vertex in parallel. We ran our PageRank implementation with $\epsilon = 10^{-6}$ and a damping factor of 0.85.

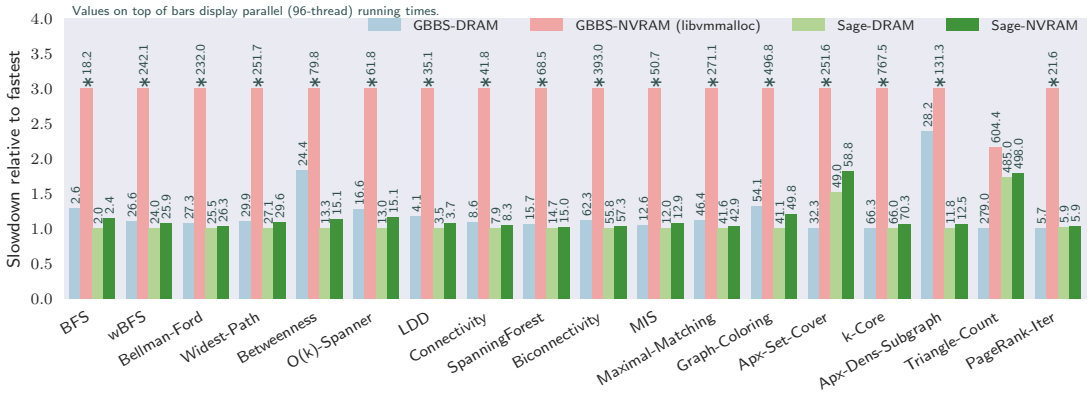


Figure 6.6: Performance of Sage on the ClueWeb graph compared with existing state-of-the-art *in-memory* graph processing systems in terms of slowdown relative to the fastest system (smaller is better). GBBS refers to the DRAM-only codes developed in Chapter 5, and Sage refers to the parallel semi-asymmetric codes developed in this chapter. Both codes are run in two configurations: DRAM measures the running time when the graph is stored in memory, and NVRAM measures the running time when the graph is stored in non-volatile memory, and accessed either using the techniques developed in this chapter (Sage-NVRAM) or using LIBVMMALLOC to automatically convert the DRAM-only codes from GBBS to work using non-volatile memory (GBBS-NVRAM). We truncate relative times slower than 3x and mark the tops of these bars with $*$. All bars are annotated with the parallel running times of the codes on a 48-core system with 2-way hyper-threading. Note that the ClueWeb graph is the largest graph dataset studied in this thesis that fits in the main memory of this machine.

6.8.4 NVRAM vs. DRAM Performance

In this section, we study how fast Sage is compared to state-of-the-art shared-memory graph processing codes, when these codes are run *entirely in DRAM*. For these experiments, we study the ClueWeb graph since it is the largest graph among our inputs where both the graph and all intermediate algorithm-specific data fully resides in the DRAM of our machine. We consider the following configurations:

(**1**) GBBS codes run entirely in DRAM
(**2**) GBBS codes converted to use NVRAM using libvmmalloc (a robust NVRAM memory allocator)
(**3**) Sage codes run entirely in DRAM

(**4**) Sage codes run using NVRAM in App-Direct Mode

Setting (**2**) is relevant since it captures the performance of a naive approach to obtaining NVRAM-friendly code, which is to simply run existing shared-memory code using a NVRAM memory allocator.

Figure 6.6 displays the results of these experiments. Comparing Sage to GBBS when both systems are run in memory shows that our code is faster than the original GBBS implementations by 1.17x on average (between 2.38x faster to 1.73x slower). The notable exception is for triangle counting, where Sage is 1.73x slower than the GBBS code (both run in memory). The reason for this difference is due to the input-ordering the graph is provided in, and is explained in detail in the full version of the paper this chapter is based on [122]. A number of Sage implementations, like connectivity and approximate densest subgraph, are faster than the GBBS implementations due to optimizations in our codes that are absent in GBBS, such as a faster implementation of graph contraction. Our read-only codes when run using NVRAM are only about 5% slower on average than when run using DRAM-only. This difference in performance is likely due to the higher cost of NVRAM reads compared to DRAM reads. Finally, Sage is always faster than GBBS when run on NVRAM using libvmmalloc, and is 6.69x faster on average.

These results show that for a wide range of parallel graph algorithms, Sage significantly outperforms a naive approach that converts DRAM codes to NVRAM ones, is often faster than the fastest DRAM-only codes when run in DRAM, and is competitive with the fastest DRAM-only running times when run in NVRAM.

6.8.5 Alternate NVRAM approaches

We now compare Sage to the fastest available NVRAM approaches when the input graph *is larger than the DRAM size of the machine*. We focus on the Hyperlink2012 graph, which is our only graph where both the graph and intermediate algorithm data are larger than DRAM. We first compare Sage to the Galois-based implementations by Gill et al. [152], which use NVRAM configured in Memory Mode. We then compare Sage to the unmodified shared-memory codes from GBBS modified to use NVRAM configured in Memory Mode.

Comparison with Galois [152]. Gill et al. [152] study the performance of several state-of-the-art graph processing systems, including Galois [260], GBBS [117], GraphIt [383], and GAP [45] when run on NVRAM configured to use Memory Mode. Their experiments are run on a nearly identical machine to ours, with the same amount of DRAM. However, their machine has 6.144TB of NVRAM (12 NVRAM DIMMs with 512GB of capacity each).

Gill et al. [152] find that their Galois-based codes outperform GAP, GraphIt, and GBBS by between 3.8x, 1.9x, and 1.6x on average, respectively, for three large graphs inputs, including the Hyperlink2012 graph. In our experiments running GBBS on NVRAM using MemoryMode, we find that the GBBS performance using MemoryMode is 1.3x slower on average than Galois. There are several possible reasons for the small difference. First,

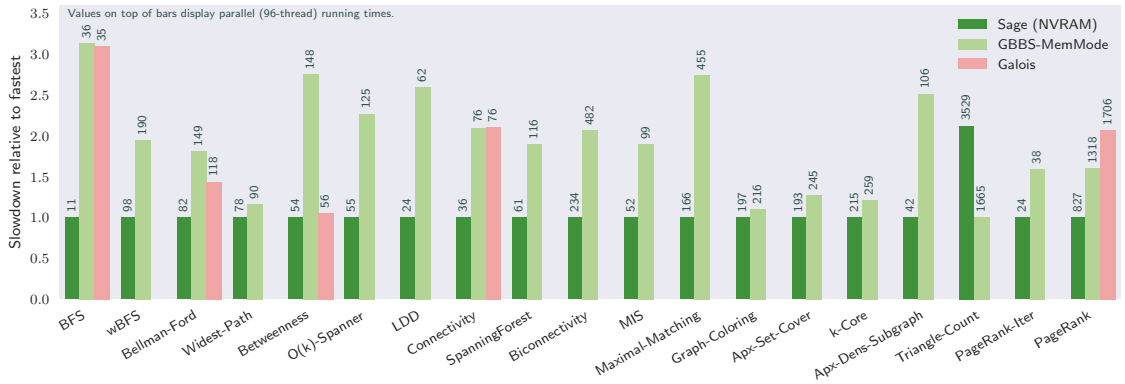


Figure 6.7: Performance of Sage on the Hyperlink2012 graph compared with existing state-of-the-art systems for processing *larger-than-memory graphs* using NVRAM measured relative to the fastest system (smaller is better). Sage (NVRAM) are the new codes developed in this chapter, GBBS-MemMode is the code developed in [117] run using MemoryMode, and Galois is the NVRAM codes from [152]. The bars are annotated with the parallel running times (in seconds) of the codes on a 48-core system with 2-way hyper-threading. Note that the Hyperlink2012 graph *does not fit in DRAM* for the machine used in these experiments.

Gill et al. [152] use the *directed* version of the Hyperlink2012 graph, which has 1.75x fewer edges than the symmetrized version (225.8B vs. 128.7B edges). The symmetrized graph exhibits a massive connected component containing 94% of the vertices, which a graph search algorithm must process for most source vertices. However, a search from the largest SCC in the directed graph reaches about half the vertices [240]. Second, they do not enable compression in GBBS, which is important for reducing the number of cache-misses and NVRAM reads. Lastly, we found that transparent huge pages (THP) significantly improves performance, while they did not, which may be due to differences regarding THP configuration on the different machines.

Figure 6.7 shows results for their Galois-based system on the directed Hyperlink2012 graph. Compared with their NVRAM codes, Sage is 1.04–3.08x faster than their fastest reported times, and 1.94x faster on average. Their codes use the maximum degree vertex in the directed graph as the source for BFS, SSSP, and betweenness centrality. We use the maximum degree vertex in the *symmetric* graph, and note that running on the symmetric graph is more challenging, since our codes must process more edges.

Despite the fact that our algorithm must perform more work, our running times for BFS are 3.08x faster than the time reported for Galois, and our SSSP time is 1.43x faster. For connectivity and PageRank, our times are 2.09x faster and 2.12x faster respectively. For betweenness, our times are 1.04x faster. The authors also report running times for an implementation of k-core that computes a *single* k-core, for a given value of k. This requires significantly fewer rounds than the k-core computation studied in this thesis,

which computes the coreness number of *every vertex*, or the largest k such that the vertex participates in the k-core. They report that their code requires 49.2 seconds to find the 100-core of the Hyperlink2012 graph. Our code finds all k-cores of this graph in 259 seconds, which requires running 130,728 iterations of the peeling algorithm and also discovers the value of the largest k-core supported by the graph ($k_{\max} = 10565$).

In summary, we find that using Sage on NVRAM using App-Direct Mode is 1.94x faster on average than the Galois codes run using Memory Mode.

Algorithms using Memory Mode. Next, we compare Sage to the unmodified shared-memory codes from GBBS modified to use NVRAM configured in Memory Mode. We run these Memory Mode experiments on the same machine with 3TB of NVRAM, where 1.5TB is configured to be used in Memory Mode.

Figure 6.7 reports the parallel running times of both Sage codes using NVRAM, and the GBBS codes using NVRAM configured in Memory Mode for the Hyperlink2012 graph. The results show that in all but one case (triangle counting) our running times are faster (between 1.15–2.92x). For triangle counting, the directed version of the Hyperlink2012 graph fits in about 180GB of memory, which fits within the DRAM of our machine and will therefore reside in memory. We note that we also ran Memory Mode experiments on the ClueWeb graph, which fits in memory. The running times were only 5–10% slower compared to the DRAM-only running times for the same GBBS codes reported in Figure 6.6, indicating a small overhead due to Memory Mode when the data fits in memory.

In summary, our results for this experiment show that the techniques developed in this chapter produce meaningful improvements (1.87x speedup on average, across all 18 problems) over simply running unmodified shared-memory graph algorithms using Memory Mode to handle graph sizes that are larger than DRAM.

6.8.6 External and Semi-External Systems

In this section we place Sage's performance in context by comparing it to existing state-of-the-art semi-external memory graph processing systems. Table 6.3 shows the running times and system configurations for state-of-the-art results on semi-external memory graph processing systems. We report the published results presented by the authors of these systems to give a high-level comparison due to the fact that (i) our machine does not have parallel SSD devices that most of these systems require, and (ii) modifying them to use NVRAM would be a serious research undertaking in its own right.

FlashGraph. FlashGraph [385] is a semi-external memory graph engine that stores vertex data in memory and stores the edge lists in an array of SSDs. Their system is optimized for I/Os at a flash page granularity (4KB), and merges I/O requests to maximize throughput. FlashGraph provides a vertex-centric API, and thus cannot implement some of the work-optimal algorithms designed in Sage, like our connectivity, biconnectivity, or parallel set cover algorithms.

Table 6.3: System configurations (memory in terabytes and threads (hyper-threads)) and running times (seconds) of existing semi-external memory results on the Hyperlink graphs. The last section shows our running times (note that our system is also equipped with NVRAM DIMMs). *These problems are run on directed versions of the graph.

Paper	Problem	Graph	Mem	Threads	Time
FlashGraph [385]	BFS*	2012	.512	64	208
	BC*	2012	.512	64	595
	Connectivity*	2012	.512	64	461
	PageRank*	2012	.512	64	2041
	TC*	2012	.512	64	7818
Mosaic [225]	BFS*	2014	0.768	1000	6.55
	Connectivity*	2014	0.768	1000	708
	PageRank (1 iter.)*	2014	0.768	1000	21.6
	SSSP*	2014	0.768	1000	8.6
Sage	BFS	2014	0.375	96	5.10
	SSSP	2014	0.375	96	32.8
	Connectivity	2014	0.375	96	15.8
	PageRank (1 iter.)	2014	0.375	96	8.99
	BFS	2012	0.375	96	11.4
	BC	2012	0.375	96	53.9
	Connectivity	2012	0.375	96	36.2
	SSSP	2012	0.375	96	82.3
	PageRank	2012	0.375	96	827
	TC	2012	0.375	96	3529

We report running times for FlashGraph for Hyperlink2012 on a 32-core 2-way hyper-threaded machine with 512GB of memory and *15 SSDs* in Table 6.3). Compared to FlashGraph, the Sage times are *9.3x faster on average*. Our BFS and BC times are 18.2x and 11x faster, and our connectivity, PageRank and triangle counting implementations are 12.7x, 2.4x faster, and 2.2x faster, respectively. We note that our times are on the symmetric version of the Hyperlink2012 graph which has twice the edges, where a BFS from a random seed hits the massive component containing 95% of the vertices (BFSes on the directed graph reach about 30% of the vertices).

Mosaic. Mosaic [225] is a hybrid engine supporting semi-external memory processing based on a Hilbert-ordered data structure. Mosaic uses co-processors (Xeon Phis) to offload edge-centric processing, allowing host processors to perform vertex-centric operations. Giving a full description of their complex execution strategy is not possible in this space, but at a high level, it is based on exploiting the fact that user-programs are written in a vertex-centric model.

We report the running times for Mosaic run using 1000 hyper-threads, 768GB of RAM, and 6 NVMes in Table 6.3. Compared with their times, Sage is 12x faster on average, solving

BFS 1.2x faster, connectivity 44.8x faster, SSSP 3.8x slower, and 1-iteration of PageRank 2.4x faster. Given that both SSSP and PageRank are implemented using an SpMV like algorithm in their system, we are not sure why their PageRank times are 2.5x slower than the total time of an SSSP computation. In our experiments, the most costly iteration of Bellman-Ford takes roughly the same amount of time as a single PageRank iteration since both algorithms require similar memory accesses in this step. Sage solves a much broader range of problems compared to Mosaic, and is often faster than it.

GridGraph. GridGraph is an out-of-core graph engine based on a 2-dimensional grid representation of graphs. Their processing scheme ensures that only a subset of the vertex-values accessed and written to are in memory at a given time. GridGraph also offers a mechanism similar to edge filtering which prevents streaming edges from disk if they are inactive. Like FlashGraph, GridGraph is a vertex-centric system and thus cannot implement algorithms that do not fit in this restricted computational model.

The authors consider significantly smaller graphs than those used in our experiments (the largest is a 6.64B edge WebGraph). However, they do solve the LiveJournal and Twitter graphs that we use. For the Twitter graph, our BFS and Connectivity times are 15690x and 359x faster respectively than theirs (our speedups for LiveJournal are similar). GridGraph does not use direction optimization, which is likely why their BFS times are much slower.

6.9 *Related Work*

A significant amount of research has focused on reducing expensive writes to NVRAMs. Early work has designed algorithms for database operators [98, 357, 358]. Blelloch et al. [53, 76, 69] define computational models to capture the asymmetric read-write cost on NVRAMs, and many algorithms and lower bounds have been obtained based on the models [51, 65, 74, 160, 187]. Other models and systems to reduce writes or memory footprint on NVRAMs have also been described [28, 27, 90, 94, 97, 211, 221, 261, 266, 311, 292].

Persistence is a key property of NVRAMs due to their non-volatility. Many new persistent data structures have been designed for NVRAMs [29, 50, 99, 103, 271, 318]. There has also been research on automatic recovery schemes and transactional memory for NVRAMs [18, 109, 214, 220, 361, 381, 387]. There are several recent papers benchmarking performance on NVRAMs [186, 219, 293].

Parallel graph processing frameworks have received significant attention due to the need to quickly analyze large graphs [296]. The only previous graph processing work targeting NVRAMs is the concurrent work by Gill et al. [152], which we discuss in Section 6.8.5. Dhulipala et al. [115, 117] design the Graph Based Benchmark Suite, and show that the largest publicly-available graph, the Hyperlink2012 graph, can be efficiently processed on a single multicore machine. We compare with these algorithms in Section 6.8.

Other multicore frameworks include Galois [260], Ligra [319, 322], Polymer [380], Gemini [389], GraphGrind [338], Green-Marl [177], Grazelle [159], and GraphIt [383]. We refer the reader to [21, 235, 315, 377] for excellent surveys of this growing literature.

6.10 Discussion

In this chapter, we introduced Sage, which takes a semi-asymmetric approach to designing parallel graph algorithms that avoid writing to the NVRAM and uses DRAM proportional to the number of vertices. We have designed a new model, the Parallel Semi-Asymmetric Model, and have shown that all of our algorithms in Sage are provably efficient, and often work-optimal in the model. Our empirical study shows that Sage graph algorithms can bridge the performance gap between NVRAM and DRAM. This enables NVRAMs, which are more cost-efficient and support larger capacities than traditional DRAM, to be used for large-scale graph processing. Interesting directions for future work include studying which filtering algorithms can be made to use only $O(n)$ words of DRAM, and to study how Sage performs relative to existing NVRAM graph-processing approaches on synthetic graphs with trillions of edges.

Part II

Batch-Dynamic Graph Algorithms

Introduction

This part of the thesis develops efficient parallel algorithms in the batch-dynamic setting. Traditional (sequential) dynamic algorithms were motivated by applications where data undergos small changes that can be adequately handled by updates of single elements. Today, however, applications operate on increasingly large datasets that undergo rapid changes over time: for example, millions of individuals can simultaneously interact with a web site, make phone calls, send emails and so on. In the context of these applications, traditional dynamic algorithms require serializing the changes made and processing them *one at a time*, missing an opportunity to exploit the parallelism afforded by processing batches of changes.

Motivated by such applications, there has been recent interest in developing theoretically efficient parallel batch-dynamic algorithms [9, 327, 4]. In the batch-dynamic setting, instead of applying one update or query at a time, a whole batch is applied. A batch could be of size $\lg n$, $\sqrt{n}$, or $n/\lg n$ for example. There are two advantages of applying operations in batches.

1. Batching operations allows for more parallelism.
2. Batching operations can reduce the cost of each update.

In this part of the thesis we are interested in developing algorithms that enjoy both of these advantages. We use the term parallel batch-dynamic to mean algorithms that process batches of operations instead of single ones, and for which the algorithm itself is parallel. The underlying parallel model used in this part is primarily the binary-forking (BF) model, presented in Chapter 2, although in Chapter 8 we briefly consider the CRCW PRAM, as one of our algorithms achieves better depth in the CRCW PRAM compared to the BF model.

We start this part of the thesis by considering the classic dynamic trees problem, introduced by Sleator and Tarjan [328]. In this problem, the objective is to maintain a forest that dynamically undergoes changes in the form of edge insertions and deletions (*links* and *cuts*). The objective is to process the updates and also answer queries that ask whether two vertices are in the same tree in the forest. Chapter 7 designs a parallel batch-dynamic algorithm for this problem based on the classic Euler tour tree data structure [173, 249]. Our new algorithm is based on an new concurrent skip-list data structure designed in the BF model which is work-efficient for single updates, but achieves asymptotically faster batch bounds when processing multiple updates. We then apply this structure to design an Euler tour tree data structure that supports parallel bulk updates. Chapter 8 then builds upon this structure for representing dynamic forests to design a parallel batch-dynamic data structure for connectivity on general graphs. Our data structure is based on the classic

multilevel data structure of Holm, de Lichtenberg, and Thorup [175] (HDT) data structure. Each level of our data structure uses a parallel batch-dynamic forest data structure, with the top-level forest representing a spanning forest of the input graph. We also design parallel procedures for the non-tree edge search procedure in the HDT structure, which sequentially searches for an edge spanning the new cut induced by a deleted spanning forest edge. Our main contribution is a parallel batch-dynamic connectivity data structure that is work-efficient with respect to the HDT data structure, and runs in polylogarithmic worst-case depth. Finally, we also show that our algorithms perform asymptotically lower amortized work than the HDT data structure for sufficiently large batches of deletions.

The results in this part of the thesis have appeared in the following publications:

- Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. "Batch-Parallel Euler Tour Trees". In: *ALENEX*. 2019
- Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. "Parallel Batch-Dynamic Graph Connectivity". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2019, pp. 381–392

7 Parallel Batch-Dynamic Forest Connectivity

7.1 Introduction

In the ***dynamic trees problem*** proposed by Sleator and Tarjan [328], the objective is to maintain a forest that undergoes *link* and *cut* operations. A ***link*** operation adds an edge to the forest, and a ***cut*** operation deletes an edge. Additionally, we want to maintain useful information about the forest. Most commonly we are concerned with whether pairs of vertices are connected (i.e., in the same tree), but we might also be interested in properties like the size of each tree in the forest. Sleator and Tarjan first studied the dynamic trees problem in order to develop fast network flow algorithms [328]. Dynamic trees are also an important component of many dynamic graph algorithms [328, 173, 175, 19, 193].

In the batch-dynamic version of the dynamic trees problem, the objective is to maintain a forest that undergoes *batches* of link and cut operations. Though many sequential data structures exist to maintain dynamic trees, the only parallel batch-dynamic data structure is a recent result by Acar et al. [4]. Their data structure is based on parallelizing RC-trees, which require transforming the represented forest to have bounded degree for the sake of efficiency [6]. Obtaining a data structure without this restriction is therefore of interest. Furthermore, it is of intellectual interest whether the arguably simplest solution to the dynamic trees problem, Euler tour trees (ETTs), can be parallelized.

In this chapter, we answer this question in the affirmative and show that Euler tour trees, a data structure introduced by Henzinger and King [173] and Miltersen et al. [249], achieve asymptotically optimal work and optimal depth in the parallel batch-dynamic setting. We also develop a parallel batch-dynamic skip list upon which we build our Euler tour trees. Note that batching is not only useful for parallel applications but also for single-threaded applications; our $O(k \lg(1 + n/k))$ work bounds for k operations over n elements on Euler tour trees and augmented skip lists beat the $O(k \lg n)$ bounds achieved by performing each operation one at a time on standard sequential data structures.

Summary of Results. Our main contributions in this chapter are as follows:

Skip lists for simple, efficient parallel joins and parallel splits. We show that we can perform k joins or k splits over n skip list elements with $O(k \lg(1 + n/k))$ expected work and $O(\lg n)$ depth *whp*. To the best of our knowledge, we are the first to demonstrate such efficiency for batch joins and splits on a sequence data structure supporting fast search. Our skip list data structure can also be augmented to support efficient computation over contiguous subsequences within the same efficiency bounds.

A parallel Euler tour tree. We apply our skip lists to develop Euler tour trees that support parallel bulk updates. Our Euler tour tree algorithms for adding and for removing

a batch of k edges achieve $O(k \lg(1 + n/k))$ expected work and $O(\lg n)$ depth *whp*. These are the *best known bounds for the batch-parallel dynamic trees problem.*

Experimental evidence of good performance. Our skip list and Euler tour tree data structures achieve good self-relative speedups, ranging from 67–96x for large batch sizes on 72 cores with hyper-threading in our experiments. We also show that they significantly outperform the fastest existing sequential alternatives.

7.2 Sequences and Parallel Batch-Dynamic Skip Lists

We start by first specifying a high-level interface for batch-dynamic sequences. We then describe our parallel batch-dynamic skip lists which implement the interface, and finally, end by discussing how our data structure can be extended to support augmentation.

7.2.1 Batch-Dynamic Sequence Interface

The goal of a batch-dynamic sequence data structure is to represent a collection of sequences under batches of operations that split and join the sequences. To *join* two sequences is to concatenate them together. To *split* a sequence A at element x is to separate the sequence into two subsequences, the first of which consists of all elements in A before and including x, the second of which consists of all elements after x.

Sequences. We now give a formal description of the interface for sequences. The data structure supports the following functions:

- **BatchJoin**$(\{(x_1, y_1), \ldots, (x_k, y_k)\})$ takes an array of tuples where the i-th tuple is a pointer to the last element x_i of one sequence and a pointer to the first element y_i of a second sequence. For each tuple, the first sequence is concatenated with the second sequence. For any distinct tuples (x_i, y_i) and (x_j, y_j) in the input, we must have $x_i \neq x_j$ and $y_i \neq y_j$.
- **BatchSplit**$(\{x_1, \ldots, x_k\}$ takes an array of pointers to elements and, for each element x_i, breaks the sequence immediately after x_i.
- **BatchFindRep**$(\{x_1, \ldots, x_k\}$ takes an array of pointers to elements. It returns an array where the i-th entry is the *representative* of the sequence in which x_i lives. The representative is defined so that representative(u) = representative(v) if and only if u and v live in the same sequence. Representatives are invalidated after sequences are modified.

Augmented Sequences. To augment a sequence, we take an associative function $f : D^2 \to D$ where D is an arbitrary domain of values. A value from D is assigned to each

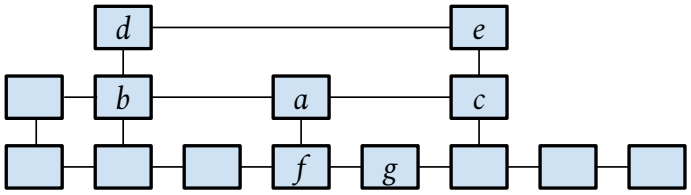


Figure 7.1: An example skip list over a sequence of eight elements. On the bottom are all the level-1 nodes.

element in the sequence, A. An augmented sequence data structure supports querying for the value of f over contiguous subsequences of A. Specifically, our interface for augmented sequences extends the interface for unaugmented sequences with the following functions:

- **BatchUpdateValue**$(\{(x_1, a_1), \ldots, (x_k, a_k)\})$ takes an array of tuples where the i-th tuple contains a pointer to an element x_i and a new value $a_i \in D$ for the element. The value for x_i is set to a_i in the sequence.
- **BatchQueryValue**$(\{(x_1, y_1), \ldots, (x_k, y_k)\})$ takes an array of k tuples where the i-th tuple contains pointers to elements x_i and y_i. The return value is an array where the i-th entry holds the value of f applied over the subsequence between x_i and y_i inclusive. For $1 \leq i \leq k$, x_i and y_i must be elements in the same sequence, and y_i must appear after x_i in the sequence.

7.2.2 Skip Lists

Skip lists are a simple randomized data structure that can be used to represent sequences [282]. To represent a sequence, skip lists assign a *height* to each element of the sequence, where each height is drawn independently from a geometric distribution. The ℓ-th level of a skip list consists of a linked list over the subsequence formed by all elements of height at least ℓ. This structure allows efficient search. Figure 7.1 shows an example skip list.

For an element x of height h, we allocate a node v_i for every level $i = 1, 2, \ldots, h$. Each node has four pointers LEFT, RIGHT, UP, and DOWN. We set $v_i \to \text{UP} = v_{i+1}$ and $v_i \to \text{DOWN} = v_{i-1}$ for each i to connect between levels. We set $v_i \to \text{RIGHT}$ to the i-th node of the next element of height at least i and similarly $v_i \to \text{LEFT}$ to the i-th node of the previous element of height at least i.

Our skip lists support *cyclicity*, which is to say that our algorithms are valid even if we link the tail and head of a skip list together. Though this is not conventionally done with sequence data structures, we will find it useful for representing Euler tours of graphs in Section 7.4 since Euler tours are naturally cyclic sequences. We cannot join upon cyclic sequences, but splitting a cyclic sequence at element x corresponds to unraveling it into a linear sequence with its last element being x. Figure 7.2 illustrates joining and splitting on our skip lists.

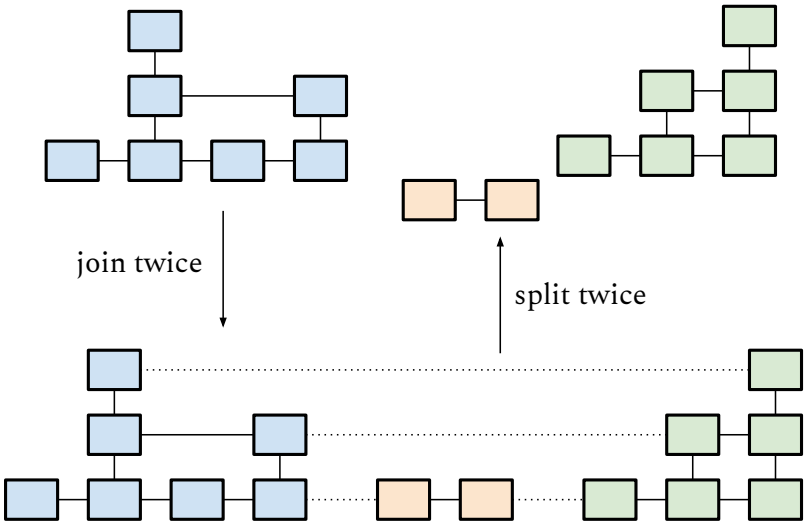


Figure 7.2: Joins and splits on skip lists.

Definitions. We now introduce definitions that describe the relationship between nodes. Say we have a node v that represents element x at some level i. We call $v \to \textsc{right}$ v's *successor*. Similarly, $v \to \textsc{left}$ is its *predecessor*. We call $v \to \textsc{up}$ its *direct parent* and $v \to \textsc{down}$ its *direct child*. For example, in Figure 7.1, consider node a. Its predecessor is b, its successor is c, its direct child is f, and it has no direct parent.

The *left parent* is the level-$(i+1)$ node of the latest element preceding and including x that has height at least $i+1$. The *right parent* is defined symmetrically. Under this definition, if v has a direct parent, then its left and right parents are both its direct parent. When we refer to v's parent, we refer to its left parent. In Figure 7.1, a's (left) parent is d, and a's right parent is e. The *(left) ancestors* consist of v's parent, v's parent's parent, and so on, and similarly for v's *right ancestors*. Thus the ancestors for both f and g in Figure 7.1 are a and d. A *child* is inverse to a parent, and a *descendant* is inverse to an ancestor.

The following definitions describe the relationship between the links connecting nodes. The *parent* of a link between v and its successor is the link between v's parent and its successor. Similarly, the *ancestors* of the link are links between v's ancestors and their successors. The *children* of the link are the links between v's children and their successors.

Joins, Splits, and Augmentation on Skip Lists. Recall that in an augmented sequence, we take an associative function $f : D^2 \to D$ for some domain D. Each element in the sequence A is assigned some value from D. By storing these values in the bottom level of our skip list and storing partial "sums" at higher levels, we can compute f over contiguous subsequences of A in logarithmic time. For instance, in Figure 7.3, we assign the value 1 to every element and choose $f : \mathbb{N}^2 \to \mathbb{N}$ to be the sum function. For each node v, we store the sum of the values of v's children. By looking at $O(\lg n)$ nodes, we can then compute

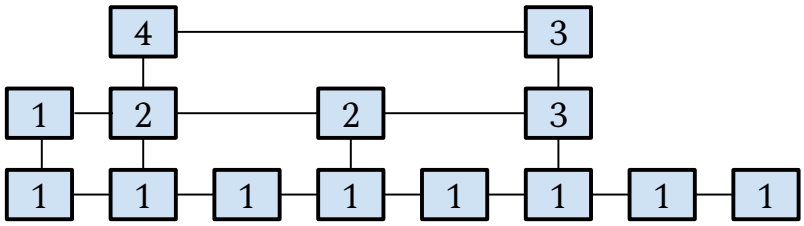


Figure 7.3: Each node in this skip list is augmented with a size value, summing all the values of its children.

the size of the sequence. These augmented values are also easy to maintain as the skip list undergoes joins and splits.

Our skip lists support batch joins, batch splits, batch point updates of augmented values, and batch finding representatives in $O(k \lg(1 + n/k))$ work in expectation and $O(\lg n)$ depth *whp*, where k is the batch size and n is the number of elements in the lists. We analyze efficiency in Section 7.3.

This improves on the $\Theta(k \lg n)$ expected work bound achieved by conventional sequential joins and splits on augmented skip lists. Intuitively, the reason we can achieve improved work-bounds is that if a node has many updated descendants, our algorithm updates its augmented value only once rather than multiple times.

7.2.3 Parallel Batch-Dynamic Algorithms for Unaugmented Lists

Algorithm 24 Creates an element with height distributed according to Geometric$(1 - p)$.

```
1: procedure CreateNode()
2:     allocate node
3:     With probability p:
4:         node → up := CreateNode()
5:         node → up → down := node
6:     return node
```

We begin by describing our algorithms for unaugmented skip lists. For creating elements in our skip list, we fix a probability $0 < p < 1$ representing the expected proportion of nodes at a particular level that have a direct parent at the next level. We generate heights of elements by allocating a node and giving each node a direct parent with probability p independently, as seen in Algorithm 24. This technique is equivalent to drawing heights from a Geometric$(1 - p)$ distribution.

We give pseudocode for Join and Split over unaugmented lists in Algorithms 26 and 45 respectively. To perform a batch of joins in parallel, we simply call Join on each join operation in the batch concurrently, and similarly for a batch of splits. We point out that

Algorithm 25 Searches for the left parent of the input node. The mirror function SearchRight is defined similarly.

```
1: procedure SearchLeft(v)
2:     current := v
3:     while current → up = null do
4:         current :=current → left
5:         if current = null or current = v then
6:             return null
7:     return current → up
```

although batches of splits and joins must be run in separate phases, there is no requirement that within a phase all operations are carried out in a single batch (they can be applied concurrently). Thus, our data structure is actually *phase-concurrent* over joins and splits, which is more general than being batch-parallel [320].

Both algorithms employ two simple helper procedures, SearchLeft and SearchRight, for finding the left and right parents of a node. We show SearchLeft in Algorithm 25, and SearchRight is implemented symmetrically. Note that these procedures avoid looping forever on cyclic skip lists.

Algorithm 26 Joins two lists together given their endpoints.

```
1: procedure Join(v_L, v_R)
2:     if CAS(&v_L → right, null, v_R) then
3:         v_R → left := v_L
4:         parent_L :=SearchLeft(v_L)
5:         parent_R :=SearchRight(v_R)
6:         if parent_L ≠ null and parent_R ≠ null then
7:             Join(parent_L, parent_R)
```

Join. Recall that the definition of Join takes a pointer to the last element of one list and a pointer to the first element of a second list and concatenates the first list with the second list. Starting at the bottom level, our algorithm links the given nodes, searches upwards to find parents to link at the next level, and repeats. We set the link with a CAS, and if the CAS is lost, the algorithm quits. This permits only one thread to set a particular link, preventing repeated work.

Theorem 7. *Let B be a set of valid Join inputs. Then calling Join concurrently over the inputs in B gives the same result as joining over the inputs in B sequentially.*

Proof. (Proof sketch) We argue inductively level-by-level that all necessary links are added and no unnecessary links are added. For the base case, at the bottom level, the links we

add are exactly those given as input to the algorithm, which are the necessary links to add at that level. For the inductive step, assume that the correct links will be added on level i. Consider any link ℓ from nodes v_L to v_R on level $i+1$ that should be added. In order for this to be a link we need to add, there must be a rightward path from v_L's direct child to v_R's direct child once all links on level i are added. Then consider the last execution of JOIN on level i to add a link on that path by finishing line 3 of Algorithm 26. That execution will have a complete path to find parents v_L and v_R when searching and thus will find ℓ as a link to add. Conversely, any level-$(i+1)$ link from nodes v_L to v_R found by a join execution was found via a complete path (albeit perhaps temporarily missing some LEFT pointers due to some executions of join completing line 2 but not yet completing line 3) between v_L's direct child and v_R's direct child, which indicates that this link should be added. Unfortunately the formal proof of this result is somewhat technical, and so we refer the interested reader to the Appendix C.1 of the full version of the paper that this chapter is based on [354]. □

Algorithm 27 Separates the input node from its successor.

```
1: procedure SPLIT(v)
2:     ngh := v → RIGHT
3:     if ngh ≠ null and CAS(&v → RIGHT, ngh, null) then
4:         ngh → LEFT := null
5:         parent := SEARCHLEFT(v)
6:         if parent ≠ null then
7:             SPLIT(parent)
```

Split. SPLIT takes a pointer to an element and breaks the list right after that element. Similar to join, it cuts the link at the bottom level and then loops in searching upwards to find parent links to remove at higher levels. Like JOIN, this uses CAS to avoid duplicate work.

Theorem 8. *Let B be a set of elements. Then calling SPLIT concurrently over the elements in B gives the same result as splitting over the elements in B sequentially.*

Proof. (Proof sketch) Like in the proof sketch of Theorem 7, we look at the links that are removed inductively level-by-level. The argument is similar, except that in the inductive step, to see that a link ℓ on level $i+1$ from nodes v_L to v_R that should be removed will indeed be removed by the phase of splits, we note that the leftmost split on the path from $v_L \to$ DOWN to $v_R \to$ DOWN will be able to find parent v_L in its SEARCHLEFT call. Like the correctness proof for joins, unfortunately the formal proof of this result is somewhat technical, and so we refer the interested reader to the Appendix C.2 of the full version of the paper that this chapter is based on [354]. □

Finding representative nodes.

Algorithm 28 Finds a representative node of the list that the input node lives in.

```
1:  procedure FindRep(v)
2:    while SearchRight(v) ≠ null do
3:      v := SearchRight(v)
4:    while SearchLeft(v) ≠ null do
5:      v := SearchLeft(v)
6:    rep := v
7:    while true do
8:      if v → left = null then                          ▷ list is acyclic
9:        return v
10:     v := v → left
11:     if v = rep then                                  ▷ list is cyclic
12:       return rep
13:     if v < rep then
14:       rep := v
```

A simple phase-concurrent implementation of FindRep that takes $O(k \lg n)$ expected work for k concurrent calls is to start at the input node and walk to the top level of the list. Then on the top level, for an acyclic list, we return the leftmost node, or for a cyclic list, we return the node with the lowest memory address. This is shown in Algorithm 28.

However, if we are given a batch of k calls up front, we can in fact achieve $O(k \lg(1 + n/k))$ expected work and $O(\lg n)$ depth *whp*. The idea is that each call of FindRep takes some path up the skip list to the top level, and calls whose paths intersect somewhere can be combined at that point to avoid duplicate work. Then the return value gets propagated back down to both original calls. The code would look similar to the code for batch updating augmented values for augmented skip lists (Subsection 7.2.4) in Algorithm 30. We omit the full details.

7.2.4 Parallel Batch-Dynamic Algorithms for Augmented Lists

We now describe how to augment our skip lists. In addition to its four pointers, each node is given a value val from some domain D and a boolean needs_update. We provide an associative function $f : D^2 \to D$ and, for each element in the list, a value from D. We assign values to val on nodes at the bottom level and then compute val at higher levels by applying f over nodes' children. The boolean needs_update is initialized to *false* and is used to mark nodes whose values need updating.

We give the main algorithm BatchUpdateValues for batch augmented value update in Algorithm 30. This takes a set of nodes at the bottom level along with values to give to

Algorithm 29 Helper function for BatchUpdateValues that updates the augmented value for v and all its descendants.

```
 1: procedure UpdateTopDown(v)
 2:     v → needs_update := false
 3:     if v → down = null then                          ▷ reached bottom level
 4:         return
 5:     current := v → down
 6:     do
 7:         if current → needs_update then
 8:             spawn UpdateTopDown(current)
 9:         current := current → right
10:     while current ≠ null and current → up = null
11:     sync
12:     sum := v → down → val
13:     current := v → down → right
14:     while current ≠ null and current → up = null do
15:         sum := f(sum, current → val)
16:         current := current → right
17:     v → val := sum
```

Algorithm 30 Takes a batch of (node, value) pairs, updates each node with its associated value, and updates other affected augmented values stored throughout the list.

```
 1: procedure BatchUpdateValues({(v_1, a_1), ..., (v_k, a_k)})
 2:     top := {null, null, ..., null}                   ▷ k-length array
 3:     for i ∈ {1, ..., k} do in parallel
 4:         v_i → val := a_i
 5:         current := v_i
 6:         while CAS(&current → needs_update, false, true) do
 7:             parent := SearchLeft(current)
 8:             if parent = null then
 9:                 top[i] := current
10:                 break
11:             current := parent
12:     for i ∈ {1, ..., k} do in parallel
13:         if top[i] ≠ null then
14:             UpdateTopDown(top[i])
```

the associated elements. For each node in the set, we start by updating its value (line 5). Then each of its ancestors have values that need updating, so we walk up its ancestors, CASing on each ancestor's NEEDS_UPDATE variable (line 6). If an execution loses a CAS, then it may quit because some other execution will take care of all the node's ancestors.

Now over all the input nodes that won all CASes on their ancestors, we know the union of their topmost ancestors' descendants contain all the input nodes. By calling the helper function UPDATETOPDOWN (Algorithm 29) on every such topmost ancestor in lines 12–14, we traverse back down and update these descendants' augmented values. Given a node, this helper function calls itself recursively on all the node's children c who need an update as indicated by $c \to$ NEEDS_UPDATE (lines 5–10). Then, after all the childrens' values are updated, we may update the original node's value (lines 11–17).

Algorithm 31 Batch join for augmented skip lists.

```
1: procedure BATCHJOIN({(l_1, r_1), (l_2, r_2), ..., (l_k, r_k)})
2:     for i ∈ {1, ..., k} do in parallel
3:         JOIN(l_i, r_i)
4:     BATCHUPDATEVALUES({(l_1, l_1 → VAL), ..., (l_k, l_k → VAL)})
```

Algorithm 32 Batch split for augmented skip lists.

```
1: procedure BATCHSPLIT({v_1, v_2, ..., v_k})
2:     for i ∈ {1, ..., k} do in parallel
3:         SPLIT(v_i)
4:     BATCHUPDATEVALUES({(v_1, v_1 → VAL), ..., (v_k, v_k → VAL)})
```

With this algorithm for batch augmented value update, batch joins (Algorithm 31) and batch splits (Algorithm 32) are simple. We first perform all the joins or splits. Then we batch update on the nodes we joined or split on. We keep all the values on the bottom level the same, but the update fixes all the values on the higher levels that are changed by adding or removing links.

A batch of k queries for the augmented values over contiguous subsequences of lists can be processed in $O(k \lg n)$ expected work and $O(\lg n)$ depth *whp* by simply performing each query with Algorithm 33 in parallel at $O(\lg n)$ expected work per query.

7.3 *Efficiency*

Recall that in our skip lists data structures, each node independently has a direct parent with probability $0 < p < 1$.

Algorithm 33 Query for the augmented value over the subsequence between v_L and v_R inclusive. Here we let $f(none, x) = x$ for all $x \in D$.

```
1:  procedure QueryValue(v_L, v_R)
2:      sum_L := none
3:      sum_R := v_R → val
4:      while v_L ≠ v_R do
5:          while v_L → up ≠ null and v_R → up ≠ null do
6:              v_L := v_L → up
7:              v_R := v_R → up
8:          if v_L → up = null then
9:              sum_L := f(sum_L, v_L → val)
10:             v_L := v_L → right
11:         else
12:             v_R := v_R → left
13:             sum_R := f(v_R → val, sum_R)
        return f(sum_L, sum_R)
```

7.3.1 Work

We prove a work bound of $O(k \lg(1 + n/k))$ in expectation over a set of k splits over n elements for unaugmented skip lists. The same strategy proves the bound for joins and for operations on augmented skip lists.

For a set of k splits, we first show that $O(k \lg(1+n/k))$ links need to be cut in expectation. Over the first $h = \left\lceil \lg_{1/p}(1 + n/k) \right\rceil$ levels, each split needs to remove at most h links (one at each level), so there are $O(kh)$ pointers to remove over the first h levels. For each level $k > h$, the number of links to remove is bounded by the number of nodes on the level. The probability that a particular node has height at least ℓ is $p^{\ell-1}$, so the expected number of nodes reaching level ℓ is $np^{\ell-1}$. Then the number of links summed across all levels $\ell > h$ is at most

$$\begin{aligned} n \sum_{\ell=h+1}^{\infty} p^{\ell-1} &= np^h \frac{1}{1-p} \leq np^{\lg_{1/p}(1+n/k)} \frac{1}{1-p} \\ &= \frac{n}{(1-p)(1+n/k)} \leq \frac{n}{(1-p)(n/k)} = O(k). \end{aligned}$$

Therefore, the expected number of links we need to cut in total is $O(kh)+O(k) = O(k \lg(1+n/k))$.

For each link to remove, the amount of work to find that link from the previous child link in a split is $O(1)$ in expectation. To search for a link at level $i + 1$ that needs removal, we call SearchLeft, which walks left from the previous place we removed a link on level i until we see a direct parent. The amount of work is proportional to the number of nodes

we touch when walking left. The probability a node has a direct parent is p independently, so the number of nodes we must touch until we see a direct parent is distributed according to Geometric(p) with expected value $1/(1-p) = O(1)$. (If we quit early due to reaching the beginning of a list or due to detecting a cycle, that only reduces the amount of work we do.) Thus this traversal work to find parent links only affects the expected work by a constant factor.

Moreover, no two split operations can both remove the same link because we remove links with a CAS. Whoever CASes the link first successfully clears the link, and whoever comes afterwards quits. The quitting execution only does $O(1)$ expected extra work from the extra traversal to find the already claimed link. Thus there is no significant duplicate work per split.

Therefore, the work overall for k splits is $O(k \lg(1 + n/k))$.

7.3.2 Depth

For analyzing depth, we know that every search path (a path from the top level of a skip list to a particular node on the bottom level, or the reverse) in an n-element skip list has length $O(\lg n)$ *whp* (a proof is given in [114]). The main critical paths of our operations consist of traversing search paths and doing up to a constant amount of extra work at each step, so we get a depth bound of $O(\lg n)$ *whp* for any of our operations.

7.4 Parallel Batch-Dynamic Euler Tour Trees

In this section we present parallel batch-dynamic Euler tour trees, a solution to the parallel batch-dynamic trees problem. In order to ease exposition, we first present a batch-dynamic interface for the dynamic trees problem.

Batch-Dynamic Trees Interface. A solution to the batch-dynamic trees problem supports representing a forest as it undergoes batches of links, cuts, and connectivity queries. Recall that a *link* links two trees in the forest, and a *cut* deletes an edge from the forest and breaks one tree into two trees. Lastly, *connectivity* queries take two vertices in the forest and return whether they are connected (that is, whether they are in the same tree). We now give a formal description of the interface. The data structure maintains a graph $G = (E, V)$, which is assumed to be a forest under the following operations:

- **BatchLink**($\{\{u_1, v_1\}, \ldots, \{u_k, v_k\}\}$) takes an array of edges and adds them to the graph G. The input edges must not create a cycle in G.
- **BatchCut**($\{\{u_1, v_1\}, \ldots, \{u_k, v_k\}\}$) takes an array of edges and removes them from the graph G.

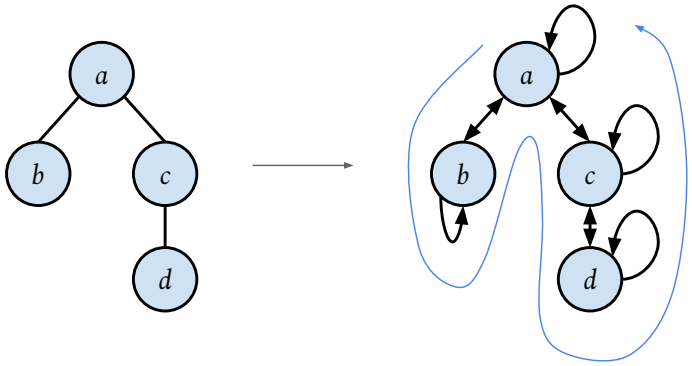


Figure 7.4: We take the tree on the left and transform it so that we get the following Euler tour of edges: (a, b) (b, b) (b, a) (a, c) (c, d) (d, d) (d, c) (c, c) (c, a) (a, a).

- **BatchConnected**$(\{\{u_1, v_1\}, \ldots, \{u_k, v_k\}\})$ takes an array of tuples representing queries. The output is an array where the i-th entry returns whether vertices u_i and v_i are connected by a path in G.

It may also be desirable for the data structure to support augmenting the trees with an associative and commutative function $f : D^2 \to D$ with values from D assigned to vertices and edges of the forest. The goal of augmentation is to compute f over subtrees of the represented forest. Note that we assume that the function is commutative in order to allow implementations to not maintain any specific order over each vertex's children. The interface supports batch updates over vertices and edges. The primitives are similar to the batch updates of values for augmented skip lists, so we elide the details. The interface for subtree queries is different, and we present it below:

- **BatchSubtree**$(\{(u_1, p_1), \ldots, (u_k, p_k)\})$ takes an array of tuples, where the i-th tuple contains a vertex u_i and its parent p_i in the tree. It returns an array where the i-th entry contains the value of f summed over u_i's subtree relative to its parent p_i in G. Note that because the represented trees are unrooted, we require providing the parent p_i in order to determine the intended subtree for u_i.

Note that some dynamic trees data structures allow queries for augmented values summed over *paths* rather than subtrees in the represented forest, but Euler tour trees do not.

Euler Tour Trees. We focus on a variant of Euler tour trees presented by Tarjan [345]. To represent a tree as an Euler tour tree, replace each edge $\{u, v\}$ with two directed edges (u, v) and (v, u) and add a loop (v, v) to each vertex v, as shown in Figure 7.4. This construction produces a connected graph in which each vertex has equal indegree and outdegree, and therefore the graph admits an Euler tour. We represent the tree as any of its Euler tours.

Linking two trees corresponds to splicing their Euler tours together, and cutting a tree corresponds to cutting out part of its Euler tour. Each of these operations reduces to a few joins and splits on the tours. We may also answer whether two vertices u and v are connected by asking whether their loops, (u, u) and (v, v), reside in the same tour. Moreover, because a subtree appears as a contiguous section of an Euler tour, we can efficiently compute information about subtrees if we can efficiently compute information about contiguous sections of tours.

Traditionally, Euler tour trees store an Euler tour by breaking it into a sequence at an arbitrary location and then placing the sequence in a balanced binary tree. We instead store Euler tours as cycles using the skip lists designed in Section 7.2. Because skip lists are easy to join and split in parallel, we can process batches of links and cuts on Euler tour trees efficiently.

We show that for a batch of k joins, k splits, or k connectivity queries over an n-vertex forest, we can achieve $O(k \lg(1 + n/k))$ expected work and $O(\lg n)$ depth *whp*. If we build our Euler tour trees over augmented skip lists, we can also answer subtree queries efficiently.

7.4.1 Description

Our Euler tour trees crucially rely on our parallel skip lists to represent Euler tours. Since a graph of n vertices has Euler tours whose lengths sum to $O(n)$, the skip lists hold $O(n)$ elements. Thus a batch of k joins or splits on the Euler tours takes $O(k \lg(1+n/k))$ expected work and $O(\lg n)$ depth *whp*.

Construction. For clarity, we describe our Euler tour trees using the phase-concurrent unaugmented skip lists given in Section 7.2. However, it is easy to organize the joins and splits into batches so as to match the augmented skip list interface seen in Subsection 7.2.4. We also treat our dictionary data structure as phase-concurrent for clarity, but again, this is easy to circumvent.

We add fields TWIN and MARK to each skip list element. For an element representing a directed edge (u, v), TWIN is a pointer to the element representing the directed edge (v, u) in the opposite direction. We initialize the field MARK to *false* and use it during splitting to mark elements that will be removed.

At initialization (Algorithm 34), the represented graph is an n-vertex forest with no edges, and we assume the vertices are labeled with integers $1, 2, \ldots, n$. We create an n-length array *verts* such that *verts*$[i]$ stores a pointer to the skip list element representing the loop edge (i, i). As such, in parallel, for $i := 1, \ldots, n$, we create a skip list element, assign it to *verts*$[i]$, and join it to itself to form a singleton cycle. These cycles are the Euler tours in an empty graph. We also keep a dictionary *edges* that maps edges (u, v) with $u \neq v$ to corresponding skip list elements. Lastly, we create an array *successors* that will be used as scratch space for batch linking.

Algorithm 34 Euler tour tree data structure initialization.

```
1: procedure Initialize(n)
2:     verts := {}                                   ▷ n-length array
3:     for i ∈ {1, ..., n} do in parallel
4:         verts[i] := CreateNode()
5:         Join(verts[i], verts[i])
6:     edges := dict()                               ▷ empty dictionary
7:     successors := {}                              ▷ n-length array
```

Connectivity queries. To check whether two vertices are connected, we simply check whether they live in the same Euler tour by comparing the representatives of their tours' skip lists. The complexity of this can be made $O(k \lg(1 + n/k))$ expected work and $O(\lg n)$ depth *whp* using an efficient BatchFindRep algorithm.

Batch Link. Algorithm 35 shows our algorithm for adding a batch of edges. The algorithm takes an array of edges A to add as input. We assume that adding the input edges preserves acyclity.

To add a single edge $\{u, v\}$ sequentially, we can find locations where u and v appear in their tours by looking up $verts[u]$ and $verts[v]$. We split on those locations and join the resulting cut up tours back together with new nodes representing (u, v) and (v, u) in between. If we want to add several edges in parallel, we need to be careful when inserting edges that are incident to the same vertex and thus attempt to join on the same location.

With that in mind, we proceed to describe our algorithm. In lines 1-8, for each input edge $\{u, v\}$, we allocate new list elements representing directed edges (u, v) and (v, u). Then, in lines 9-17, for each vertex u that appears in the input, we split u's list at $verts[u]$ as a location to splice in other tours. We also save the successor of $verts[u]$ in $successors[u]$ so that we can join everything back together at the end.

For each vertex u, say that the input tells us that we want to newly connect u to vertices $w_1, w_2, \ldots, w_k$. Then we join together the nodes representing (u, u) to (u, w_1), (w_i, u) to (u, w_{i+1}) for $1 \leq i < k$, and (w_k, u) to what was the successor to (u, u) before splitting. In our code, we arrange this in lines 18-29 by semisorting the input to collect together all edges incident on a vertex. The ordering of $w_1, w_2, \ldots, w_k$ is unimportant, only corresponding to the order in which they appear after u in the Euler tour.

As an example of the desired result from a batch link, consider the graph in Figure 7.5 on which we wish to perform a batch link with input $\{\{a, b\}, \{b, c\}, \{c, e\}\}$. Prior to the batch link, the Euler tour may look like Figure 7.6, and after the batch link, the Euler tour may look like Figure 7.7. Let us focus on what happens to the list containing vertex c. After allocating the new nodes representing the new edges to add, we split node (c, c) from its successor (c, d). We want to add edges connecting c to b and to e. As such, we perform joins adding links from nodes (c, c) to (c, b), (b, c) to (c, e), and (e, c) to (c, d). These new

Algorithm 35 Add a batch of edges to Euler tour tree.

```
1:  procedure BatchLink({{u_1, v_1}, {u_2, v_2}, ..., {u_k, v_k}})
2:    for i ∈ {1, ..., k} do in parallel
3:      uv := CreateNode()
4:      vu := CreateNode()
5:      uv → twin := vu
6:      vu → twin := uv
7:      edges[(u_i, v_i)] := uv
8:      edges[(v_i, u_i)] := vu
9:                                  ▷ cut at locations at which we splice in other tours
10:   for i ∈ {1, ..., k} do in parallel
11:     for w ∈ {u_i, v_i} do
12:       w_node := verts[w]
13:       w_succ := w_node → right
14:       if w_succ ≠ null then
15:                   ▷ benign race; this assignment and split are idempotent
16:         successors[w] := w_succ
17:         Split(w_node)
18:   sorted_edges := Semisort({(u_1, v_1), (v_1, u_1), ..., (u_k, v_k), (v_k, u_k)})
19:                                  ▷ join together tours with new edge nodes in between
20:   for i ∈ {1, ..., 2k} do in parallel
21:     (u, v) := sorted_edges[i]
22:     (u_prev, v_prev) := sorted_edges[i − 1]
23:     (u_next, v_next) := sorted_edges[i + 1]
24:     if i = 1 or u ≠ u_prev then
25:       Join((verts[u], edges[(u, v)]))
26:     if i = 2k or u ≠ u_next then
27:       Join((edges[(v, u)], successors[u]))
28:     else
29:       Join((edges[(v, u)], edges[(u_next, v_next)]))
```

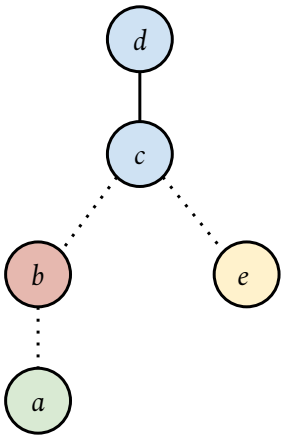


Figure 7.5: An example graph for illustrating batch linking. The dashed edges $\{a, b\}, \{b, c\}, \{c, e\}$ are new edges to add in a batch, whereas the solid edge is an already existing edge.

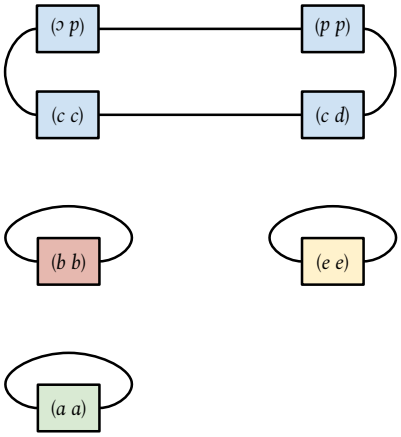


Figure 7.6: We represent the graph from Figure 7.5 prior to adding the dashed edges with Euler tours stored in cyclic linked lists.

links correspond to the blue links in figure 7.7. Doing this over all vertices provides all the joins needed to form an Euler tour over the whole graph.

Using our skip lists and an efficient semisort [161], we see that the work is $O(k \lg(1 + n/k))$ in expectation, and the depth is $O(\lg n)$ *whp*.

Batch Cut. Algorithm 37 describes how to remove a batch of edges. Our algorithm assumes that each edge exists in the forest and that there are no duplicates.

Cutting a single edge is simple. If we cut an edge $\{u, v\}$, we split before and after (u, v) and (v, u) in the tour and join their neighbors together appropriately. However, as with batch linking, the task gets more difficult if we want to cut many edges out of a single node, because those neighbors that we want to join together may themselves be split off.

As an example, consider the graph in Figure 7.8 in which we remove four edges. If our

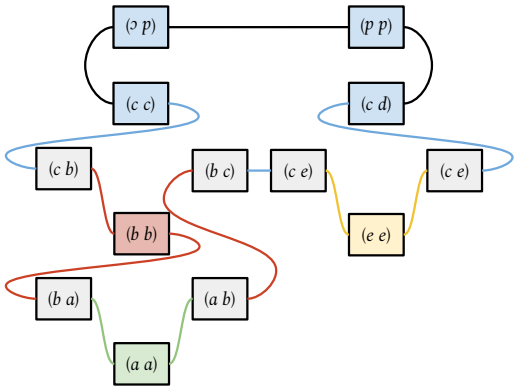


Figure 7.7: After adding the dashed edges in figure 7.5, the Euler tour stored in a linked list may look like this. In this figure, we color links in the list the same colors as the nodes "responsible" for adding those links in through calls to JOIN.

Algorithm 36 Computes locations at which to join for batch cut.

```
1:  procedure GetNextUnmarked(elements = {z_1, z_2, ..., z_k})
2:                                                  ▷ input is a set of skip list elements
3:      for i ∈ {1, ..., k} do in parallel
4:          next := z_i → twin → right
5:          if next → mark then
6:              z_i → next_edge := next
7:          else
8:              z_i → next_edge := null
9:                          ▷ Use list tail-finding on the linked lists induced by next_edge pointers.
                              Get an array last_marked such that last_marked[i] points to the last
                              node in z_i's linked list.
10:     last_marked := ListTailFind(elements)
11:     result := {}                                                    ▷ k-length array
12:     for i ∈ {1, ..., k} do in parallel
13:         result[i] := last_marked[i] → twin → right
14:     return result
```

Algorithm 37 Remove a batch of edges from Euler tour tree.

```
1:  procedure BatchCut({{u_1, v_1}, {u_2, v_2}, ..., {u_k, v_k}})
2:                                  ▷ input edges must be in graph and must have no duplicates
3:    directed_edges := {}                                              ▷ 2k-length array
4:    for i ∈ {1, ..., k} do in parallel
5:      directed_edges[2i − 1] := edges[(u_i, v_i)]
6:      directed_edges[2i] := edges[(v_i, u_i)]
7:    for i ∈ {1, ..., k} do in parallel
8:      edges → RemoveFromDict((u_i, v_i))
9:      edges → RemoveFromDict((v_i, u_i))
10:   join_lefts := {}                                                  ▷ 2k-length array
11:   for i ∈ {1, ..., 2k} do in parallel
12:     join_lefts[i] := directed_edges[i] → left
13:     directed_edges[i] → mark := true
14:   join_rights := GetNextUnmarked(directed_edges)
15:                                                  ▷ Cut edges out of tour
16:   for i ∈ {1, ..., 2k} do in parallel
17:     Split(directed_edges[i])
18:     pred := directed_edges[i] → Left
19:     if pred ≠ null then
20:       Split(pred)
21:                                                  ▷ Join tours back together
22:   for i ∈ {1, ..., 2k} do in parallel
23:     if not join_lefts[i] → mark then
24:       Join(join_lefts[i], join_rights[i])
25:   for i ∈ {1, ..., 2k} do in parallel
26:     DeleteNode(directed_edges[i])
```

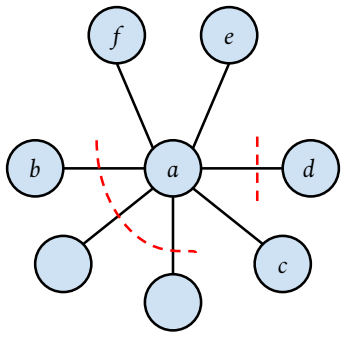


Figure 7.8: Batch cutting four edges. If we take an Euler tour counter-clockwise around this graph, this batch cuts may require us to join (f, a) to (a, c) in the tour.

tour on the graph goes counter-clockwise around the diagram, then we may need to join (c, a) to (a, e) in the tour as a result of cutting $\{a, d\}$ and join (f, a) all the way around to (a, c) as a result of the three contiguous cuts. How do we identify that we need to join (f, a) to (a, c)? We could mark edges that are going to be cut, then start from (f, a) and walk along "adjacent" edges incident to a using the TWIN pointers until we reach an edge that will not be cut. Then we would know to join (f, a) to that edge. However, the search for an unmarked edge will have poor depth if lots of edges will be cut.

To achieve low depth in this step, we use list tail-finding. Consider the linked lists induced by having each edge point at its adjacent edge if it is marked. Note that each linked list must terminate because traversing adjacent edges will eventually reach a loop edge of the form (v, v), which will certainly be unmarked. Then running list tail-finding on these linked lists finds for every edge the next unmarked edge as desired.

In Algorithm 37, we first fetch all the skip list nodes corresponding to the edges in lines 2-6. Then we invoke Algorithm 36 on line 14, which performs the list tail-finding described above. We cut out all the input edges on lines 15-20 and rejoin all the tours together on lines 21-24. In total these steps take $O(k \lg(1 + n/k))$ expected work and $O(\lg n)$ depth *whp*.

Augmentation. We build our augmented Euler tour trees over the concurrent augmented skip lists from Subsection 7.2.4 and achieve the same efficiency bounds. Recall that we have an associative and commutative function $f : D^2 \to D$ and assign values from D to vertices and edges of the forest. The goal is to compute f over subtrees of the represented forest.

Say we want to compute f over a vertex v's subtree relative to v's parent in the tree, p. Then if we look up the skip list elements corresponding to (p, v) and (v, p) in *edges*, the value of f over v's subtree is the result of applying f on the subsequence between (p, v) and (v, p). This may be done by calling BATCHQUERYVALUE with the elements corresponding to (p, v) and (v, p) in the underlying augmented skip list. The complexity for k such queries is $O(k \lg n)$ expected work and $O(\lg n)$ depth *whp*.

7.4.2 Implementation

We provide details about our implementation of Euler tour trees in Section 7.5.2.

7.5 Algorithm Implementation

7.5.1 Skip Lists

In our implementation of our skip lists, instead of representing an element of height h as h distinct nodes, we instead allocate an array holding h LEFT and RIGHT pointers. This avoids jumping around in memory when traversing up direct parents. In fact, we allocate an

array whose size is h rounded up to the next power of two. This decreases the number of distinct-sized arrays, which makes memory allocation easier when performing concurrent allocation. We also cap the height at 32, again for easier allocation. We set the probability of a node having a direct parent to be $p = 1/2$.

We also need to be careful about read-write reordering on architectures with relaxed memory consistency. For JOIN (Algorithm 26), if the reads from the searches in lines 4 to 5 are reordered before the write on line 3, then we can fail to find a parent link that should be added. Thus we disallow reads from being reordered before line 3.

For augmented skip lists, instead of keeping h NEEDS_UPDATE booleans for each element, we keep a single integer that saves the lowest level on which the element needs an update. This works because if a node needs updating, then all its direct ancestors need updating as well.

Another optimization saves a constant factor in the work for batch split. In BATCHUPDATEVALUES, we first walk up the skip list claiming nodes and then walk back down to update all the augmented values. We perform these two passes because in order to update a node's value, we need to know all of its childrens' values are already updated too, which is easier to coordinate when walking top-down through the list. However, in a batch split, after cutting up the list, no nodes on the bottom level share any ancestors. As a result, we can update all augmented values in a single pass walking up the list without even using CAS.

7.5.2 Euler Tour Trees

We implemented our parallel Euler tour tree algorithms, making several adjustments for performance and for ease of implementation. For simplicity, we use the unaugmented skip lists and do not support subtree queries.

To achieve good parallelism, we need to allocate and deallocate skip list nodes in parallel. We use lock-free concurrent fixed-size allocators that rely on both global and local pools. To reduce the number of fixed-size allocators used, we constrain the skip list heights and arrays as described in Section 7.5.1.

For the dictionary *edges*, we use the deterministic hash table dictionary from the Problem Based Benchmark Suite (PBBS) [324]. This hash table is based upon a phase-concurrent hash table developed by Shun and Blelloch [320]. As an additional storage optimization, for an edge $\{u, v\}$ where $u < v$, we only store (u, v) in our dictionary and use the TWIN pointer to look up (v, u).

Instead of performing a semisort when batch joining, we found it faster to use the parallel radix sort from PBBS.

For batch cut, we do not use list tail-finding because efficient list tail-finding is challenging to implement. Instead, we opt for a recursive batch cut algorithm. Recall why we used list tail-finding in Algorithm 37: we do not want to spend too much time walking

around adjacent edges to find one that is unmarked. In our recursive batch cut algorithm, we resolve the issue by randomly selecting a constant fraction of the edges from the input to ignore and not cut. Then if we walk around adjacent edges naively, the number of edges we need to walk around until we see an unmarked edge is constant in expectation and $O(\lg n)$ *whp*. Thus we can cut out the unignored edges quickly. Then we recurse on the ignored edges. This increases the depth of the implementation by a factor of $O(\lg k)$ but does not asymptotically affect the work.

7.6 Experiments

We run our experiments on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4 × 2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use Cilk Plus to express parallelism and are compiled with the g++ compiler (version 5.5.0). When running in parallel, we use the command `numactl -i all` to evenly distribute the allocated memory among the processors. On our figures, a thread count of 72(h) denotes using all 72 cores with hyper-threading, i.e. using 144 threads.

7.6.1 Unaugmented Skip Lists

We evaluate the performance of our skip lists (with the probability of a node having a direct parent set to $p = 1/2$) by comparing them against other sequence data structures. In particular, we compare against sequential skip lists, which are the same as our skip lists except that they do not use CAS to set pointers. In addition, for an element of height h, they allocate an array of exactly length h for holding pointers rather than an array of length $O(h)$ as our parallel skip list implementation does. We also implemented splay trees [329] and treaps [304].

So that we can compare against another parallel data structure, we implement parallel batch join and batch split operations on treaps. To batch join, we first ignore a constant fraction of the joins. If we imagine each join from treap T to treap S as a pointer from T to S, we get lists on the treaps. No list can be very long because of the ignored joins. We get parallelism by processing each list independently. If we store extra information on the treap nodes, we can walk along a list and perform its joins sequentially. Then we recursively process the previously ignored joins. For batch split, we semisort the splits keyed on the root of the treap to be split. This lets us find all splits that act on a particular treap. We process each treap independently. When performing multiple splits on a treap, we get parallelism by divide and conquer—we perform a random split and recursively split the resulting two treaps in parallel. The randomized efficiency bounds are $O(k \lg n)$ work for batch join, $O(k \lg n \lg k)$ work for batch split, and $O(\lg n \lg k)$ depth for both. In

Data structure	Batch size k	Operation	Number of threads								
			1	2	4	8	16	32	64	72	72(h)
Concurrent skip list	10^4	join	.0301	.0165	.0101	.00632	.00298	.00166	.000937	.000839	.000530
		split	.0331	.0173	.0113	.00579	.00298	.00154	.000865	.000775	.000542
	10^7	join	12.6	6.43	4.07	2.05	1.03	.528	.279	.267	.156
		split	10.4	5.34	3.34	1.86	.869	.426	.228	.214	.122

Table 7.1: Running time (in seconds) of our concurrent skip lists with $n = 10^8$.

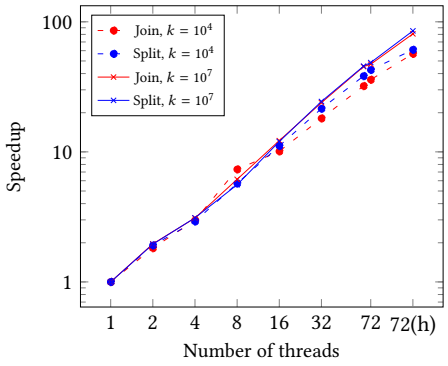


Figure 7.9: Speedup of our concurrent skip lists with $n = 10^8$.

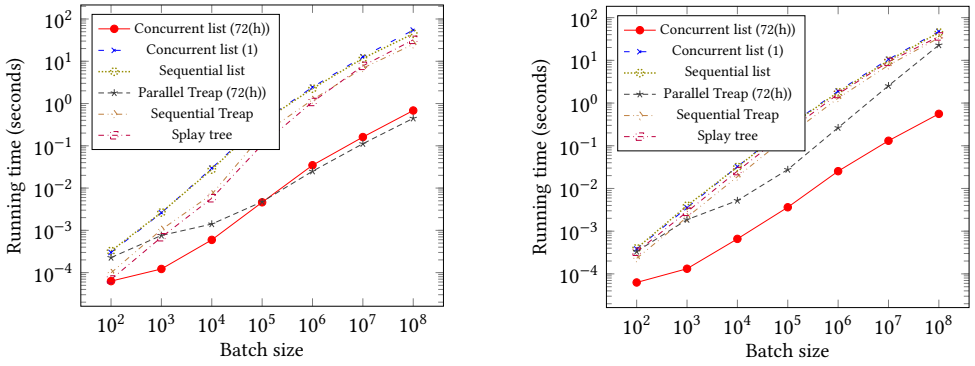
Data structure	Operation	Batch size k						
		10^2	10^3	10^4	10^5	10^6	10^7	$10^8 - 1$
Concurrent skip list (72(h))	join	.0000629	.000122	.000595	.00461	.0347	.161	.684
	split	.0000629	.000132	.000660	.00363	.0254	.131	.559
Concurrent skip list (1)	join	.000300	.00260	.0295	.376	2.44	12.8	54.7
	split	.000383	.00355	.0325	.281	1.90	10.6	47.6
Sequential skip list	join	.000324	.00265	.0275	.362	2.26	11.7	44.9
	split	.000396	.00359	.0319	.272	1.80	9.87	45.0
Parallel treap (72(h))	join	.000227	.000758	.00141	.00475	.0250	.112	.447
	split	.000335	.00186	.00521	.0277	.265	2.54	22.8
Sequential treap	join	.0000989	.00104	.00712	.183	1.23	6.86	25.2
	split	.000231	.00213	.0189	.168	1.30	7.69	33.5
Splay tree	join	.0000689	.000688	.00575	.106	1.09	7.79	32.1
	split	.000329	.00284	.0255	.215	1.63	9.21	36.3

Table 7.2: Running time (in seconds) of sequence data structures with $n = 10^8$ and varying batch size.

the future, we would like to further compare our skip lists against other parallel data structures, such as the (a, b)-trees of Akhremtsev and Sanders [15].

For an experiment, we take $n = 10^8$ elements and fix a batch size k. We set up a trial by joining all the elements in a chain, and then we time how long it takes to split and rejoin the sequence at k pseudorandomly sampled locations. We report the median time over three trials. As an artifact of this setup, the splay tree has an advantage on joining small batches after splitting due to how splay trees exploit locality.

Table 7.1 and Figure 7.9 illustrate that our skip list implementation running on 72



(a) Join **(b)** Split

Figure 7.10: Running time of sequence data structures operations with varying batch size.

Data structure	Batch size k	Operation	Number of threads								
			1	2	4	8	16	32	64	72	72(h)
Parallel skip list	10^4	join	.0908	.0457	.0251	.0164	.00871	.00487	.00292	.00282	.00227
		split	.0546	.0283	.0195	.0134	.00561	.00306	.00177	.00164	.00113
	10^7	join	24.9	12.7	9.39	4.41	2.10	1.09	.614	.61	.374
		split	19.7	10.1	7.49	3.48	1.61	.810	.422	.398	.253

Table 7.3: Running time (in seconds) of our parallel augmented skip lists with $n = 10^8$.

cores with hyper-threading demonstrates over 80× speedup relative to the implementation running on a single thread for $k = 10^7$ and over 55× speedup for $k = 10^4$. We compare our skip list to our other sequence data structures in Table 7.2 and Figure 7.10 . Our implementation of parallel batch join on treaps is 1.4× faster than our batch join on skip lists on the largest batch sizes, but, as seen in Figure 7.10, the parallel batch split on treaps is much slower due to lots of overhead work. Moreover, through parallelism, our data structure is significantly faster than all the sequential algorithms at all batch sizes. When used sequentially, our data structure behaves similarly to a traditional sequential skip list, suggesting that using CAS does not significantly degrade the performance of a skip list.

7.6.2 Augmented Skip Lists

We compare the performance of our batch-parallel augmented skip lists against a sequential augmented skip list. Besides not using CAS, the sequential skip list updates augmented values after every join and split. This achieves only an $O(k \lg n)$ work bound for k operations. Our experiment is the same as in Subsection 7.6.1.

Table 7.3 and Figure 7.11 show that when running our augmented skip list with a random batch of size $k = 10^7$ on 72 cores with hyper-threading, we see a speedup of 67× for joins and 78× for splits. For $k = 10^4$, we found a speedup of 33× for joins and 48× for splits. The running times are a factor of two worse than the times for the unaugmented

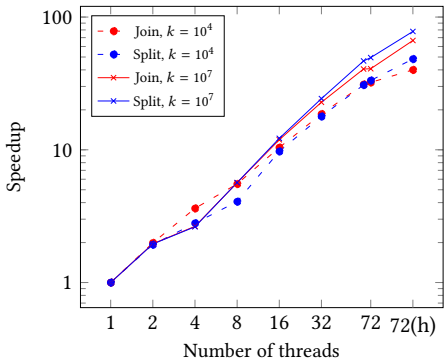
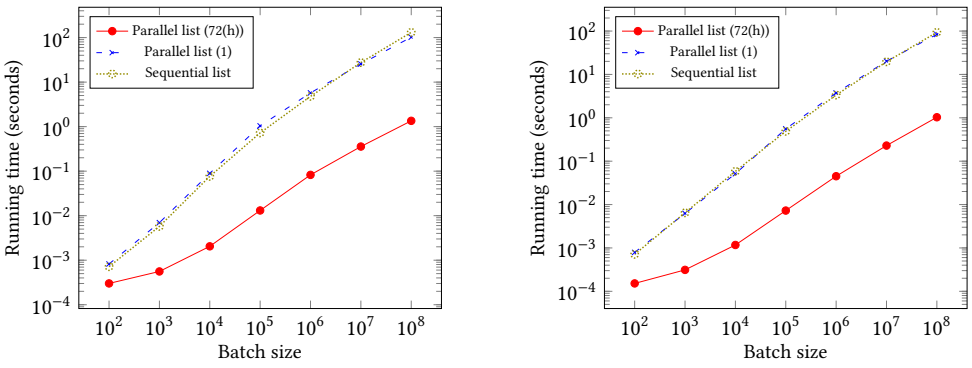


Figure 7.11: Speedup of our parallel augmented skip lists with $n = 10^8$.

Data structure	Operation	Batch size k						
		10^2	10^3	10^4	10^5	10^6	10^7	$10^8 - 1$
Parallel skip list (72(h))	join	.000301	.000559	.00205	.0131	.0825	.357	1.35
	split	.000152	.000312	.00117	.00727	.0450	.229	1.03
Parallel skip list (1)	join	.000823	.00697	.0893	1.04	5.73	25.5	102
	split	.00079	.00625	.0514	.557	3.74	20.2	83.7
Sequential skip list	join	.000716	.00575	.0764	.724	4.79	27.5	131
	split	.000712	.00647	.0583	.489	3.35	19.6	93.3

Table 7.4: Running time (in seconds) of augmented skip lists with $n = 10^8$ on random batches of varying size.



(a) Join **(b)** Split

Figure 7.12: Running time of augmented sequence data structure operations with $n = 10^8$ on random batches of varying size.

Data structure	Operation	Batch size k						
		10^2	10^3	10^4	10^5	10^6	10^7	$10^8 - 1$
Parallel skip list (72(h))	split	.000111	.000192	.000272	.000596	.00281	.0259	.252
Parallel skip list (1)	split	.0000350	.000148	.00119	.0115	.119	1.20	11.9
Sequential skip list	split	.000296	.00258	.0219	.210	2.11	21.5	205

Table 7.5: Running time (in seconds) of splitting augmented skip lists with $n = 10^8$ as batch size varies with splits taking single elements off the end of the list. This is a difficult test case for standard augmented skip lists.

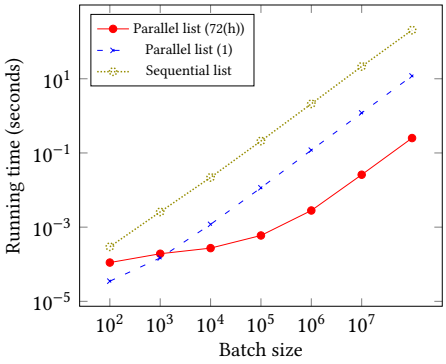


Figure 7.13: Running time of splitting augmented skip lists with $n = 10^8$ as batch size varies with splits taking off single elements off the end of the list.

skip list of Subsection 7.6.1, which is expected due to the extra passes through the skip list to update augmented values. Moreover, the batch-parallel skip list hugely outperforms single-threaded skip lists on all tested batch sizes, as seen in Table 7.4 and Figure 7.12.

To more prominently display the work savings that batching provides, we try a different test case in which a batch of k splits, rather than being chosen at random, consists of splitting at the last k elements of the sequence in right-to-left order. This is particularly bad for the sequential skip list because after every split, it walks to the top of the skip list to update augmented values. In comparison, when processing the splits as a batch, we update the augmented values in only one pass. Table 7.5 and Figure 7.13 show that, as expected, even on a single thread, our skip list is significantly faster than the standard sequential one in this adversarial experiment.

7.6.3 Euler Tour Trees

We compare against sequential dynamic trees data structures. Using the sequential skip list and splay trees from Subsection 7.6.1, we build traditional Euler tour trees. We also compare to ST-trees built on splay trees [328, 329]. They achieve $O(\lg n)$ amortized work links and cuts. Though conceptually more complicated than Euler tour trees, ST-trees are a more streamlined data structure that do not require allocation beyond initialization and

Data structure	Graph	Batch size k	Operation	Number of threads								
				1	2	4	8	16	32	64	72	72(h)
Parallel ETT	Path graph	10^4	link	.175	.109	.0559	.0172	.00988	.00595	.00374	.00345	.0029
			cut	.185	.129	.0641	.0270	.0139	.00743	.00417	.00378	.00267
		10^6	link	7.25	5.10	2.53	1.10	.548	.279	.143	.131	.0813
			cut	8.74	6.48	3.22	1.36	.672	.348	.177	.159	.0980
	Random recursive tree	10^4	link	.111	.0579	.0266	.0162	.00849	.00524	.00334	.00316	.00278
			cut	.149	.0759	.0358	.0192	.00962	.00525	.00301	.00275	.00199
		10^6	link	8.77	6.31	3.20	1.34	.684	.344	.182	.161	.0972
			cut	7.87	5.87	2.96	1.26	.628	.323	.171	.147	.0882
	Star graph	10^4	link	.0154	.0147	.00693	.00533	.00344	.00249	.00203	.00191	.00198
			cut	.0170	.0112	.00733	.00442	.00247	.00136	.00102	.00102	.000922
		10^6	link	3.49	2.05	1.01	.454	.237	.125	.0681	.0618	.0408
			cut	2.60	1.56	.740	.339	.171	.0883	.0467	.0419	.0271

Table 7.6: Running time (in seconds) of our batch-parallel Euler tour tree on various graphs with $n = 10^7$.

Data structure	Graph	
Static connectivity (72(h))	Path graph	.576
	Random recursive tree	.174
	Star graph	.113

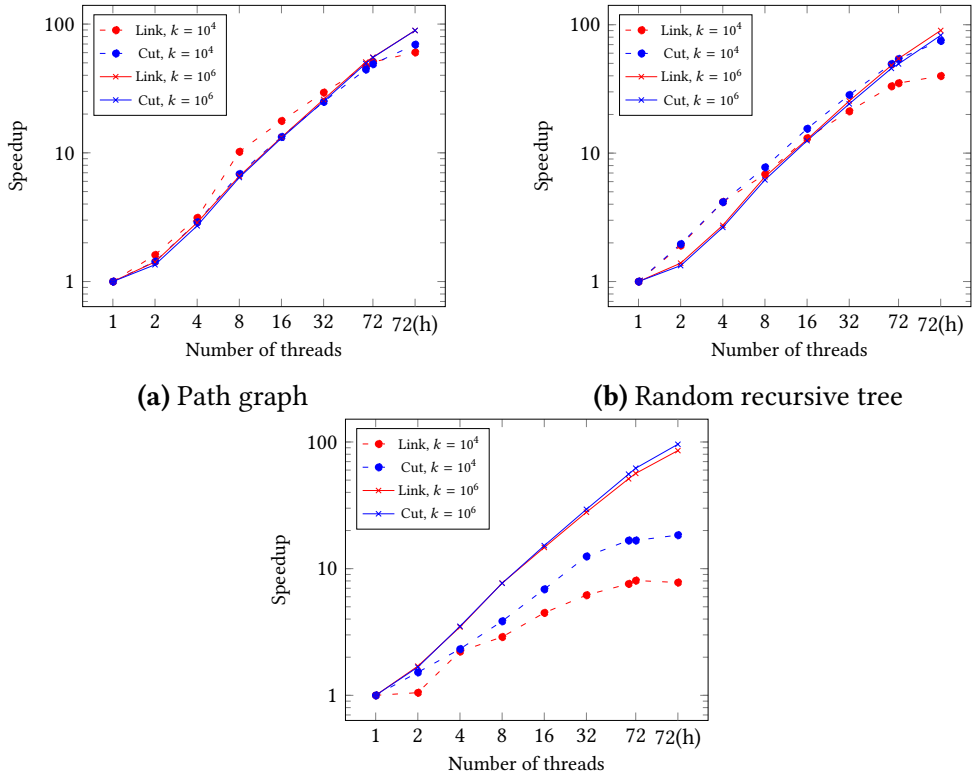
Table 7.7: Running time (in seconds) of static connectivity on various graphs with $n = 10^7$

do not require dictionary lookups. We wrote all of these implementations. In future work, we would like to compare against parallel data structures such as that of Acar et al. [4]

Because one of the important uses of Euler tour trees is to answer connectivity queries, we also compare with statically computing the connected components of the graph. We use the work-efficient parallel connectivity algorithm designed and implemented by Shun et al. [321] (this implementation is an earlier version of the implementation presented in Chapter 5). We optimistically measure the execution time of the implementation based only on the execution time of the connectivity algorithm; we do not include the time taken to maintain the graph itself, which is non-trivial because the adjacency array graph representation used in their implementation does not support edge insertion or deletion easily.

For our experiment, we fix a tree. We set up a trial by adding all the edges of the tree in pseudorandom order to our data stucture. Then we time how long it takes to cut and relink the forest at k pseudorandomly sampled edges. We report median times over three trials. Again, note that our experimental setup may give the splay tree data structures an advantage on linking small batches after cutting due to how splay trees exploit locality. We experimented on three trees, all with $n = 10^7$ vertices: a path graph, a star graph, and a random recursive tree. To form a random recursive tree over n vertices, for each $1 < i \leq n$, draw j uniformly at random from $\{1, 2, \ldots, i-1\}$ and add the edge $\{j, i\}$.

Table 7.6 and Figure 7.14 display the speedup of our parallel Euler tour tree algorithms with a batch sizes of $k = 10^4$ and $k = 10^6$. When running on 72 cores with hyper-threading,



(a) Path graph

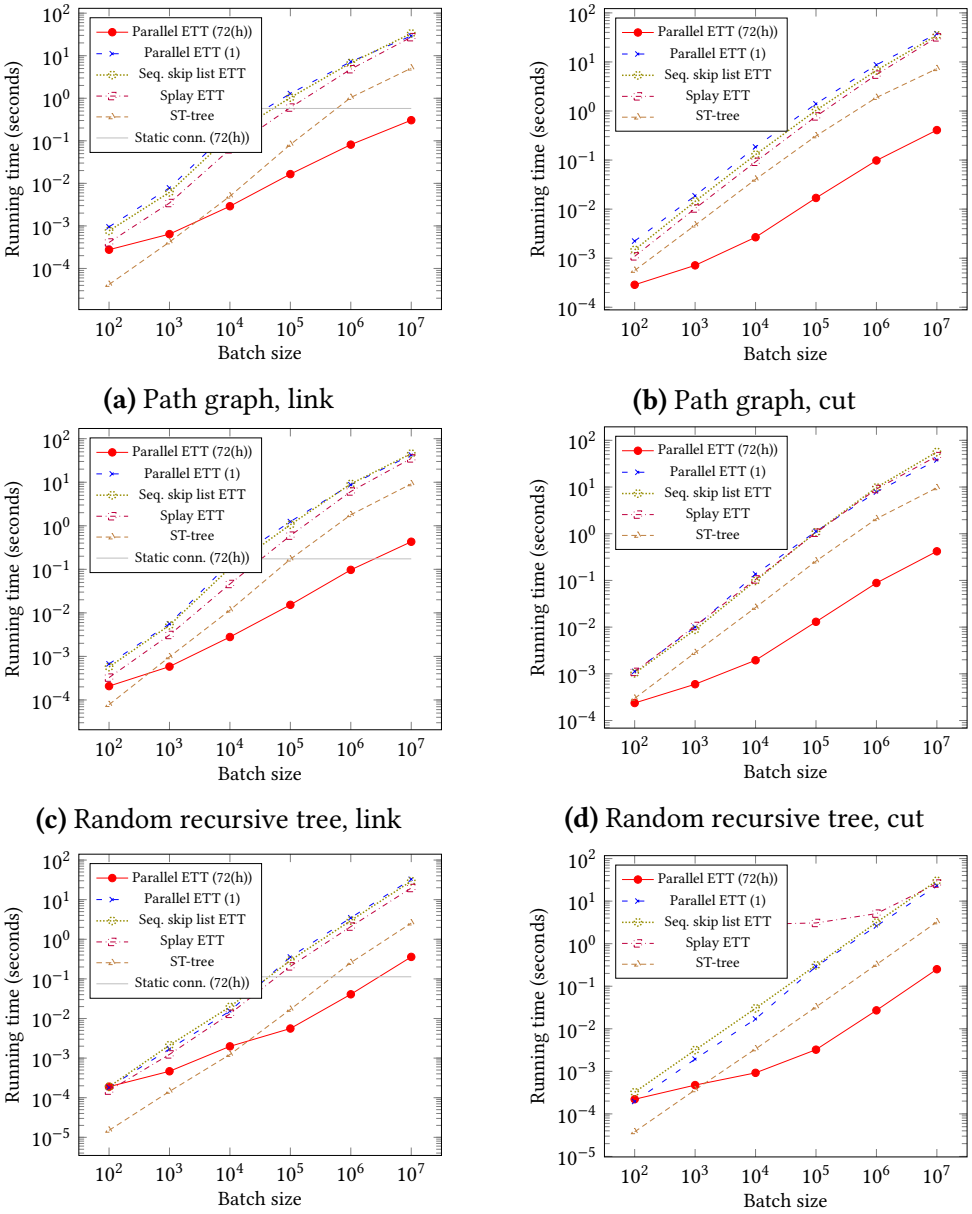
(b) Random recursive tree

(c) Star graph

Figure 7.14: Speedup of our parallel Euler tour trees on a various forests of size $n = 10^7$.

we get good speedup ranging from 82× to 96× for $k = 10^6$ across all tested graphs. For $k = 10^4$, we found speedup ranging from 7.5× to 75× where the worst speedup was on the star graph. On the star graph, the single-threaded running time is already fast for batch size $k = 10^4$, so there is not as much room for speedup.

In Table 7.8 and Figure 7.15, we show the running times of our Euler tour tree along with the times for sequential dynamic trees data structures. We also show in Table 7.7 the time to run the static connectivity algorithm on the full graphs for comparison. On large batch sizes, parallelism beats all the sequential data structures, as expected. Though ST-trees are faster than Euler tour trees sequentially and are unusually fast on the star graph due to them performing well on graphs with small diameter, our parallel Euler tour tree eventually outspeeds ST-trees on large batches even on the star graph. (As an artifact of our testing setup, the splay-tree-based Euler tour tree performs poorly on the star graph. The access pattern on the splay trees when constructing the graph leads to a very deep splay tree, so the first few cut operations after the graph construction setup are expensive.) In addition, the performance of our Euler tour tree running on a single thread is similar to that of conventional sequential Euler tour trees. We also see that the time to update our



(a) Path graph, link

(b) Path graph, cut

(c) Random recursive tree, link

(d) Random recursive tree, cut

(e) Star graph, link

(f) Star graph, cut

Figure 7.15: Running time of dynamic trees data structure operations on trees of size $n = 10^7$ with varying batch size.

Graph	Data structure	Operation	Batch size k					
			10^2	10^3	10^4	10^5	10^6	10^7-1
Path graph	Parallel ETT (72(h))	link	.000276	.000642	.00290	.0165	.0813	.305
		cut	.000297	.000714	.00267	.0169	.0980	.408
	Parallel ETT (1)	link	.000950	.00784	.175	1.29	7.25	29.2
		cut	.00223	.0187	.185	1.40	8.74	37.6
	Seq. skip list ETT	link	.000767	.00617	.131	1.04	6.77	33.4
		cut	.00148	.0144	.129	1.03	6.73	35.0
	Splay ETT	link	.000408	.00345	.0631	.605	4.82	27.3
		cut	.00109	.0103	.0922	.484	5.51	31.7
	ST-tree	link	.0000420	.000418	.00499	.0818	1.04	5.09
		cut	.000562	.00471	.0405	.310	1.88	7.27
Random recursive tree	Parallel ETT (72(h))	link	.000208	.000584	.00279	.0153	.0972	.430
		cut	.000237	.000599	.00195	.013	.0882	.420
	Parallel ETT (1)	link	.000674	.00560	.144	1.25	8.77	42.3
		cut	.00113	.0100	.137	1.12	7.82	37.9
	Seq. skip list ETT	link	.000572	.00520	.107	1.06	9.17	45.5
		cut	.00110	.0106	.105	1.05	9.06	45.9
	Splay ETT	link	.000326	.00311	.0461	.594	6.14	35.3
		cut	.000942	.00890	.0856	.839	7.20	39.4
	ST-tree	link	.0000780	.000975	.0115	.171	1.79	9.17
		cut	.000298	.00284	.0263	.259	2.09	9.62
Star graph	Parallel ETT (72(h))	link	.000190	.000465	.00198	.00558	.0408	.359
		cut	.000221	.000475	.000922	.00323	.0271	.251
	Parallel ETT (1)	link	.000182	.00170	.0154	.357	3.49	32.7
		cut	.000196	.00194	.0170	.289	2.60	23.1
	Seq. skip list ETT	link	.000189	.00212	.0197	.293	2.97	29.4
		cut	.000323	.00317	.0303	.311	3.15	30.1
	Splay ETT	link	.000151	.00124	.0131	.207	2.09	19.7
		cut	2.85	2.85	2.86	3.07	5.09	25.9
	ST-tree	link	.0000150	.000144	.00123	.0169	.257	2.56
		cut	.0000379	.000367	.00336	.0325	.323	3.24

Table 7.8: Running time (in seconds) of dynamic trees data structures on various graphs with $n = 10^7$.

Euler tour tree is much less than the time to statically compute connectivity for all but the largest batch sizes.

7.7 Discussion

We showed that skip lists are a simple, fast data structure for parallel joining and splitting of sequences and that we can use these skip lists to build a batch-parallel Euler tour tree. Both of these data structures achieve strong theoretical bounds on their work and depth in the BF model and achieve good performance in practice.

8 Parallel Batch-Dynamic Connectivity

8.1 Introduction

Computing the connected components of a graph is a fundamental problem that has been studied in many different models of computation [347, 316, 290, 175, 14, 25]. The ***connectivity problem*** takes as input an undirected graph G and assigns labels to vertices so that two vertices have the same label if and only if they are in the same connected component. The ***dynamic connectivity problem*** requires maintaining a data structure over an n vertex undirected graph that supports operations which query whether two vertices are in the same connected component, or inserts and deletions of edges. Despite the large body of work on the dynamic connectivity problem over the past two decades [173, 135, 175, 349, 350, 172, 374, 193, 197, 179, 256, 375], little is known about batch-dynamic connectivity algorithms that process *batches of queries and updates*, either sequentially or in parallel.

Understanding the connectivity structure of graphs is of significant practical interest, for example, due to its use as a primitive for clustering the vertices of a graph [296]. Due to the importance of connectivity there are several implementations of parallel batch-dynamic connectivity algorithms [233, 184, 370, 185, 309, 359]. In the worst case, however, these algorithms may recompute the connected components of the entire graph even for very small batches. Since this requires $O(m + n)$ work, it makes the worst-case performance of the algorithms no better than running a static parallel algorithm. On the theoretical side, existing batch-dynamic efficient connectivity algorithms have only been designed for restricted settings, e.g., in the incremental setting when all updates are edge insertions [327], or when the underlying graph is a forest [289, 4, 353]. Therefore, two important questions are:

1. *Is there a batch-dynamic connectivity algorithm that is asymptotically faster than existing dynamic connectivity algorithms for large enough batches of insertions, deletions and queries?*
2. *Can the batch-dynamic connectivity algorithm be parallelized to achieve low worst-case depth?*

This chapter presents an algorithm that answers both of these questions affirmatively. To simplify exposition and present the main ideas, this chapter first gives a less efficient version of the algorithm that runs in $O(\lg^4 n)$ depth *whp* and performs $O(\lg^2 n)$ expected amortized work per update, making it work-efficient with respect to the classic HDT algorithm. Next, we describe the improved algorithm which achieves an improved work

bound that is asymptotically faster than the HDT algorithm for sufficiently large batch sizes, and runs in $O(\lg^3 n)$ depth *whp* on the CRCW PRAM.[1] We note that our depth bounds hold even when processing the updates one a time, ignoring batching. Our improved work bounds are derived by a novel analysis of the work performed by the algorithm over all batches of deletions.

Our main contribution in this chapter is summarized by the following theorem:

Theorem 9. *There is a parallel batch-dynamic data structure which, given batches of edge insertions, deletions, and connectivity queries processes all updates in* $O\left(\lg n \lg\left(1+\frac{n}{\Delta}\right)\right)$ *expected amortized work per edge insertion or deletion where* Δ *is the average batch size of deletion. The cost of connectivity queries is* $O(k \lg(1+n/k))$ *expected work and* $O(\lg n)$ *depth whp for a batch of* k *queries. The depth to process a batch of edge insertions and deletions is* $O(\lg n + D_{\mathsf{SF}})$ *whp and* $O(\lg^2 n(\lg n + D_{\mathsf{SF}}))$ *whp respectively, where* D_{SF} *is the depth of a linear expected work spanning forest algorithm in either the CRCW PRAM or the* BF *model.*

8.2 Preliminaries

Model and Depth Analysis. In this chapter we consider our algorithms in both the binary-forking (BF) model and the CRCW PRAM model. The work bounds of our algorithms are identical in both models, but the *depth* in the BF model can be an $O(\lg n)$ factor higher than the CRCW PRAM depth, and so we explicitly distinguish between the two. The primary reason for the difference is due to the depth of computing the spanning forest of a graph: the best depth for this problem in the CRCW PRAM is $O(\lg n)$ using Gazit's algorithm [148], but this algorithm (and to the best of our knowledge, all other spanning forest algorithms) require $O(\lg^2 n)$ in the BF model.[2] Therefore, whenever the depth can vary due to the use of spanning forest, we parametrize the depth of the algorithm by D_{SF} which is the best depth of a linear expected work connectivity algorithm in the chosen model.

8.2.1 Lemmas

We start with a few simple lemmas that help analyze the work of the dynamic algorithms presented in this chapter.

[1] The depth bound of this algorithm on the BF model depends on the depth of a spanning forest algorithm; to the best of our knowledge, all existing spanning forest algorithms in the BF model run in $O(\lg^2 n)$ depth.

[2] Designing an optimal $\Theta(\lg n)$ depth connectivity or spanning forest algorithm in the BF is an interesting open problem, even if we allow for work-inefficient algorithms.

Lemma 4. *Let $n_1, n_2, \ldots, n_c$ and $k_1, k_2, \ldots, k_c$ be sequences of non-negative integers such that $\sum k_i = k$, and $\sum n_i = n$. Then*

$$\sum_{i=1}^{c} k_i \lg\left(1 + \frac{n_i}{k_i}\right) \leq k \lg\left(1 + \frac{n}{k}\right). \tag{8.1}$$

Lemma 5. *For any non-negative integers n and r,*

$$\sum_{w=0}^{r} 2^w \lg\left(1 + \frac{n}{2^w}\right) = O\left(2^r \lg\left(1 + \frac{n}{2^r}\right)\right). \tag{8.2}$$

Lemma 6. *For any $n \geq 1$, the function $x \lg\left(1 + \frac{n}{x}\right)$ is strictly increasing with respect to x for $x \geq 1$.*

The proofs of these lemmas are elementary, and can be found in the full version of the paper that this chapter is based on [8].

8.2.2 Data Structures

Next, we describe a simple adjacency-list like data structure that efficiently supports insertion and deletion of arbitrary edges, and quickly fetching a batch of l edges. This is the data structure that we use to store adjacency lists of vertices at each level. Note that we actually store two adjacency lists, one for tree edges, and one for non-tree edges. The adjacency list data structure supports the following operations:

- **InsertEdges**($\{e_1, \ldots, e_l\}$): Insert a batch of edges adjacent to this vertex.
- **DeleteEdges**($\{e_1, \ldots, e_l\}$): Delete a batch of edges adjacent to this vertex.
- **FetchEdges**(l): Return a set of l arbitrary edges adjacent to this vertex.

We now show how to implement a data structure that gives us the following bounds:

Lemma 7. *InsertEdges, DeleteEdges, and FetchEdges can be implemented in $O(1)$ amortized work per edge and in $O(\lg n)$ depth.*

Proof. For a given vertex, the data structure stores a list of pointers to each adjacent edge in a resizable array. Each edge correspondingly stores its positions in the adjacency arrays of its two endpoints. Since each vertex can have at most $O(n)$ edges adjacent to it, the adjacency arrays are of size at most $O(n)$.

Insertions are easily handled by inserting the batch onto the end of the array, and resizing if necessary. This costs $O(1)$ amortized work per edge and $O(\lg n)$ depth. To fetch l elements, we simply return the first l elements of the array, which takes $O(1)$ work per edge and $O(\lg n)$ depth.

To delete a batch of l edges, the algorithm first determines which of the edges to be deleted are contained within the final l elements of the array. It then compacts the final l elements of the array, removing those edges. Compaction costs $O(l)$ work and $O(\lg n)$ depth. The algorithm then considers the remaining l' edges to be deleted, and in parallel, swaps these elements with the final l' elements of the array. The final l' elements in the array can then be safely removed. Note that any operation that moves an element in the array also updates the corresponding position value stored in the edge. Swapping and deleting can be implemented in $O(l')$ work and $(\lg n)$ depth, and hence all operations cost $O(1)$ amortized work per edge and $O(\lg n)$ depth. □

8.2.3 Tree Operations

We use the following results from Chapter 7:

Theorem 10. *A batch of k links, k cuts, k connectivity queries, or k representative queries over an n-vertex forest can be processed in $O(k \lg(1 + n/k))$ expected work and $O(\lg n)$ depth with high probability.*

We also make use of additional tree operations that are needed to efficiently implement the batch-dynamic algorithms in this chapter, which we discuss next.

Retrieving and Pushing Down Edges. The batch-parallel ET-trees used in this chapter augment each node in the tree with two values indicating the number of tree and non-tree edges whose level is equal to the level of the forest currently stored in that subtree. The augmentation is necessary for efficiently fetching the tree edges that need to be pushed down before searching the data structure, and for fetching a subset of non-tree edges in a tree.

We extend the batch-dynamic trees interface described earlier with operations which enable efficiently retrieving, removing and pushing down batches of tree or non-tree edges.

These primitives are all similar and can be implemented as follows. We first describe the primitives which fetch and remove a set of l tree (or non-tree) edges. The algorithm starts by finding a set of vertices containing l edges. To do this we perform a binary search on the skip-list in order to find the first node that has augmented value greater than l. The idea is to sequentially walk at the highest level, summing the augmented values of nodes we encounter and marking them, until the first node that we hit whose augmented value makes the counter larger than l, or we return to v. In the former case, we descend a level using this node's downwards pointer, and repeat, until we reach a level 0 node. We also keep a counter, ctr, indicating the number of tree (non-tree) edges to take

from the rightmost marked node at level 0. Otherwise, all nodes at the topmost level are marked. The last step of the algorithm is to find all descendants of marked nodes that have a non-zero number of tree (non-tree) edges, and return all tree (non-tree) edges incident on them. The only exception is the rightmost marked node, from which we only take ctr many tree (non-tree) edges

Insertions are handled by first inserting the edges into the adjacency list data structure. We then update the augmented values in the ET-tree using the primitive from Tseng et al. [353].

We now argue that these implementations achieves good work and depth bounds.

Lemma 8. *Given some vertex, v in a batch-parallel ET-tree, we can fetch the first l tree (or non-tree) edges referenced by the augmented values in the tree in $O\left(l \lg\left(1 + \frac{n_c}{l}\right)\right)$ work and $O(\lg n)$ depth* whp *where n_c is the number of vertices in the ET-tree at the current level. Furthermore, removing the edges can be done in the same bounds.*

Proof. Standard proofs about skip-lists shows that the number of nodes traversed in the binary search is $O(\lg n)$ *whp* (see Chapter 7). We can fetch l edges from each vertex's adjacency list data structure in $O(l)$ amortized work and $O(\lg n)$ depth by Lemma 7. The total work is therefore $O\left(l \lg\left(1 + \frac{n_c}{l}\right)\right)$ in expectation, and the depth is $O(\lg n)$ *whp* since the depth of the adjacency list access is an additive increase of $O(\lg n)$. Observe that removing the edges can be done in the same bounds since updating the augmented values after deleting the edges costs $O\left(l \lg\left(1 + \frac{n_c}{l}\right)\right)$ expected work. □

Lemma 9. *Decreasing the level of l tree (or non-tree) edges in a batch-parallel ET-tree can be performed in $O\left(l \lg\left(1 + \frac{n_c}{l}\right)\right)$ expected work and $O(\lg n)$ depth* whp *where n_c is the number of nodes in the ET-tree at the current level.*

Proof. The proof is identical to the proof of Lemma 8. The only difference is that the augmented values of the nodes that receive an edge must be updated after insertion which costs at most $O\left(l \lg\left(1 + \frac{n_c}{l}\right)\right)$ in expectation. Note that since the forest on the lower level is a subgraph of the tree at the current level, it has size at most n_c, proving the bounds. □

8.3 The Holm, de Lichtenberg, and Thorup Algorithm

Our parallel algorithm is based on the sequential algorithm of Holm, de Lichtenberg, and Thorup [175], which we refer to as the HDT algorithm. The HDT algorithm assigns to each edge in the graph, an integer *level* from 1 to $\lg n$. The levels correspond to sequence of subgraphs $G_1 \subset G_2 \subset \ldots \subset G_{\lg n} = G$, such that G_i contains all edges with level at most i. The algorithm also maintains a spanning forest F_i of each G_i such that $F_i \subset F_2 \subset \ldots \subset F_{\lg n}$. Each forest is maintained using a set of augmented ET-trees which we describe shortly. Throughout the algorithm, the following invariants are maintained.

Invariant 1. *$\forall i = 1 \ldots \lg n$, the connected components of G_i have size at most 2^i.*

Invariant 2. *$F_{\lg n}$ is a minimum spanning forest where the weight of each edge is its level.*

Connectivity Queries. To perform a connectivity query in G, it suffices to query $F_{\lg n}$, which takes $O(\lg n)$ time by querying for the root of each Euler tour tree and returning whether the roots are equal. We note that in [175], a query time of $O(\lg n / \lg \lg n)$ is achieved by storing the Euler tour of $F_{\lg n}$ in a B-tree with branching factor $\lg n$.

Inserting an Edge. An edge insertion is handled by assigning the edge to level $\lg n$. If the edge connects two currently disconnected components, then it is added to $F_{\lg n}$.

Deleting an Edge. Deletion is the most interesting part of the algorithm. If the deleted edge is not in the spanning forest $F_{\lg n}$, the algorithm removes the edge and does nothing to $F_{\lg n}$ as the connectivity structure of the graph is unchanged. Otherwise, the component containing the edge is split into two. The goal is to find a ***replacement edge***, that is, an edge crossing the split component.

If the deleted edge had level i, then the *smaller* of the two resulting components is searched starting at level i in order to locate a replacement edge. Before searching this component, all tree edges whose level is equal to i have their level decremented by one. As the smaller of the split components at level i has size $\leq 2^{i-1}$, pushing the entire component to level $i-1$ does not violate Invariant 1. Next, the non-tree edges at level i are considered one at a time as possible replacement edges. Each time the algorithm examines an edge that is not a replacement edge, it decreases the level of the edge by one. If no replacement is found, it moves up to the next level and repeats. Note that because the algorithm first pushes all tree edges to level $i-1$, any subsequent non-tree edges that may be pushed from level i to level $i-1$ will not violate Invariant 2.

Implementation and Cost. To efficiently search for replacement edges, the ET-trees are augmented with two additional pieces of information. The first augmentation is to maintain the number of non-tree edges whose level equals the level of the tree. The second augmentation maintains the number of tree-edges whose level is equal to the level of the tree.

Using these augmentations, each successive non-tree edge (or tree edge) whose level is equal to the level of the tree can be found in $O(\lg n)$ time. Furthermore, checking whether the edge is a replacement edge can be done in $O(\lg n)$ time. Lastly, the cost of pushing an edge that is not a replacement edge to the lower level is $O(\lg n)$, since it corresponds to inserting the edge into an adjacency structure and updating the augmented values. Since each edge can be processed at most once per level, paying a cost of $O(\lg n)$, and there are $\lg n$ levels, the overall amortized cost per edge is $O(\lg^2 n)$.

8.4 A Simple Parallel Batch-Dynamic Algorithm

In this section, we give a simple parallel batch-dynamic connectivity algorithm based on the HDT algorithm. The underlying invariants maintained by our parallel algorithm are identical to the sequential HDT algorithm: we maintain $\lg n$ levels of spanning forests subject to Invariants 1 and 2. The main challenge, and where our algorithm departs from the HDT algorithm is in how we search for replacement edges in parallel, and how we search multiple components in parallel. We show by a charging argument that this parallel algorithm is work-efficient with respect to the HDT algorithm—it performs $O(\lg^2 n)$ amortized work per edge insertion or deletion. Furthermore, we show that the depth of this algorithm is $O(\lg^4 n)$. Although these bounds are subsumed by the improved parallel algorithm we describe in Section 8.5, the parallel algorithm in this section is useful to illustrate the main ideas in this chapter.

Data Structures. Each spanning forest, F_i, is represented using a set of parallel batch-dynamic ETTs (see Chapter 7). We represent the edges of the graph in a parallel dictionary E_D for convenience (see Chapter 2), which can be done within the required work and depth bounds. We also store an adjacency array, $A_i[u]$, at each level i, and for each vertex u to store the tree and non-tree edges incident on u with level i. Note that tree and non-tree edges are stored separately so that they can be accessed separately.

8.4.1 Connectivity Queries

As in the sequential algorithm, a connectivity query can be answered by simply performing a query on $F_{\lg n}$. Algorithm 38 gives pseudocode for the batch connectivity algorithm. The bound we achieve follows from the batch bounds on batch-parallel ET-trees.

Algorithm 38 The batch query algorithm

```
1: procedure BatchQuery({(u_1, v_1), (u_2, v_2), ..., (u_k, v_k)})
2:     return F_{lg n}.BatchQuery({(u_1, v_1), (u_2, v_2), ..., (u_k, v_k)})
```

Theorem 11. *A batch of k connectivity queries can be processed in $O\left(k \lg\left(1 + \frac{n}{k}\right)\right)$ expected work and $O(\lg n)$ depth whp.*

Proof. Follows from the bounds for connectivity queries over ETTs obtained in Chapter 7. □

8.4.2 Inserting Batches of Edges

To perform a batch insertion, we first determine a set of edges in the batch that increase the connectivity of the graph. To do so, we treat each current connected component of

the graph as a vertex, and build a spanning forest of the edges being inserted over this contracted graph. The edges in the resulting spanning forest are then inserted into the topmost level in parallel.

Algorithm 39 The batch insertion algorithm

```
1: procedure BatchInsert( U = {(u_1, v_1), ..., (u_k, v_k)} )
2:     For all e_i ∈ U, set l(e_i) := lg n in parallel
3:     Update A_{lg n}[u] for edges incident on u
4:     R := {(F_{lg n}.FindRepr(u), F_{lg n}.FindRepr(u)) | (u, v) ∈ U}
5:     T' := SpanningForest(R)
6:     T := edges in U corresponding to T'
7:     Promote edges in T to tree edges
8:     F_{lg n}.BatchInsert(T)
```

Algorithm 39 gives pseudocode for the batch insertion algorithm. We assume that the edges given as input in U are not present in the graph. Each vertex u that receives an updated edge inserts its edges into $A_{\lg n}[u]$ (Line 3). This step can be implemented by first running a semisort to collect all edges incident on u.

The last step is to insert edges that increase the connectivity of the graph as tree edges (Lines 4–8). The algorithm starts by computing the representatives for each edge (Line 4). The output is an array of edges, R, which maps each original (u, v) edge in U to $(\textsc{FindRepr}(u), \textsc{FindRepr}(v))$ (note that these calls can be batched using BatchFindRepr). Next, it computes a spanning forest over the tree edges (Line 5). Finally, the algorithm promotes the corresponding edges in U to tree edges. This step is done by updating the appropriate adjacency lists and inserting them into $F_{\lg n}$ (Lines 7–8).

Theorem 12. *A batch of k edge insertions can be processed in $O\left(k \lg\left(1 + \frac{n}{k}\right)\right)$ expected work and $O(\lg n + D_{\mathsf{SF}})$ depth whp.*

Proof. Lines 2–3 cost $O(k)$ work and $O(\lg k)$ depth *whp* using our bounds for updating A (see Lemma 7). The find representative queries (Line 4) can be implemented using a BatchFindRepr call, which costs $O\left(k \lg\left(1 + \frac{n}{k}\right)\right)$ expected work and $O(\lg n)$ depth *whp* by Theorem 10. Computing a spanning forest (Line 5) can be done in the CRCW PRAM in $O(k)$ expected work and $O(\lg k)$ depth *whp* using Gazit's connectivity algorithm [148], and in the same work but $O(\lg^2 k)$ depth in the BF model. We summarize the model-dependent depth of this step as D_{SF}. Finally, updating the adjacency lists and inserting the spanning forest edges into $F_{\lg n}$ costs $O\left(k \lg\left(1 + \frac{n}{k}\right)\right)$ expected work and $O(\lg n)$ depth *whp* (Lines 7–8). □

8.4.3 Deleting Batches of Edges

As in the sequential HDT algorithm, searching for replacement edges after deleting a batch of tree edges is the most interesting part of our parallel algorithm. A natural idea for parallelizing the HDT algorithm is to simply scan all non-tree edges incident on each disconnected component in parallel. Although this approach has low depth per level, it may examine a huge number of candidate edges, but only push down a few non-replacement edges. In general, it is unable to amortize the work performed checking all canidates edges at a level to the edges that experience level decreases. To amortize the work properly while also searching the edges in parallel we must perform a more careful exploration of the non-tree edges. Our approach is to use a *doubling* technique, in which we geometrically increase the number of non-tree edges explored as long as we have not yet found a replacement edge. We show using the doubling technique, the work performed (and number of non-tree edges explored) is dominated by the work of the last phase, when we either find a replacement edge, or run out of non-tree edges. Our amortized work-bounds follow by a per-edge charging argument, as in the analysis of the HDT algorithm.

The Deletion Algorithm. Algorithm 40 shows the pseudocode for our parallel batch deletion algorithm. As with the batch insertion algorithm, we assume that each edge is present in U in both directions. Given a batch of k edge deletions, the algorithm first deletes the given edges from their respective adjacency lists in parallel (Line 2). It then filters out the tree edges (Line 3) and deletes each tree edge e from $F_i \ldots, F_{\lg n}$, where i is the level of e (Line 4). Next, it computes C, a set of *components* (representatives) from the deleted tree edges (Line 5). For each deleted tree edge, e, the algorithm includes the representatives of both endpoints in the forest at $l(e)$, which must be in different components as e is a deleted tree edge. Finally, the algorithm loops over the levels, starting at the lowest level where a tree edge was deleted (Line 7), and calls ParallelLevelSearch at each level. Each call to ParallelLevelSearch takes i, the level to search, C, the current set of disconnected components, and S, an initially empty set of replacement edges that the algorithm discovers over the course of the searches (Line 8)

Algorithm 40 The batch deletion algorithm

```
1: procedure BatchDeletion(U = {e_1, ..., e_k})
2:     Delete e ∈ U from A_0, ..., A_{lg n}
3:     T := {e ∈ U | e ∈ F_{lg n}}                          ▷ tree edges to delete
4:     Delete e ∈ T from F_0, ..., F_{lg n}
5:     C := ∪_{e=(u,v)∈T} (F_{l(e)}.FindRepr(u), F_{l(e)}.FindRepr(v))
6:     S := ∅
7:     for i ∈ [min_l := min_{e∈T}, lg n] do
8:         (C, S) := ParallelLevelSearch(i, C, S)
```

Algorithm 41 The parallel level search algorithm

```
1:  procedure ComponentSearch(i, c)
2:      w := 1
3:      w_max := c.NumNonTreeEdges
4:      while w ≤ w_max do
5:          w := min(w, w_max)
6:          E_c := First w non-tree edges in c
7:          Push all non-replacement edges in E_c to level i − 1
8:          if E_c contains a replacement edge then
9:              return {r}, where r is any replacement edge in E_c
10:         w := 2w
11:     return ∅
12: procedure ParallelLevelSearch(i, L = {c_1, c_2, ...}, S)
13:     F_i.BatchInsert(S)
14:     C := c ∈ L with size ≤ 2^(i−1)
15:     D := c ∈ L with size > 2^(i−1)
16:     while |C| > 0 do
17:         Push level i tree edges of components in C to level i − 1
18:         R := ∪_{c∈C} ComponentSearch(i, c)                    ▷ in parallel
19:         R' := {(F_i.FindRepr(u), F_i.FindRepr(v)) | (u, v) ∈ R}
20:         T' := SpanningForest(R')
21:         T := Edges in R corresponding to edges in T'
22:         Promote edges in T to tree edges
23:         F_i.BatchInsert(T)
24:         S := S ∪ T
25:         C := {F_i.Repr(c) | c ∈ C}
26:         Q := {c ∈ C with no non-tree edges, or size > 2^(i−1)}
27:         D := D ∪ Q
28:         C := C \ Q
29:     return (D, S)
```

The bulk of the work done by the deletion algorithm is performed by Algorithm 41, which implements a subroutine that searches the disconnected components at a given level of the data structure in parallel. The input to ParallelLevelSearch is an integer i, the level to search, a set of representatives of the disconnected components, L, and the set of replacement spanning forest edges that were found in levels lower than i, S. The output of ParallelLevelSearch is the set of components that are still disconnected after considering the non-tree edges at this level, and the set of replacement spanning forest edges found so far.

ParallelLevelSearch starts by inserting the new spanning forest edges in S into F_i

(Line 13). Next, it computes C and D, which are the components that are active and inactive at this level, respectively (Lines 14–15). The main loop of the algorithm (Lines 16–28) operates in a number of ***rounds***. Each round first pushes down all tree edges at level i of every active component. It then finds a single replacement edge incident to each active component, searching the active components in parallel, pushing any non-replacement edge to level $i-1$. It then promotes a maximal acyclic subset of the replacement edges found in this round to tree edges, and proceeds to the next round. The rounds terminate once all components at this level are deactivated by either becoming too large to search at this level, or because the algorithm finished examining all non-tree edges incident to the component at this level.

The main loop (Lines 16–28) works as follows. The algorithm first pushes any level i tree edges in an active component down to level $i-1$. The active components in C have size at most 2^{i-1}, meaning that any tree edges they have at level i can be pushed to level $i-1$ (Line 17) without violating Invariant 1. Next, the algorithm searches each active component for a replacement edge in parallel by calling the ComponentSearch procedure in parallel over all components (Line 18). This procedure either returns an empty set if there are no replacement edges incident to the component, or a set containing a single replacement edge. Next, the algorithm maps the replacement edge endpoints to their current component's representatives by calling FindRepr on each endpoint (Line 19). It then computes a spanning forest over these replacement edges (Line 20) and maps the edges included in the spanning forest back to their original endpoints ids (Line 21). Observe that the edges in T constitute a maximal acyclic subset of replacement edges of R in F_i. The algorithm therefore promotes the edges in T to tree edges (Lines 22– 23). Note that the new tree edges are not immediately inserted into all higher level spanning trees. Instead, the edges are buffered by adding them to S (Line 24) so that they will be inserted when the higher level is reached in the search. Finally, the algorithm updates the set of components by computing their representatives on the updated F_i (Line 25), and filtering out any components which have no remaining non-tree edges, or become larger than 2^{i-1} (i.e., become unsearchable at this level) into D (Lines 30–28).

We now describe the ComponentSearch procedure (Lines 1–11). The search consists of a number of ***phases***, where the i'th phase searches the first 2^i non-tree edges, or all of the non-tree edges if 2^i is larger than the number of non-tree edges in c. The search terminates either once a replacement edge incident to c is found (Line 8), or once the algorithm unsuccessfuly examines all non-tree edges incident to c (Line 4). Initially w, the search size, is set to 1 (Line 5). On each phase, the algorithm retrieves the first w many non-tree edges, E_c (Line 6). It pushes all non-tree edges that are not replacements to level $i-1$ (Line 7). It then checks whether any of the edges in E_c are a replacement edge, and if so, returns one of the replacement edges in E_c (Line 9). Note that checking whether an edge is a replacement edge is done using BatchFindRepr. Otherwise, if no replacement edge was found it doubles w (Line 10) and continues.

Cost Bounds. We now prove that our parallel algorithm has low depth, and is work-efficient with respect to the sequential HDT algorithm. For simplicity, we assume that we start with no edges in a graph on n vertices.

Theorem 13. *A batch of k edge deletions can be processed in $O(\lg^4 n)$ depth whp.*

Proof. The algorithm doubles the number of edges searched in each phase. Therefore, after $\lg m = O(\lg n)$ phases, all non-tree edges incident on the component will be searched.

In every round, each active component is either deactivated, or has a replacement edge found. In the worst case, the edges found for each active component pair the components off, leaving us with half as many active components in the subsequent round. As we lose a constant fraction of the active components per round, the algorithm takes $O(\lg n)$ rounds.

A given level can therefore perform at most $O(\lg^2 n)$ phases. Each phase consists of fetching, examining, and pushing down non-tree edges, and hence can be implemented in $O(\lg n)$ depth *whp* by Lemma 8, Theorem 10, and Lemma 9. Therefore, the overall depth for a given level is $O(\lg^3 n)$ *whp*. As all $\lg n$ levels will be processed in the worst case, the overall depth of the algorithm is $O(\lg^4 n)$. Note that the depth of the spanning forest call within a round is subsumed by the depth of the ComponentSearch procedure, regardless of whether we are working in the BF or CRCW PRAM model. □

We now analyze the work performed by the batch deletion algorithm.

Lemma 10. *The work performed by BatchDeletion excluding the calls to ParallelLevelSearch is*

$$O\left(k \lg n \lg\left(1 + \frac{n}{k}\right)\right), \tag{8.3}$$

in expectation.

Proof. The edge deletions performed by Line 2 cost $O(k)$ work by Lemma 7. Filtering the tree edges (Line 3) can be done in $O(k)$ work. Deleting the tree edges costs at most $O\left(k \lg\left(1 + n/k\right)\right)$ work by Lemma 6 (Line 4).

Line 5 perform a FindRepr call for each endpoint of each deleted tree edge. These calls can be implemented as a single BatchFindRepr call which costs $O\left(k \lg\left(1 + n/k\right)\right)$ work in expectation by Theorem 10. Since in the worst case each tree edge must be deleted from $\lg n$ levels, the overall cost of this step is $O\left(k \lg n \lg\left(1 + n/k\right)\right)$ in expectation. Summing up the costs for each level proves the lemma. □

Theorem 14. *The expected amortized cost per edge insertion or deletion is $O(\lg^2 n)$.*

Proof. Algorithm 40 takes as input a batch of k edge deletions. By Lemma 10, the expected work performed by BatchDeletion excluding the calls to ParallelLevelSearch is

$$O\left(k \lg n \lg\left(1 + \frac{n}{k}\right)\right), \tag{8.4}$$

which is at most $O(k \lg^2 n)$ in expectation. We now consider the cost of the calls to ParallelLevelSearch. Specifically, we show that the work performed during the calls to ParallelLevelSearch can either be charged to level decreases on edges, or is at most $O(k \lg n)$ per call in expectation. Since the total number of calls to ParallelLevelSearch is at most $\lg n$, the bounds follow.

First, observe that the number of spanning forest edges we discover, $|S|$, is at most k, since at most k tree edges were deleted initially. Therefore, the batch insertion on Line 13 costs $O(k \lg n)$ in expectation by Theorem 10. Similarly, L, the number of components that are supplied to ParallelLevelSearch, is at most k. Therefore, the cost of filtering the components in L based on their size, and checking whether their representative exists in F_i is at most $O(k \lg n)$ in expectation (Lines 14–15).

To fetch, examine, and push down l tree or non-tree edges costs

$$O\left(l \lg\left(1 + \frac{n}{l}\right)\right), \tag{8.5}$$

work in expectation, by Lemma 8, Theorem 10, and Lemma 9. Note that this is at most $O(\lg n)$ per edge. In particular, the cost of retrieving and pushing the tree edges of active components to level $i - 1$ (Line 7) is therefore at most $O(\lg n)$ per edge in expectation, which we charge to the corresponding level decreases.

We now show that all work done while searching for replacement edges (Lines 16–28) can be charged to level decreases. Consider an active component, c in some round. Suppose the algorithm performs $q > 0$ phases before either the component is exhausted (all incident non-tree edges have been checked), or a replacement edge is found. First consider the case where it finds a replacement edge. If $q = 1$, only a single edge was inspected, so then we charge the $\lg n$ work for the round to the edge, which will become a tree edge. Otherwise, it performs $q - 1$ phases which do not produce any replacement edge.

Since phase w inspects 2^w edges, it costs $O(2^w \lg n)$ work. The total work over all q phases is therefore

$$\sum_{w=0}^{q} 2^w \lg n = O(2^q \lg n) \tag{8.6}$$

in expectation. However, since no replacement was found during the first $q - 1$ phases, there are at least $2^{q-1} = O(2^q)$ edges that will be pushed down, so we can charge $O(\lg n)$ work to each such edge to pay for this. In the other case, q phases run without finding a replacement edge. In this case, all edges inspected are pushed down, and hence each assumes a cost of $O(\lg n)$ in expectation.

Now, we argue that the work done while processing the replacement edges is $O(k \lg n)$ in expectation over all rounds. Since k edges were deleted, the algorithm discovers at most k replacement edges. We charge the work in these steps to the replacement edges that we find. Let k' be the number of replacement edges that we find. Filtering the edges, and

computing a spanning forest all costs $O(k')$ expected work. Promoting the edges to tree edges (inserting them into F_i and updating the adjacency lists) costs $O(k' \lg n)$ work in expectation. Finally, updating the components costs $O(k' \lg n)$ work in expectation, which we can charge to either the component, if it is removed from C in this round, or to the replacement edge that it finds, which is promoted to a tree edge. Since the algorithm can find at most k replacement edges, the cost per level is $O(k \lg n)$ in expectation for these steps as necessary.

In total, on each level the algorithm performs $O(k \lg n)$ expected work that is not charged to a level decrease. Summing over $\lg n$ levels, this yields an amortized cost of $O(\lg^2 n)$ expected work per edge deletion. Finally, since the level of an edge can decrease at most $\lg n$ times, and an edge is charged $O(\lg n)$ expected work each time its level is decreased, the expected amortized cost per edge insertion is $O(\lg^2 n)$. □

8.5 An Improved Algorithm

In this section we design a improved version of the parallel algorithm that performs less work than our algorithm from Section 8.4. Furthermore, the improved algorithm runs in $O(\lg^3 n)$ depth *whp* in the CRCW PRAM, improving on the $O(\lg^4 n)$ depth *whp* obtained by using Algorithm 41.

8.5.1 The Interleaved Deletion Algorithm

Overview. Algorithm 42 is based on *interleaving* the phases of doubling that search for replacement edges with the spanning forest computation performed on the replacement edges. Recall that in Algorithm 41, the number of edges examined in each round *is reset*, and the doubling algorithm must therefore start with an initial search size of 1 on the next round. Because the doubling resets from round to round, the number of phases per round can be $O(\lg n)$ in the worst case, making the total number of phases per level $O(\lg^2 n)$, and the depth per level $O(\lg^3 n)$. Instead, the interleaved algorithm avoids resetting the search size by maintaining a *single*, geometrically increasing search size over all rounds of the search.

The second important difference in Algorithm 42 compared with Algorithm 41 is that it *defers* inserting tree edges found on this level until the end of the search. Instead, it continues to search for replacement edges from the *initial* components until the component is deactivated. This property is important to show that the work done for a component across all rounds is dominated by the cost of the last round, since the number of vertices in the component is fixed, but the number of non-tree edges examined doubles in each round. For the same reason, it also defers inserting the pushed edges onto level $i - 1$. We crucially use this property to obtain improved batch work bounds (Section 8.6).

Algorithm 42 The interleaved level search algorithm

```
1:  procedure ComponentSearch(i, c, s)
2:      w_max := c.NumNonTreeEdges
3:      w := min(s, w_max)
4:      E_c := First w non-tree edges in c
5:      return {All replacement edges in E_c}
6:  procedure PushEdges(i, c, s, M)
7:      w_max := c.NumNonTreeEdges
8:      w := min(s, w_max)
9:      E_c := {First w non-tree edges in c}
10:     if M[c].size ≤ 2^(i−1) and w < w_max then
11:         Remove edges in E_c from level i
12:         return E_c
13:     return ∅
14: procedure InterleavedLevelSearch(i, L = {c_1, c_2, ...}, S)
15:     F_i.BatchInsert(S)
16:     C := c ∈ L with size ≤ 2^(i−1)
17:     D := c ∈ L with size > 2^(i−1)
18:     Push level i tree edges of all components in C to level i − 1
19:     r := 0, T := ∅, E_P := ∅
20:     M := {c → c | c ∈ C}
21:     while |C| > 0 do
22:         w := 2^r
23:         R := ∪_{c∈C} ComponentSearch(i, c, w)                ▷ in parallel
24:         R' := {(F_i.FindRepr(u), F_i.FindRepr(v)) | (u, v) ∈ R}
25:         T'_r := SpanningForest(R')
26:         T_r := Edges in R corresponding to edges in T'_r
27:         T := T ∪ T_r
28:         Update M, the map of supercomponents and their sizes
29:         E_P := E_P ∪_{c∈C} PushEdges(i, c, w, M)             ▷ in parallel
30:         D_r := {c ∈ C with no non-tree edges, or size > 2^(i−1)}
31:         D := D ∪ D_r
32:         C := C \ D_r
33:         r := r + 1
34:     Promote edges in T not in E_p to tree edges at level i
35:     F_i.BatchInsert(T)
36:     Insert non-tree and tree edges in E_P to level i − 1
37:     return (D, S ∪ T)
```

Another difference in the modified algorithm is that if a component is still active after adding the replacement edges found in this round (i.e., the component on level i still has size at most 2^{i-1}), then *all* of the edges found in this round can be pushed to level $i-1$ without violating Invariant 1. Notice now that when pushing down edges, both the *tree and non-tree* edges that are found in this round are pushed. Pushing down all edges ensures that the algorithm performs enough level decreases to which to charge the work performed during the next round. The component deactivates either once it runs out of incident non-tree edges, or when it becomes too large. Since the algorithm defers adding the new tree edges found until the end of the level, it also maintains an auxiliary data structure that dynamically tracks the size of the resulting components as new edges are found.

The Deletion Algorithm. We briefly describe the main differences between InterleavedLevelSearch, the new level search procedure, and ParallelLevelSearch. The algorithm consists of a number of ***rounds*** (Lines 21–33). We use r to track the round numbers, and we use E_P to store the set of both tree and non-tree edges that will be pushed to level $i-1$ at the end of the search at this level (Line 19). T stores the set of tree edges that have been selected, which will be added to the spanning forest at the end of the level. Lastly, we use M to maintain a dynamic mapping from all the components in L to a unique representative for their contracted supercomponent (initially itself), and the size of the contracted supercomponent.

In round r, the algorithm first retrieves the first 2^r (or fewer) edges of each the active components in parallel, and finds replacement edges. All replacement edges are added to the set R (line 23).

The algorithm then computes a spanning forest over the edges in R, and computes T_r, which are the original replacement edges in R that were selected as spanning forest edges (lines 25–27). The spanning forest computation returns, in addition to the tree edges, a mapping from the vertices in R' to their connectivity label (line 25), which can be used on line 28 to efficiently update the representatives of all affected components and the sizes of the supercomponents.

The next step maps over the components in parallel again, calling PushEdges on each active component, and checks whether the edges searched in this round can be (lazily) pushed to level $i-1$ (Line 29).[3] If a component is still active (its new size is small enough to still be searched, and the component still has some non-tree edges remaining) (line 10), all of the searched edges are removed from the adjacency lists at level i (line 11) and are added to the set of edges that will be pushed to level $i-1$ at the end of the level (Lines 12 and 29). Note that this set of edges contains both replacement tree edges we

[3] Note that the set of edges retrieved by PushEdges in Line 9 is assumed to be the same as the one in Line 4. This assumption is satisfied by using our FetchEdges primitive on a batch-parallel ET-tree, and can be satisfied in general by associating the edges retrieved in ComponentSearch to be used in PushEdges.

discovered, and non-tree edges. The tree-edges can be pushed down to level $i-1$ because the component with the tree edges added has size $\leq 2^{i-1}$.

The end of the round (lines 30–33) handles updating the set of components and incrementing the round number, as in Algorithm 41.

Finally, once all components are inactive, the tree edges found at this level that are not contained in E_p are promoted (the tree edges added to E_p have their level decreased to $i-1$) and inserted into F_i (Lines 34–35), and all edges added to E_P in Line 29 are pushed down to level $i-1$ (Line 36). Note that any tree-edges found in this set are promoted in level $i-1$ and added to F_{i-1}. The procedure returns the set of components and all replacement edges found at this level and levels below it (Line 37).

8.5.2 Cost Bounds

We start by showing that the depth of Algorithm 42 is $O(\lg^3 n)$.

Lemma 11. *The number of rounds performed by Algorithm 42 is $O(\lg n)$ and the depth of each round is $O(\lg n + D_{\mathsf{SF}})$ whp. The depth of the* InterleavedLevelSearch *is therefore $O(\lg(\lg n + D_{\mathsf{SF}}))$ whp.*

Proof. Each round of the algorithm increases the search size of a component by a factor of 2. Therefore, after $O(\lg n)$ rounds, every non-tree edge incident on a component will be considered and the algorithm will terminate.

To argue the depth bound, we consider the main steps performed during a round. Fetching, examining and removing the edges from level i takes $O(\lg n)$ depth *whp* by Lemma 8, Theorem 10, and Lemma 9. Computing a spanning forest on the replacement edges and filtering the components (at most k replacement edges, or components) can be done in $O(D_{\mathsf{SF}})$ depth. The depth per round is therefore $O(\lg n + D_{\mathsf{SF}})$ *whp* and the depth of InterleavedLevelSearch is $O(\lg n(\lg n + D_{\mathsf{SF}}))$ *whp* □

Combining Lemma 11 with the fact that there are $\lg n$ levels gives the following theorem.

Theorem 15. *A batch of k edge deletions can be processed in $O(\lg^2 n(\lg n + D_{\mathsf{SF}}))$ depth whp.*

We have the following corollary on the CRCW PRAM, applying the bounds for Gazit's algorithm [148].

Corollary 2. *A batch of k edge deletions can be processed in $O(\lg^3 n)$ depth whp on the CRCW PRAM.*

We now consider the work performed by the algorithm. We start with a lemma showing that the search-size for a component increases geometrically until the round where the component is deactivated.

Lemma 12. *Consider a component, c, that is active at the end of round $r-1$. If c is not removed from C, then it examines $\geq 2^{r-1}$ edges that are pushed down to level $i-1$ at the end of the search.*

Proof. We prove the contrapositive. Suppose that $< 2^{r-1}$ edges are pushed down in total by c in the last round. Then, we will show that c cannot be active in the next round (i.e., it is removed from C in round $r-1$).

Notice that c must be active at the start of round $r-1$. Consider the check on Line 10, which checks whether $w \leq 2^{r-1}$ and $w < w_{\max}$ on this round. Suppose for the same of contradiction that both conditions are true. Then, by the fact that $w < w_{\max}$, it must be the case that $w = 2^{r-1}$ by Line 8. If the condition is true, then on Line 11 the algorithm adds 2^{r-1} edges to be pushed to level $i-1$, contradicting our assumption that $< 2^{r-1}$ edges are pushed.

Therefore the check on Line 10 must be false, giving that either $w > 2^{i-1}$, or $w = w_{\max}$. This means that c will be marked as inactive on Line 30, and then become deactivated on line 32. Therefore, if $< 2^{r-1}$ edges are pushed down by c in round $r-1$, c is deactivated at the end of the round, concluding the proof.

□

Lemma 13. *Consider the work done by some component c over the course of* InterleavedLevelSearch *at a given level. Let R be the total number rounds that c is active. Then, c pushes down $p_c = 2^R - 1$ edges in total. Furthermore, the total cost of searching for and pushing down replacement edges performed by c is*

$$O\left(p_c \lg\left(1 + \frac{n_c}{p_c}\right)\right) \tag{8.7}$$

in expectation, where n_c is the number of vertices in c.

Proof. By Lemma 12, for each round $r < R$, c adds 2^r edges to be pushed down. Summing over all rounds shows that the total number of edges added to be pushed down is $2^R - 1$. The cost of pushing down these edges at the end of the search at this level is exactly

$$O\left(p_c \lg\left(1 + \frac{n_c}{p_c}\right)\right). \tag{8.8}$$

by Lemma 9, since the size of the tree that is affected is n_c.

We now consider the cost of fetching and examining the edges over all rounds. The cost of fetching and examining 2^r edges is

$$O\left(2^r \lg\left(1 + \frac{n_c}{2^r}\right)\right), \tag{8.9}$$

in expectation by Theorem 10 and Lemma 8. Summing over all rounds $r < R$, the work is

$$\sum_{r=1}^{R-1} O\left(2^r \lg\left(1 + \frac{n_c}{2^r}\right)\right) \tag{8.10}$$

in expectation to fetch and examine edges in the first $R - 1$ rounds, which is equal to

$$O\left(2^R \lg\left(1 + \frac{n_c}{2^R}\right)\right), \tag{8.11}$$

by Lemma 5. Since on round R, the algorithm searches at most 2^R edges, the total cost of searching for replacement edges over all rounds is at most

$$O\left(2^R \lg\left(1 + \frac{n_c}{2^R}\right)\right) = O\left(p_c \lg\left(1 + \frac{n_c}{p_c}\right)\right). \tag{8.12}$$

□

Lemma 14. *The cost of* InterleavedLevelSearch *is at most*

$$O\left(k \lg\left(1 + \frac{n}{k}\right) + p \lg\left(1 + \frac{n}{p}\right)\right) \tag{8.13}$$

in expectation where p is the total number of edges pushed down.

Proof. First consider lines 2–5. Since we are deleting a batch of k edges, we can find at most k replacement edges to reconnect these components. Therefore line 2 performs $O\left(k \lg\left(1 + \frac{n}{k}\right)\right)$ expected work by Theorem 10. Pushing t spanning tree edges to the next level (line 5) can be done in $O\left(t \lg\left(\frac{n}{t} + 1\right)\right)$ expected work by Lemmas 8, 9, and 4, and Theorem 10. Hence in total, lines 2–5 perform at most $O\left(k \lg\left(1 + \frac{n}{k}\right) + t \lg\left(1 + \frac{n}{t}\right)\right)$ work in expectation.

Now, consider the cost of the steps which scan or update the components that are active in each round. On the first round, this cost is $O(k)$. In every subsequent round, r, by Lemma 12 each currently active component must have added 2^{r-1} edges to be pushed down on the previous round. Therefore, we can charge the $O(1)$ work per component performed in this round to these edge pushes.

Next, we analyze the work done while searching for and pushing replacement edges. Consider some component $c \in C$ that is searched on this level. By Lemma 13, the cost of searching for and pushing down the replacement edges incident on this component is

$$O\left(p_c \lg\left(1 + \frac{n_c}{p_c}\right)\right) \tag{8.14}$$

in expectation, where n_c is the number of vertices in c and p_c is the total number of edges pushed down by c.

The total work done over all components to search for replacement edges and push down both the original tree edges, and the edges in each round is therefore

$$O\left(t \lg\left(1 + \frac{n}{t}\right) + \sum_{c \in C} p_c \lg\left(1 + \frac{n_c}{p_c}\right)\right). \tag{8.15}$$

in expectation. Since $\sum n_c = n$, by Lemma 4 this costs

$$O\left(p \lg\left(1 + \frac{2n}{p}\right)\right) = O\left(p \lg\left(1 + \frac{n}{p}\right)\right) \tag{8.16}$$

work in expectation, where $p = t + \sum p_c$ is the total number of edges pushed, including tree and non-tree edges. Therefore, the total cost is

$$O\left(k \lg\left(1 + \frac{n}{k}\right) + p \lg\left(1 + \frac{n}{p}\right)\right) \tag{8.17}$$

in expectation. □

Theorem 16. *The expected amortized cost per edge insertion or deletion is* $O(\lg^2 n)$.

Proof. The proof follows from the same argument as Theorem 14, by using Lemma 14. □

8.6 Improved Work Analysis

We now show that by a more careful analysis, we can obtain a tighter bound on the amount of work performed by the interleaved algorithm. In particular, we show in this section that the algorithm performs

$$O\left(\lg n \lg\left(1 + \frac{n}{\Delta}\right)\right) \tag{8.18}$$

amortized work per edge in expectation, where Δ is the average batch size of all batches of deletions. Therefore, if we process batches of deletions of size $O(n/\text{polylog}(n))$ on average, our algorithm performs $O(\lg n \lg\lg n)$ expected amortized work per edge, rather than $O(\lg^2 n)$. Furthermore, if we have batches of size $O(n)$, the cost is just $O(\lg n)$ per edge.

At a high level, our proof formalizes the intuition that in the worst case, all edges are pushed down at every level, and that performing fewer deletion operations results in larger batches of pushes which take advantage of work bounds of the ET-tree. Our proof crucially relies on the fact that although the deletion algorithm at a level can perform $O(\lg n)$

ET-tree operations per component, since the batch sizes are geometrically increasing, these operations have the cost of a single ET-tree operation per component. Furthermore, Lemma 14 shows that the costs per component can be combined so that the total cost is equivalent to the cost of a single ET-tree operation on all the vertices. Therefore, the number of deletion operations can be exactly related to the effective number of ET-tree operations at a level. We relate the number of deletions to the average batch size, which lets us obtain a single unified bound for both insertions and deletions.

Theorem 17. *Using the interleaved deletion algorithm, the amortized work performed by* BatchDeletion *and* BatchInsertion *on a batch of k edges is*

$$O\left(k \lg n \lg\left(1 + \frac{n}{\Delta}\right)\right), \tag{8.19}$$

in expectation where Δ is the average batch size of all batch deletions.

Proof. Batch insertions perform only $O\left(k \lg\left(1 + \frac{n}{k}\right)\right)$ work by Theorem 12, so we focus on the cost of deletion since it dominates. Consider the total amount of work performed by all batch deletion operations at any given point in the lifetime of the data structure. We will denote by k_b, the size of batch b, and by $p_{b,i}$, the number of edges pushed down on level i during batch b. Combining Lemmas 10, and 14, the total work is bounded above by

$$O\left(\sum_{\text{batch } b} \sum_{\text{level } i} k_b \lg\left(1 + \frac{n}{k_b}\right) + p_{b,i} \lg\left(1 + \frac{n}{p_{b,i}}\right)\right). \tag{8.20}$$

We begin by analyzing the first term, which is paid for by the deletion algorithm. Let

$$K = \sum_{\text{batch } b} k_b \tag{8.21}$$

denote the total number of deleted edges. Applying Lemma 4, and using the fact that there are $\lg n$ levels, we have

$$O\left(\sum_{\text{batch } b} \sum_{\text{level } i} k_b \lg\left(1 + \frac{n}{k_b}\right)\right) = O\left(K \lg n \lg\left(1 + \frac{n \cdot d}{K}\right)\right), \tag{8.22}$$

where d is the number of batches of deletions. Since $K/d = \Delta$, this is equal to

$$O\left(K \lg n \lg\left(1 + \frac{n}{\Delta}\right)\right), \tag{8.23}$$

work in expectation. Each batch can therefore be charged a cost of $\lg n \lg(1 + n/\Delta)$ per edge, and hence the amortized cost of batch deletion is

$$O\left(k \lg n \lg\left(1 + \frac{n}{\Delta}\right)\right) \tag{8.24}$$

in expectation.

The remainder of the cost, which comes entirely from searching for replacement edges, is charged to the insertions. Consider this cost and let

$$P = \sum_{\text{batch } b} \sum_{\text{level } i} p_{b,i} \tag{8.25}$$

denote the total such number of edge pushes. Since the total number of terms in the double sum is $d \lg n$, Lemma 4 allows us to bound the total work of all pushes by

$$\sum_{\text{batch } b} \sum_{\text{level } i} p_{b,i} \lg \left(1 + \frac{n}{p_{b,i}}\right) = O\left(P \lg \left(1 + \frac{nd \lg n}{P}\right)\right). \tag{8.26}$$

in expectation. Since every edge can only be pushed down once per level, we have

$$P \leq m \lg n, \tag{8.27}$$

where m is the total number of edges ever inserted. Therefore by Lemma 6, the total work is at most

$$O\left(m \lg n \lg \left(1 + \frac{nd \lg n}{m \lg n}\right)\right) = O\left(m \lg n \lg \left(1 + \frac{nd}{m}\right)\right) \tag{8.28}$$

in expectation. Since $d = K/\Delta$, this is equal to

$$O\left(m \lg n \lg \left(1 + \frac{nK}{m\Delta}\right)\right) \tag{8.29}$$

in expectation. Since each edge can be deleted only once, we have $K \leq m$, and hence we obtain that the total work to push all tree edges down is at most

$$O\left(m \lg n \lg \left(1 + \frac{n}{\Delta}\right)\right). \tag{8.30}$$

in expectation. We can therefore charge $O(\lg n \lg(1 + n/k))$ per edge to each batch insertion. Since this dominates the cost of the insertion algorithm itself, the amortized cost of batch insertion is therefore

$$O\left(k \lg n \lg \left(1 + \frac{n}{\Delta}\right)\right), \tag{8.31}$$

in expectation as desired, concluding the proof. □

8.7 Discussion

In this chapter, we presented a novel batch-dynamic algorithm for the connectivity problem. Our algorithm is always work-efficient with respect to the Holm, de Lichtenberg and

Thorup dynamic connectivity algorithm, and is asymptotically faster than their algorithm when the average batch size is sufficiently large. A parallel implementation of our algorithm achieves $O(\lg^3 n)$ depth *whp*, and is, to the best of our knowledge, the first parallel algorithm for the dynamic connectivity problem performing $O(T \text{ polylog}(n))$ total expected work, where T is the total number of edge operations.

There are several natural questions to address in future work. First, can the depth of our algorithm be improved to $O(\lg^2 n)$ without increasing the work? Investigating lower bounds in the batch setting would also be very interesting—are there non-trivial lower-bounds for batch-dynamic connectivity? Lastly, in this chapter we show expected amortized bounds. One approach to strengthen these bounds is to show that our tree operations hold *whp* and argue that our amortized bounds hold *whp*. Another is to design a deterministic batch-dynamic forest connectivity data structure with the same asymptotic complexity as the batch-parallel ET-tree, which would make the randomized bounds in this chapter deterministic.

Two additional questions are whether we can extend our results to give parallel work-efficient batch-dynamic MST, 2-edge connectivity and biconnectivity algorithms. MST seems solvable using the techniques presented in this chapter, although our dynamic tree structure would need to be extended with additional primitives. Existing sequential 2-edge connectivity and biconnectivity algorithms require a dynamic tree data structure supporting path queries which are not supported by ET-trees. However, RC-trees [4] can be extended to support path queries, which makes them a possible candidate for this line of work. Finally, it seems likely that ideas from our work can be extended to give a parallel batch-dynamic Monte-Carlo connectivity algorithm based on the Kapron-King-Mountjoy algorithm [193].

Part III

Streaming Graph Processing

Introduction

In recent years, there has been growing interest in programming frameworks for processing streaming graphs due to the fact that many real-world graphs change in real-time (e.g., [130, 139, 156, 100, 226, 379]). These graph-streaming systems receive a stream of queries and a stream of updates (e.g., edge and vertex insertions and deletions, as well as edge weight updates) and must process both updates and queries with low latency, both in terms of query processing time and the time it takes for updates to be reflected in new queries. There are several existing graph-streaming frameworks, such as STINGER, based on maintaining a single mutable copy of the graph in memory [130, 139, 156]. Unfortunately, these frameworks require either blocking queries or updates so that they are not concurrent, or giving up serializability [379]. Another approach is to use snapshots [100, 226]. Existing snapshot-based systems, however, are either very space-inefficient, or suffer from high latency on updates. Therefore, an important question is whether we can design a data structure that supports lightweight snapshots which can be used to concurrently process queries and updates, while ensuring that the data structure is safe for parallelism and achieves good asymptotic and empirical performance.

In principle, representing graphs using *purely-functional balanced search trees* [2, 264] can satisfy both criteria. Such a representation can use a search tree over the vertices (the vertex-tree), and for each vertex store a search tree of its incident edges (an edge-tree). Because the trees are purely-functional, acquiring an immutable snapshot is as simple as acquiring a pointer to the root of the vertex-tree. Updates can then happen concurrently without affecting the snapshot. In fact, any number of readers (queries) can concurrently acquire independent snapshots without being affected by a writer. A writer can make an individual or bulk update and then set the root to make the changes immediately and atomically visible to the next reader without affecting current active readers. A single update costs $O(\lg n)$ work, and because the trees are purely-functional it is relatively easy and safe to parallelize a bulk update.

However, there are several challenges that arise when comparing purely-functional trees to compressed sparse row (CSR), the standard data structure for representing static graphs in shared-memory graph processing [295]. In CSR, the graph is stored as an array of vertices and an array of edges, where each vertex points to the start of its edges in the edge-array. Therefore, in the CSR format, accessing all edges incident to a vertex v takes $O(deg(v))$ work, instead of $O(\lg n + deg(v))$ work for a graph represented using trees. Furthermore, the format requires only one pointer (or index) per vertex and edge, instead of a whole tree node. Additionally, as edges are stored contiguously, CSR has good cache locality when accessing the edges incident to a vertex, while tree nodes could be

spread across memory. Finally, each set of edges can be compressed internally using graph compression techniques [322], allowing massive graphs to be stored using just a few bytes per edge [117]. This approach cannot be used directly on trees. This would all seem to put a search tree representation at a severe disadvantage.

Outline. This part of the thesis first presents *C*-trees, a compressed purely-functional search tree data structure that significantly improves on the space usage and locality of purely-functional trees. The key idea is to use a chunking technique over trees in order to store multiple entries per tree-node. We design theoretically-efficient and practical algorithms for performing batch updates to *C*-trees. Our presentation of *C*-trees can be found in Chapter 9.

Chapter 10 then shows how to apply *C*-trees to build an efficient graph-streaming system. We will show that we can use *C*-trees to store massive dynamic real-world graphs using only a few bytes per edge, thereby achieving space usage close to that of the best static graph processing frameworks. To study the efficiency and applicability of our data structure, we designed Aspen, a graph-streaming framework that extends the interface of Ligra with operations for updating graphs. We show that Aspen is faster than two state-of-the-art graph-streaming systems, STINGER and LLAMA, while requiring less memory, and is competitive in performance with the state-of-the-art static graph frameworks, Galois, GAP, and Ligra+. With Aspen, we are able to efficiently process the largest publicly-available graph with over two hundred billion edges in the graph-streaming setting using a single commodity multicore server with 1TB of memory.

The results in this part of the thesis have appeared in the following publication:

Laxman Dhulipala, Guy E Blelloch, and Julian Shun. "Low-Latency Graph Streaming using Compressed Purely-Functional Trees". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2019, pp. 918–934

9 Compressed Purely-Functional Trees

This chapter of the thesis presents *C*-trees, a compressed purely-functional search tree data structure that significantly improves on the space usage and locality of purely-functional trees. The key idea is to use a chunking technique over trees in order to store multiple entries per tree-node. We design theoretically-efficient and practical algorithms for performing batch updates to *C*-trees, and also show that we can store massive dynamic real-world graphs using only a few bytes per edge, thereby achieving space usage close to that of the best static graph processing frameworks.

9.1 Preliminaries

We assume that we have access to a family of uniformly (purely) random hash functions which we can draw from in $O(1)$ work [108, 267]. In functions from such a family, each key is mapped to an element in the range with equal probability, independent of the values that other keys hash to, and the function can be evaluated for a given key in $O(1)$ work.

Purely-Functional Trees. Purely-functional (mutation-free) data structures preserve previous versions of themselves when modified and yield a new structure reflecting the update [264]. The trees studied in this chapter are binary search trees, which represent a set of ordered elements. In a purely-functional tree, each element is used as a *key*, and is stored in a separate tree node. The elements can be optionally associated with a value, which is stored in the node along with the key. Trees can also be augmented with an associative function f (e.g., +), allowing the sum with respect to f in a range of the tree be queried in $O(\lg n)$ work and depth, where n is the number of elements in the tree.

9.2 Compressed Purely-Functional Trees

In this section, we describe a compressed purely-functional search tree data structure which we refer to as a *C-tree*. After describing the data structure in Section 9.2.1, we argue that our design improves locality and reduces space-usage relative to ordinary purely-functional trees (Section 9.2.2). Finally, we compare the *C*-tree data structure to other possible design choices, such as *B*-trees (Section 9.2.3).

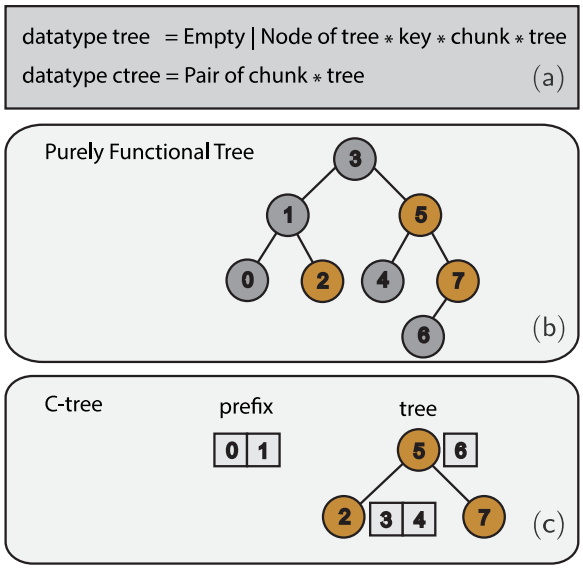


Figure 9.1: This figure gives the definition of the C-tree data structure in an ML-like language (subfigure (a)) and illustrates the difference between a purely-functional tree and a C-tree when representing a set of integers, S. Subfigure (b) shows a purely-functional tree where each element in S is stored in a separate tree node. We color the elements in S that are sampled as heads yellow, and color the non-head elements gray. Subfigure (c) illustrates how the C-tree stores S, given the heads. Notice that the C-tree has a chunk (the prefix) which contains non-head elements that are not associated with any head, and that each head stores a chunk (its tail) containing all non-head elements that follow it until the next head.

9.2.1 C-tree Definition

The main idea of C-trees is to apply a chunking scheme over the tree to store multiple elements per tree-node. The chunking scheme takes the ordered set of elements to be represented and "promotes" certain elements to be heads, which are stored in a tree. The remaining elements are stored in tails associated with each tree node. To ensure that the same keys are promoted in different trees, a hash function is used to choose which elements are promoted. An important goal for C-trees is to maintain similar asymptotic cost bounds as for the uncompressed trees while improving space and cache performance, and to this end we describe theoretically efficient implementations of tree primitives in Section 9.3.

More formally. For an element type K, fix a hash function, $h : K \to \{1, \ldots N\}$, drawn from a uniformly random family of hash functions (N is some sufficiently large range). Let b be a *chunking parameter*, a constant which controls the granularity of the chunks. Given a set E of n elements, we first compute the set of ***heads*** $H(E) = \{e \in E \mid h(e) \bmod b = 0\}$. For each $e \in H(E)$ let its ***tail*** be $t(e) = \{x \in E \mid e < x < next(H(E), e)\}$,

where $next(H(e), e)$ returns the next element in $H(E)$ greater than e. We then construct a purely-functional tree with keys $e \in H(E)$ and associated values $t(e)$.

Thus far, we have described the construction of a tree over the head elements, and their tails. However, there may be a "tail" at the beginning of E that has no associated head, and is therefore not part of the tree. We refer to this chunk of elements as the ***prefix***. We refer to either a tail or prefix as a ***chunk***. We represent each chunk as a (variable-length) array of elements. As described later, when the elements are integers we can use difference encoding to compress each of the chunks. The overall ***C-tree*** data structure consists of the tree over head keys and tail values, and a single (possibly empty) prefix. Figure 9.1 illustrates the C-tree data structure over a set of integer elements.

Properties of C-trees. The expected size of chunks in a C-tree is b as each element is independently selected as a head under h with probability $1/b$. Furthermore, the chunks are unlikely to be much larger than b—in particular, a simple calculation shows that the chunks have size at most $O(b \lg n)$ *whp*, where n is the number of elements in the tree. Notice that an element chosen to be a head will be a head in any C-trees containing it, a property that simplifies the implementation of primitives on C-trees.

Our chunking scheme has the following bounds:

Lemma 15. *The number of heads (keys) in a C-tree over a set E of n elements is $O(n/b)$* whp*. Furthermore, the maximum size of a tail (the non-head nodes associated with a head) or prefix is $O(b \lg n)$* whp*.*

Proof. Each element is selected as a head with probability $1/b$, and so by linearity of expectations, the expected number of heads is n/b. Define X_i to be the independent random variable that is 1 if E_i is a head and 0 otherwise. Let X be their sum, and $E[X] = n/b$. Applying a Chernoff bound proves that the number of heads is $O(n/b)$ *whp*.

We now show that each tail is not too large *whp*. Consider a subsequence of length $t = b \cdot (c \ln n)$ for a constant $c > 1$. The probability that none of the t elements in the subsequence are selected as a head is $(1 - 1/b)^t \leq (1/e)^{c \ln n} = 1/n^c$. Therefore, a subsequence of E of length t has a head *whp*. We complete the proof by applying a union bound over all length t subsequences of E. □

We also obtain the following corollary.

Corollary 3. *When using a balanced binary tree for the heads (one with $O(\lg n)$ height for n keys), the height of a C-tree over a sequence E of n elements is $O(\lg(n/b))$* whp*.*

9.2.2 C-tree Compression

In this section, we first discuss the improved space usage of C-trees relative to purely-functional trees without any assumption on the underlying type of elements. We then

discuss how we can further reduce the space usage of the data structure in the case where the elements are integers.

Space Usage and Locality. Consider the layout of a C-tree compared to a purely-functional tree. By Lemma 15, the expected number of heads is $O(n/b)$. Therefore, compared to a purely-functional tree, which allocates n tree nodes, we reduce the number of tree nodes allocated by a factor of b. As each tree node is quite large (in our implementation, each tree node is at least 32 bytes), reducing the number of nodes by a factor of b can significantly reduce the size of the tree. Experimental results are given in Section 10.4.1.

In a purely-functional tree, in the worst case, accessing each element will incur a cache miss, even in the case where elements are smaller than the size of a cache line. In a C-tree, however, by choosing b, the chunking parameter, to be slightly larger than the cache line size (≈ 128), we can store multiple elements contiguously within a single chunk and amortize the cost of a cache miss across all elements read from the chunk. Furthermore, note that the data structure can provide locality benefits even in the case when the size of an element is larger than the cache line size, as a modest value of b will ensure that reading all but the heads, which constitute an $O(1/b)$ fraction of the elements, will be contiguous loads from the chunks.

Integer C-trees. In the case where the elements are integers, the C-tree data structure can exploit the fact that elements are stored in sorted order in the chunks to further compress the data structure. We apply a *difference encoding* scheme to each chunk. Given a chunk containing d integers, $\{I_1, \ldots, I_d\}$, we compute the differences $\{I_1, I_2 - I_1, \ldots, I_d - I_{d-1}\}$. The differences are then encoded using a byte-code [322, 372]. We applied byte-codes due to the fact that they are fast to decode while achieving most of the memory savings that are possible using a shorter code [59, 372].

Note that in the common case when b is a constant, the size of each chunk is small ($O(\lg n)$ *whp*). Therefore, despite the fact that each chunk must be processed sequentially, the cost of the sequential decoding does not affect the overall work or depth of parallel tree methods. For example, mapping over all elements in the C-tree, or finding a particular element have the same asymptotic work as purely-functional trees and optimal ($O(\lg n)$) depth. To make the data structure dynamic, chunks must also be recompressed when updating a C-tree, which has a similar cost to decompressing the chunks. In the context of graph processing, the fact that methods over a C-tree are easily parallelizable and have low depth lets us avoid designing and implementing a more complex parallel decoding scheme, like the parallel byte-code in Ligra+ [322].

9.2.3 Other Approaches

Our data structure is loosely based on a previous sequential approach to chunking [58]. That approach was designed to be a generic addition to any existing balanced tree scheme for a dictionary and has overheads due to this goal.

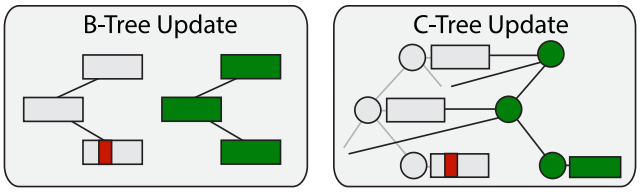


Figure 9.2: This figure shows the difference between performing a single update in a *B*-Tree versus an update in a *C*-tree. The data marked in green is newly allocated in the update. Observe that updating single element in a *C*-tree in the worst-case requires copying a path of nodes, and copying a single chunk if the element is not a head. Updating an element in a *B*-tree requires copying *B* pointers (potentially thousands of bytes) per level of the tree, which adds significant overhead in terms of memory and running time.

Another option is to use *B*-trees [43]. However, the objective of a *B*-tree is to reduce the height of a search tree to accelerate searching a tree in external memory, whereas our goal is to build a data structure that stores many contiguous segments in a single node to make compression possible. The problem with *B*-trees in our purely-functional setting is that we require path copying during functional updates, as illustrated in Figure 9.2. In our trees, this only requires copying a single binary node (32 or 40 bytes in our implementation) per level of the tree. For a *B*-tree, it would require copying *B* pointers (potentially thousands of bytes) per level of the tree, adding significant overhead in terms of memory and running time.

There is also work on chunking of functional trees for representing strings or (unordered) sequences [5, 143, 80, 55]. The motivation is similar (decrease space and increase locality), but the fact they are sequences rather than search trees makes the tradeoffs different. None of this work uses the idea of hashing or efficiently searching the trees. Using a hash function to select the heads has an important advantage in simplifying much of the code, and proving asymptotic bounds. Keeping the elements with internal nodes and using a prefix allows us to access the first b elements (or so) in constant work.

9.3 *Operations on C-trees*

In this section, we show how to support various tree operations over *C*-trees, such as building, searching and performing batch-updates to the data structure. These are operations that we will need for efficiently processing and updating graphs. We argue that the primitives are theoretically efficient by showing bounds on the work and depth of each operation. We also describe how to support augmentation in the data structure using an underlying augmented purely-functional tree. We note that the *C*-tree interfaces defined in this section operate over element-value pairs, whereas the *C*-trees defined in Section 9.2.1 only stored a set of elements for the sake of illustration. The algorithm descriptions elide

the values associated with each element for the sake of clarity. We use operations on an underlying purely-functional tree data structure in our description, and state the bounds for operations on these trees as necessary (e.g., the trees described in Blelloch et al. [63] and Sun et al. [340]). The primitives in this section for a C-tree containing elements of type E and values of type V are defined as follows.

- ***Build***(S, f_V) takes a sequence of element-value pairs and returns a C-tree containing the elements in S with duplicate values combined using a function $f_V : V \times V \to V$.
- ***Find***(T, e) takes a C-tree T and an element e and returns the entry of the largest element $e' \leq e$.
- ***Map***(T, f) takes a C-tree T and a function $f : V \to ()$ and applies f to each element in T.
- ***MultiInsert***(T, f, S) and ***MultiDelete***(T, S) take a C-tree T, (possibly) a function $f : V \times V \to V$ that specifies how to combine values, and a sequence S of element-value pairs, and returns a C-tree containing the union or difference of T and S.

Building. Building (Build(S, f_V)) the data structure can be done in $O(n \lg n)$ work and $O(b \lg n)$ depth *whp* for a sequence of length n. Given an unsorted sequence of elements, we first sort the sequence using a comparison sort which costs $O(n \lg n)$ work and $O(\lg n)$ depth [188]. Duplicate values in S can now be combined by applying a scan with f_V, propagating the sum with respect to f_V rightward, and keeping only the rightmost value in the resulting sequence using a filter.

Next, we hash each element to compute the set of heads and their indices, which can be done using a parallel map and filter in $O(n)$ work and $O(\lg n)$ depth. Constructing the tails for each head can be done in $O(n)$ work and $O(b \lg n)$ depth *whp* by mapping over all heads in parallel and sequentially scanning for the tail, and applying Lemma 15. The prefix is generated similarly. Finally, we build a purely-functional tree over the sequence of head and tail pairs, with the heads as the keys, and the tails as the values, which takes $O(n)$ work and $O(\lg n)$ depth.

Note that the cost is dominated by the cost of sorting, and that the building algorithm only requires $O(n)$ work if the input is sorted.

Searching. Searching (Find(T, e)) for a given element e can be implemented in $O(b \lg n)$ work and depth *whp* and $O(b + \lg n)$ work and depth in expectation. The idea is to simply search the keys in the C-tree for the first head $\leq e$. If the head e' that we find is equal to e we return TRUE, otherwise we check whether e lies in the tail associated with e' sequentially and return TRUE if and only if e is in the tail. The depth of the tree is $O(\lg n)$ and the size of the tail is $O(b \lg n)$ *whp* ($O(b)$ in expectation) by Lemma 15, giving the bounds.

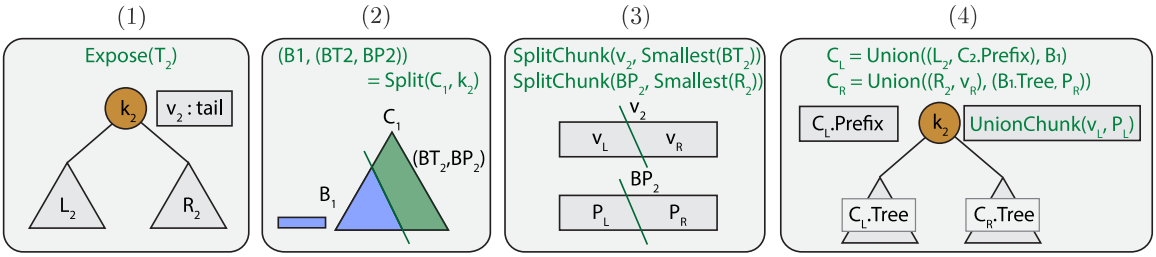


Figure 9.3: This figure illustrates how the UNION algorithm computes the union of two C-trees, T_1 and T_2. The text at the top of each figure (in green) denotes the sub-routine that is called, and the bottom portion of the figure illustrates the output of the call.

Mapping. Mapping (MAP(T, f)) over a C-tree containing n elements with a constant-work function f can be done in $O(n)$ work and $O(b \lg n)$ depth *whp*. We simply apply a parallel map over the underlying purely-functional tree, which runs in $O(n)$ work and $O(\lg n)$ depth [340]. The map operation for each node in the tree simply calls f on the key (a head), and then sequentially processes the tail, applying f to each element in it. We then apply f to each element in the prefix. The work is $O(n)$ as each element is processed once. As each chunk has size $O(b \lg n)$ *whp* by Lemma 15, the overall depth is $O(b \lg n)$ *whp*.

9.4 *Algorithms for Batch Insertions and Deletions*

Our MULTIINSERT and MULTIDELETE algorithms are based on more fundamental algorithms for UNION, INTERSECTION, and DIFFERENCE on C-trees. Since we can simply build a tree over the input sequence to MULTIINSERT and call UNION (or DIFFERENCE for MULTIDELETE), we focus only on the set operations. Furthermore, because the algorithms for INTERSECTION and DIFFERENCE are conceptually very similar to the algorithm for UNION, we only describe in detail the UNION algorithm, and SPLIT, an important primitive used to implement UNION.

Union. Our UNION algorithm (Algorithm 43) is based on the recursive algorithm for UNION given by Blelloch et al. [63]. The main differences between the implementations are how to split a C-tree by a given element, and how to handle elements in the tails and prefixes. The algorithm takes as input two C-trees, C_1 and C_2, and returns a C-tree C containing the elements in the union of C_1 and C_2. Figure 9.3 provides an illustration of how our UNION algorithm computes the union of two C-trees. The algorithms use the following operations defined on C-trees and chunks. The **EXPOSE** operation takes as input a tree and returns the left subtree, the element and prefix at the root of the tree, and the right subtree. The **SPLIT** operation takes as input a C-tree B and an element k, and returns two C-trees B_1 and B_2, where B_1 (resp. B_2) are a C-tree containing all elements less than (resp. greater than) k. It can also optionally return a boolean indicating whether

Algorithm 43 UNION

```
1:  function Union(C1, C2)
2:   case (C1, C2) of
3:     ((null, _), _) → UnionBC(C1, C2)
4:     (_, (null, _)) → UnionBC(C2, C1)
5:     ((T1, P1), (T2, P2)) →
6:   let
7:     val (L2, k2, v2, R2) = Expose(T2)
8:     val (B1, (BT2, BP2)) = Split(C1, k2)
9:     val (vL, vR) = SplitChunk(v2, Smallest(BT2))
10:    val (PL, PR) = SplitChunk(BP2, Smallest(R2))
11:    val v2' = UnionChunk(vL, PL)
12:    val (CL, CR) = Union(B1, (L2, P2)) ||
                      Union((BT2, PR), (R2, vR))
13:  in
14:    ctree(Join(CL.Tree, CR.Tree, k2, v2'), CL.Prefix)
15:  end
```

k was found in B, which is used when implementing DIFFERENCE and INTERSECTION. The **SMALLEST** operation returns the smallest head in a tree. The **UNIONBC** algorithm merges a C-tree consisting of a prefix and empty tree, and another C-tree. We also use the **SPLITCHUNK** and **UNIONCHUNK** operations, which are defined similarly to SPLIT and UNION for chunks.

The idea of the algorithm is to call EXPOSE on the tree of one of the two C-trees (C_2), and split the other C-tree (C_1) based on the element exposed at the root of C_2's tree (Line 7). The split on C_1 returns the trees B_1 and B_2 (Line 8). The algorithm then recursively calls UNION on the C-trees constructed from L_2 and R_2, the left and right subtrees exposed in C_2's tree with the C-trees returned by SPLIT, B_1, and B_2.

However, some care must be taken, since elements in k_2's tail, v_2, may come after some heads in B_2. Similarly, elements in B_2's prefix may come after some heads of R_2. In both cases, we should merge these elements with their corresponding heads' tails. We handle these cases by splitting v_2 by the leftmost element of B_2 (producing v_L and v_R), and splitting B_2's prefix by the leftmost element of R_2 (producing P_L and P_R). The left recursive call to UNION just takes the C-trees B_1 and (L_2, P_2). The right recursive call takes the C-trees (B_2.TREE, P_R), and (R_2, v_R). Note that all elements in the prefixes P_R and v_R are larger than the smallest head in B_2 and R_2. Therefore, the C-tree returned from the right recursive call has an empty prefix. The output of UNION is the C-tree formed by joining the left and right trees from the recursive calls, k_2, and the tail v_2' formed by unioning v_L and P_L, with the prefix from C_L.

UnionBC. Algorithm 44 implements UNIONBC, the base-case of UNION, which computes

Algorithm 44 UnionBC

```
1:  function UnionBC(C1, C2)
2:   case (C1, C2) of
3:     ((null, null), _) → C2
4:   | ((_, P1), (T2, P2)) →
5:   let
6:     val (PL, PR) = SplitChunk(P1, Smallest(T2))
7:     val keys = map(λe.(FindHead(T2, e), e), PR)
8:     val ranges = UniqueKeyRanges(keys)
9:     val updates = map(λ(k, s, e).UnionRange(T2, k, s, e), ranges)
10:    val T2' = MultiInsert(updates, T2)
11:  in
12:    ctree(T2', UnionLists(PL, P2))
13:  end
```

the union of a prefix and a C-tree. If P_1 is null, we return C_2 (Line 3). Otherwise, P_1 is non-empty, and some of its elements may need to be unioned with P_2, while others may belong in tails in T_2. We split P_1 by the first key in T_2 (Line 6), returning the keys in P_1 less than (P_L) and greater than (P_R) the first key in T_2. We first deal with P_R, which contains elements that should be sent to T_2. First, we find the head for each element in P_R in parallel by applying a map over the elements $e \in P_R$ (Line 7). Next, we compute the unique ranges for each key by calling UniqueKeyRanges, which packs out the keys into a sequence of key, start index, and end index triples containing the index of the first and last element that found the key. This step can be implemented by a map followed by a scan operation to propagate the indices of boundary elements, and a pack (Line 8). Next, in parallel for each unique key, we call UnionRange, which unions the elements sent to k with its current tail in T_2 and constructs *updates*, a sequence of head-tail pairs that are to be updated in T_2 (Line 9). Finally, we call MultiInsert with T_2 and *updates*, which returns the tree that we will output (Line 10). Note that the MultiInsert call here operates on the underlying purely-functional tree. We return a C-tree containing this tree, and the union of P_L and P_2 (Line 12). Using the fact that the expected size of P_1 is b, the overall work of UnionBC is $O(b \lg |C_2| + b \cdot b) = O(b^2 + b \lg |C_2|)$ in expectation to perform the finds and merge the elements in P_1 with a corresponding tail. The depth is $O(\lg b \lg |C_2|)$ due to the MultiInsert.

Split. The Split algorithm (Algorithm 45) takes a C-tree (C) and a split element (k), and returns a pair of C-trees where the first contains all elements less than the split element, and the second contains all elements larger than it. The pseudocode is given in an ML-like syntax using pattern matching constructs that enable a simpler description of the algorithm. It first checks to see if C is empty, and returns two empty C-trees if so (Line 3).

Algorithm 45 SPLIT

```
 1: function Split(C, k)
 2:   case C of
 3:     (null, null) → (empty, false, empty)
 4:   | (T, null) →
 5:     let
 6:       val (L, h, v, R) = Expose(T)
 7:     in
 8:       case compare(k, h) of
 9:         EQ → (ctree(L, null), ctree(R, v))
10:       | LT →
11:           let
12:             val (L_L, (LT_R, LP_R)) = Split((L, null), k)
13:           in
14:             (L_L, ctree(Join(LT_R, R, h, v), LP_R))
15:           end
16:       | GT →
17:           if (k ≤ Largest(v)) then
18:             let
19:               val (v_L, v_R) = SplitList(v, k)
20:             in
21:               (ctree(Join(L, null, h, v_L), ctree(R, v_R)))
22:             end
23:           else
24:             let
25:               val ((RT_L, RP_L), R_R) = Split((R, null), k)
26:             in
27:               (ctree(Join(L, RT_L, h, v), RP_L)
28:             end
29:           end
30:   | (T, P) →
31:       let
32:         val (e_l, e_r) = (Smallest(P), Largest(P))
33:       in
34:         if k ≤ e_r then
35:           let
36:             val (P_L, P_R) = SplitChunk(P, k)
37:           in
38:             (ctree(null, P_L), (T, P_R))
39:           end
40:         else
41:           let
42:             val ((T_L, _), C_R) = Split(T, null)
43:           in
44:             (ctree(T_L, P), C_R)
45:           end
46:         end
47:       end
```

Otherwise, if C has a tree but not a prefix (Line 4), the algorithm proceeds into the recursive case which splits a tree. It first exposes T (Line 6), binding h to the head at the root of the tree, v to the head's tail, and L and R to its left and right subtrees, respectively. The algorithm then compares k to the head, h. There are three cases. If k is equal to h (the EQ case on Line 9), the algorithm returns a C-tree constructed from L and a null prefix as the left C-tree, and (R, v) as the right C-tree, since all elements in v are strictly greater than h. Otherwise, if k is less than h (the LT case on Line 10), the algorithm recursively splits the C-tree formed by the left tree with a null prefix, binding L_L as the left C-tree from the recursive call, and (LT_R, LT_P) as the right tree and prefix from the recursive call. It returns L_L as the left C-tree. The right C-tree is formed by joining LT_R with the right subtree (R), with h and v as the head and prefix, and taking the prefix as LT_P. The last case, when k is greater than h (the GT case on Line 16) is more complicated since k can split v, h's tail. The algorithm checks if k splits v (the case $k \leq \textsc{Largest}(v)$ on Line 17), and if so calls SplitList on v based on k (Line 19) to produce v_L and v_R. The algorithm returns a C-tree constructed from L joined with h, and v_L as h's tail as the left C-tree, and a C-tree containing R and v_R as the prefix as the right C-tree. Finally, if $k > \textsc{Largest}(v)$, the algorithm recursively splits R, which is handled similarly to the case where it splits L.

The last case is if C has a non-null prefix, P. In this case, the algorithm tries to split the prefix, and recurses on the tree if the prefix was unsuccessfully split. The algorithm first binds e_l and e_r to the smallest and largest elements in P. It then checks whether $k \leq e_r$. If so, then it splits P based on k to produce P_L and P_R, which contain elements less than and greater than k, respectively. It then returns a C-tree containing an empty tree and P_L as the left C-tree, and T and P_R as the right C-tree. Otherwise, P is not split, but the tree, T may be, and so the algorithm recursively splits T by supplying the C-tree (T, null) to Split. Since T has an empty prefix, splitting T cannot output a left C-tree with a non-empty prefix. We return the recursive result, with P included as the left C-tree's prefix.

9.5 *Parallel Cost Bounds*

Building. Building (Build(S, f_V)) a C-tree can be done in $O(n \lg n)$ work and $O(b \lg n)$ depth *whp* for a sequence of length n. Building a C-tree can be done in $O(n)$ work and $O(b \lg n)$ depth *whp* for a sorted sequence of length n.

Searching. Searching (Find(T, e)) for an element e in a C-tree can be implemented in $O(b \lg n)$ work and depth *whp*, and $O(b + \lg n)$ work and depth in expectation.

Mapping. Mapping (Map(T, f)) over a C-tree containing n elements with a constant-work function f can be done in $O(n)$ work and $O(b \lg n)$ depth *whp*.

Batch Updates. Batch updates (MultiInsert(T, f, S) and MultiDelete(T, S)) can be performed in $O(b^2(k \lg((n/k) + 1)))$ expected work and $O(b \lg k \lg n)$ depth *whp*, where

$k = \min(|T|, |S|)$ and $n = \max(|T|, |S|)$.

Theorem 18. *Split(T, k) performs $O(b \lg n)$ work and depth whp for a C-tree T with n elements. The result holds for all balancing schemes described in [63].*

Proof Sketch. As Split is a sequential algorithm, the depth is equal to the work. We observe that the Split algorithm performs $O(1)$ work at each internal node except in a case where the recursion stops due to the split element, k, lying between Leftmost(P) and Rightmost(P) (line 6), or before Rightmost(v) (line 15). Naively checking whether k lies before Rightmost(v) for each tail, v, on a root-to-leaf path could make us perform $\omega(b \lg n)$ work, but recall that we can store Rightmost(P) at the start of P to make the check run in $O(1)$ work. Therefore, the algorithm performs $O(1)$ work for each internal node.

If the C-tree is represented using a weight-balanced tree, AVL tree, red-black tree, or treap then its height will be $O(\lg n)$ (*whp* for a treap). In the worst-case, the algorithm must recurse until a leaf, and split the tail at the leaf, which has size $O(b \lg n)$ *whp* by Lemma 15. Therefore the work and depth of Split is $O(b \lg n)$ *whp*. The correctness proof follows by induction and case analysis. □

Theorem 19. *For two C-trees T_1 and T_2, the Union algorithm runs in $O(b^2(k \lg((n/k) + 1)))$ work in expectation and $O(b \lg k \lg n)$ depth whp, where $k = \min(|T_1|, |T_2|)$ and $n = \max(|T_1|, |T_2|)$.*

Proof sketch. The extra work performed in our algorithm is due to splitting and unioning tails at each recursive call, and the work performed in UnionBC. Using the fact that the expected size of each tail is $O(b)$ we can modify the proof of the work of Union given in Theorem 6 in [63] to bound our work. In particular, we perform $O(b)$ work in expectation for each node with non-zero splitting cost which pays at least 1 unit of cost in the proof in [63]. To account for the work of the UnionBC, observe that the dominant cost in the algorithm are the calls to Find on Line 6. Also notice that calls operate on a tree, T, generated by a Split from the parent of this call, and the work of this step is $O(b(b + \lg |T|))$ in expectation. We can therefore bound this work by charging each call to UnionBC to the Split call that generated it and applying linearity of expectations over all calls to UnionBC. As we already pay $O(b \lg |T|)$ for the call to Split in the proof from [63] the overall work is affected by an extra factor b^2, resulting in the stated work bound. Note that for $b = O(1)$ the work is affected by a constant factor in expectation.

To bound the depth, observe that the depth of the call-tree (including the depth of splits) can be bounded as $O(b \lg n \lg k)$ using the recurrence as Theorem 8 in [63]. Furthermore, the depth due to splitting tails in recursive calls of Union is $O(b \lg n)$ *whp* per level, which is the same as the depth due to a call to Split, and does not therefore increase the depth.

Finally, although UnionBC can potentially have $O((\lg b + \lg\lg n)\lg k)$ depth due to the MultiInsert, UnionBC only appears as a leaf in the call-tree, and so its contribution to the depth is additive. Thus the overall depth is $O(b \lg n \lg k)$ *whp*. □

Intersection and Difference. Lastly, for Intersection and Difference, we note that the main difference between Union and Intersection and Difference is that they may require removing the split key (which is always maintained and joined with using Join in Union). The only extra work is an implementation of Join2 over C-trees which is similar to Join except it does not take a key in the middle (see [63] for details on Join2).

9.6 Discussion

In this chapter, we have presented a compressed purely-functional balanced tree data structure called a C-tree, and designed efficient parallel batch update algorithms the structure. We have also compared the structure to existing tree data structures, and provided some intuition explaining our design. For future work, it would be interesting to understand whether the dependence on b in the bounds for the data structure can be improved, and also whether one can show lower bounds for batch updates to compressed purely-functional trees in a realistic machine model such as the pointer-machine [49]. Finally, it would be interesting to design a similar data structure that provides deterministic bounds on its operations.

10 Aspen: A Graph-Streaming Framework

This chapter of the thesis presents Aspen, a low-latency graph-streaming framework extending the Ligra interface with operations for updating graphs. In Section 10.1, we first introduce a new purely-functional graph representation based on nested purely-functional trees or C-trees, and discuss how various update operations can be efficiently supported using this representation. In Section 10.2, we then describe several optimizations that enable fast parallel graph processing over graphs stored in this format. Finally, in Sectionsec:aspen we present the Aspen graph processing framework. We end the chapter with a detailed performance evaluation of our implementation, as well as a discussion of how the C-tree based representation enables lower memory usage, showing that we can represent massive real-world graphs in this format using only a few bytes per edge (Section 10.4).

10.1 Representing Graphs as Trees

An undirected graph can be represented using purely functional tree-based data structures by representing the set of vertices as a tree, which we call the ***vertex-tree***. Each vertex in the vertex-tree represents its adjacency information by storing a tree of identifiers of its adjacent neighbors, which we call the ***edge-tree***. Directed graphs can be represented in the same way by simply storing two edge-trees per vertex, one for the out-neighbors, and one for the in-neighbors. The resulting graph data structure is a tree-of-trees that has $O(\lg n)$ overall depth using any balanced tree implementation (*whp* using a treap).

Figure 10.1 illustrates the vertex-tree and the edge-trees for an example graph (subfigure (a)). Subfigure (b) illustrates how the graph is represented using simple trees for both the vertex-tree and edge-tree. Subfigure (c) illustrates using a simple tree for the vertex-tree and a C-tree for the edge-tree.

We augment the vertex-tree to store the number of edges contained in its subtrees, which is needed to compute the total number of edges in the graph in $O(1)$ work. Weighted graphs can be represented using the same structure, with the only difference being that the elements in the edge-trees are modified to store an associated edge weight. Note that computing associative functions over the weights (e.g., aggregating the sum of all edge-weights) could be easily done by augmenting the edge and vertex-trees. We also note that the vertex-tree could also be compressed using a C-tree but defer evaluating this idea for future work.

Basic Graph Operations. We can compute the number of vertices and number of edges

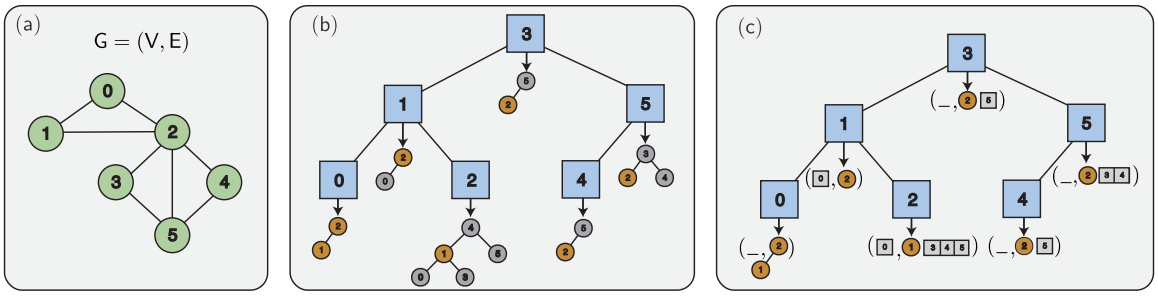


Figure 10.1: We illustrate how the graph (shown in subfigure (a)) is represented as a simple tree of trees (subfigure (b)) and as a tree of C-trees (subfigure (c)). As in Figure 9.1, we color elements (in this case vertex IDs) that are sampled as heads yellow. The prefix and tree in each C-tree are drawn as a tuple, following the datatype definition in Figure 9.1.

in the graph by querying the size (number of keys) in the vertex-tree and the augmented value of the vertex-tree respectively, which can both be done in $O(1)$ work. Finding a particular vertex just searches the vertex-tree, which takes $O(\lg n)$ work and depth.

EdgeMap. We implement EdgeMap (defined in Chapter 2) by mapping over the vertices in the input vertexSubset U in parallel and for each vertex $u \in U$ searching the vertex-tree for its edge-tree, and then mapping over u's incident neighbors, again in parallel. For each of u's neighbors v, we apply the map function $F(u, v)$ if the filter function $C(v)$ returns *true*. Other than finding vertices in the input vertexSubset in G and traversing edges via the tree instead of an array, the implementation is effectively the same as in Ligra [319]. The direction optimization [44, 319] can also be implemented, and we describe more details later in this section. Assuming the functions F and C take constant work, EdgeMap takes $O(\sum_{u \in U} deg(u) + |U| \lg n)$ work and $O(\lg n)$ depth.

Batch Updates. Inserting and deleting edges are defined similarly, and so we only provide details for InsertEdges. Note that updates (e.g., to the weight) of existing edges can be done within this interface. Let A be the sequence (batch) of edge updates and let $k = |A|$.

We first sort the batch of edge pairs using a comparison sort. Next, we compute an array of source vertex IDs that are being updated and for each ID, in parallel, build a tree over its updated edges. We can combine duplicate updates in the batch by using the duplicate-combining function provided by the C-tree constructor. As the sequence is sorted, the build costs $O(k)$ work and $O(\lg k)$ depth. Next, in the update step, we call MultiInsert over the vertex-tree with each $(source, tree)$ pair in the previous sequence. The combine function for MultiInsert combines existing values (edge-trees) with the new edge-trees by calling Union on the old edge-tree and new edge-tree.

Simple Work Analysis. We first give a simple worst-case analysis of the batch update algorithm and show that the algorithm performs $O(k \lg n)$ expected work overall, and has $O(\lg^3 n)$ depth *whp*. All steps before the MultiInsert cost $O(k \lg k)$ work, and $O(\lg k)$

depth in total, as they sort and apply parallel sequence operations to sequences of length k [188]. As the depth of both the vertex-tree and edge-tree is $O(\lg n)$, the overall work of updating both the vertex-tree and each affected edge-tree can be upper bounded by $O(k \lg n)$. The depth of MultiInsert is $O(\lg n(\lg m + D_{\text{Union}}))$, where $D_{\text{Union}})$ is the depth of union. This quantity simplifies to $O(\lg^3 n)$ by upper-bounding D_{Union} on any two trees as $O(\lg^2 n)$.

Improved Work Analysis. Here, we provide a more careful analysis showing that the batch update algorithm described above runs in $O(k \lg(1 + \max(m, n)/k))$ expected work and $O(\lg^3 n)$ depth *whp* for k edge updates. First, assume that the k updates are sorted (otherwise, they can be sorted within the required bounds using a parallel radix sort). The cost to perform a MultiInsert to update and insert vertices in the vertex-tree depends on the number of distinct vertices being updated by the k updates, which can range anywhere between $[1, \min(k, n)]$. Let the number of vertices updated be k'. The cost of this MultiInsert is $O(k' \lg(1 + n/k))$, which is maximized for $k' = k$ since this function is strictly increasing (see Lemma 6 from Section 8.2).

Next, we must bound the cost of applying the updates to each edge-tree. Without loss of generality, the updates are insertions. Constructing C-trees for the new updates to each updated vertex can be done in $O(k)$ work and $O(\lg k)$ depth overall. The remaining cost comes from applying Union over the old trees and the newly constructed trees. The cost of this step depends again on how the updates are distributed, and the sizes of the trees the updates are being applied to. Notice that the total size of the trees being updated is m. Specifically, we are distributing k updates into a set of k' trees that contain a total of m elements. If the i'th tree has size T_i and receives k_i elements, the overall cost of this step is:

$$\sum_{i=1}^{k'} k_i \lg(1 + S_i/k_i)$$

where $\sum_{i=1}^{k'} S_i \leq m$ and $\sum_{i=1}^{k'} k_i = k$. By Lemma 4 from Section 8.2, the total cost of this step is $O(k \lg(1 + m/k))$. For C-trees this bound holds in expectation, but note that it holds deterministically for deterministic purely-functional trees such as weight-balanced trees, at the expense of losing compression. The depth analysis is identical to the simple analysis shown above. Thus, for any constant b, we have the following theorem.

Theorem 20. *Our nested purely-functional graph representation based on C-trees supports k insertions or deletions of edges in $O(k \lg(1 + \max(m, n)/k))$ expected work and $O(\lg^3 n)$ depth whp.*

10.2 Efficiently Implementing Graph Algorithms

We now address how to efficiently implement graph algorithms using a tree of C-trees, mitigating the increase in access times due to using trees. We first describe a technique for handling the asymptotic increase in work for global graph algorithms due to using trees. We then consider local algorithms, and argue that for many local algorithms, the extra cost of searching the vertex-tree can be amortized. Finally, we describe how direction optimization [44] can be easily implemented over the C-tree data structure.

Flat Snapshots. Notice that algorithms in our framework that use EDGEMAP incur an extra $O(K \lg n)$ factor in their work, where K is the total number of vertices accessed by EDGEMAP over the course of the algorithm. For an algorithm like breadth-first search, which runs in $O(m + n)$ work and $O(D \lg n)$ depth for a graph with diameter D using a static-graph processing framework [117], a naive implementation using our framework will require performing $O(m + n \lg n)$ work (the depth is the same, assuming that b is a constant).

Instead, for *global graph algorithms*, which we loosely define as performing $\Omega(n)$ work, we can afford to take a *flat snapshot* of the graph, which reduces the $O(K \lg n)$ term to $O(K)$. The idea of a flat snapshot is very simple—instead of accessing vertices through the vertex-tree, and calling FIND for each v supplied to EDGEMAP, we just precompute the pointers to the edge-trees for all $v \in V$ and store them in an array of size n. This can be done in linear work and $O(\lg n)$ depth by traversing the vertex-tree once to fetch the pointers. By providing this array, which we call a ***flat snapshot*** to each call to EDGEMAP, we can directly access the edges tree in $O(1)$ work and reduce the work of EDGEMAP on a vertexSubset, U, to $O(\sum_{u \in U} deg(u) + |U|)$. In practice, using a flat snapshot speeds up BFS queries on our input graphs by an average of 1.26x (see Table 10.7).

Local Algorithms. In the case of local graph algorithms, we often cannot afford to create a flat snapshot without a significant increase in the work. We observe, however, that after retrieving a vertex many local algorithms will process all edges incident to it. Because the average degree in real-world graphs is often in the same range or larger than $\lg n$ (see Table 10.1), the logarithmic overhead of accessing a vertex in the vertex-tree in these graphs can be amortized against the cost of processing the edges incident to the vertex, on average.

Direction Optimization. Direction optimization is a technique first described for breadth-first search in Beamer et al. [44], and later generalized as part of Ligra in its EDGEMAP implementation [319]. It combines a sparse traversal, which applies the F function in EDGEMAP to the outgoing neighbors of the input vertexSubset U, with a dense traversal, which applies F to the incoming neighbors u of all vertices v in the graph where $C(v) = true$ and $u \in U$. The dense traversal improves locality for large input vertexSubsets, and reduces edge traversals in some algorithms, such as breadth-first search. The traversal mode on

each iteration is selected based on the size of U and its out-degrees. We implemented the optimization by implementing a sparse traversal and a dense traversal that traverses the underlying C-trees.

10.3 Aspen Graph-Streaming Framework

Interface Overview. In this section, we outline the Aspen interface and implementation for processing streaming graphs. The Aspen interface is an extension of Ligra's interface. It includes the full Ligra interface—vertexSubsets, EDGEMAP, and various other functionality on a fixed graph. On top of Ligra, we add a set of functions for updating the graph—in particular, for inserting or deleting sets of edges or sets of vertices. We also add a flat-snapshot function. Aspen currently does not support weighted edges, but we plan to add this functionality using a similar compression scheme for weights as used in Ligra+ in the future. All of the functions for processing and updating the graph work on a *fixed and immutable version (snapshot)* of the graph. The updates are functional, and therefore instead of mutating the version, return a handle to a new graph. The implementation of these operations follow the description given in the previous sections.

The Aspen interface supports three functions, ACQUIRE, SET, and RELEASE, for acquiring the current version of a graph, setting a new version, and releasing a the version. The interface is based on the recently defined *version maintenance problem* and implemented with the corresponding lock-free algorithm to solve it [52]. RELEASE returns whether it is the last copy on that version, and if so we garbage collect it. The three functions each act atomically. The framework allows any number of concurrent readers (i.e., transactions that ACQUIRE and RELEASE but do not set) and a single writer (ACQUIRES, SETS, and then RELEASES). Multiple concurrent readers can acquire the same version, or different versions depending on how the writer is interleaved with them. The implementation of this interface is non-trivial due to race conditions between the three operations. Importantly, however, no reader or writer is ever blocked or delayed by other readers or writers. The Aspen implementation guarantees strict serializability, which means that the state of the graph and outputs of queries are consistent with some serial execution of the updates and queries corresponding to real time.

10.3.1 Aspen Interface

We start by defining a few types used by the interface. A **versioned_graph** is a data type that represents multiple snapshots of an evolving graph. A **version** is a purely-functional snapshot of a versioned_graph. A **T seq** is a sequence of values of type **T**. Finally, a **vertex** is a purely-functional vertex contained in some version.

Building and Update Primitives. The main functions in our interface are a method to construct the initial graph, methods to acquire and release versions, and methods to modify a graph. The remaining functions in the interface are for traversing and analyzing versions and are similar to the Ligra interface. Aspen's functions are listed below:

BuildGraph(n : int, m : int,
S : int seq seq) : versioned_graph
Creates a versioned graph containing n vertices and m edges. The edges incident to the i'th vertex are given by $S[i]$.

acquire() : (VG : versioned_graph) : version
Returns a valid version of a versioned_graph VG. Note that this version will be persisted until the user calls release.

release() : (VG : versioned_graph, G : version)
Releases a version of a versioned_graph VG.

InsertEdges() : (VG : versioned_graph, E': int $\times$ int seq)
Updates the latest version of the graph, $G = (V, E)$, by inserting the edges in E' into G. Makes a new version of the graph equal to $G[E \cup E']$ visible to readers.

DeleteEdges() : (VG : versioned_graph, E': int $\times$ int seq)
Updates the latest version of the graph, $G = (V, E)$, by deleting the edges in E' from G. Singleton vertices (those with degree 0 in the new version of the graph) can be optionally removed. Makes a new version of the graph equal to $G[E \setminus E']$ visible to readers.

InsertVertices() : (VG : versioned_graph, V': int seq)
Updates the latest version of the graph, $G = (V, E)$, by inserting the vertices in V' into G. Makes a new version of the graph equal to $G[V \cup V']$ visible to readers.

DeleteVertices() : (VG : versioned_graph, V': int seq)
Updates the latest version of the graph, $G = (V, E)$, by deleting the vertices in V' from G. Makes a new version of the graph equal to $G[V \setminus V']$ visible to readers.

Our framework also supports similarly-defined primitives for updating values associated with edges (e.g., edge weights) and updating values associated with vertices (e.g., vertex weights). The interface is similar to the basic primitives for updating edges and vertices.

Access Primitives. The functions for accessing a graph are defined similarly to Ligra. For completeness, we list them below. We also provide primitives over the vertex object, such as Degree, Map, and Intersection.

NumVertices (NumEdges)() : (G : version) : int
Returns the number of vertices (edges) in the graph.

FindVertex() : (G : version, v: int) : {vertex $\cup$ $\square$}
Returns either the vertex corresponding to the vertex identifier v, or $\square$ if v is not present in G.

Graph	Num. Vertices	Num. Edges	Avg. Deg.
LiveJournal	4,847,571	85,702,474	17.8
com-Orkut	3,072,627	234,370,166	76.2
Twitter	41,652,231	2,405,026,092	57.7
ClueWeb	978,408,098	74,744,358,622	76.4
Hyperlink2014	1,724,573,718	124,141,874,032	72.0
Hyperlink2012	3,563,602,789	225,840,663,232	63.3

Table 10.1: Statistics about our input graphs.

EDGEMAP() : (G : version, U : vertexSubset, F: int × int → bool, C: int → bool) : vertexSubset
Given a vertexSubset U, returns a vertexSubset U' containing all v such that $(u, v) \in E$ for $u \in U$ and $C(v) =$ true and $F(u, v) =$ true.

10.3.2 Implementation

Aspen is implemented in C++ and uses PAM [340] as the underlying purely-functional tree data structure for storing the heads. Our C-tree implementation requires about 1400 lines of C++, most of which are for implementing UNION, DIFFERENCE, and INTERSECT. Our graph data structure uses an augmented purely-functional tree from PAM to store the vertex-tree. Each node in the vertex tree stores an integer C-tree storing the edges incident to each vertex as its value. We note that the vertex-tree could also be compressed using a C-tree, but we did not explore this direction in the present work. To handle memory management, our implementations use a parallel reference counting garbage collector along with a custom pool-based memory allocator. The pool-allocation is critical for achieving good performance due to the large number of small memory allocations in the the functional setting. Although C++ might seem like an odd choice for implementing a functional interface, it allows us to easily integrate with PAM and Ligra. We also note that although our graph interface is purely-functional (immutable), our global and local graph algorithms are not. They can mutate local state within their transaction, but can only access the shared graph through an immutable interface.

10.4 Experiments

Algorithms. We implemented five algorithms in Aspen, consisting of three global algorithms and two local algorithms. Our global algorithms are breadth-first search (***BFS***), single-source betweenness centrality (***BC***), and maximal independent set (***MIS***). Our BC implementation computes the contributions to betweenness scores for shortest paths emanating from a single vertex. The algorithms are similar to the algorithms in [117] and required only minor changes to acquire a flat snapshot and include it as an argument to

Graph	Flat Snap.	Aspen Uncomp.	Aspen (No DE)	Aspen (DE)	Savings
LiveJournal	0.0722	2.77	0.748	0.582	4.75x
com-Orkut	0.0457	7.12	1.47	0.893	7.98x
Twitter	0.620	73.5	15.6	9.42	7.80x
ClueWeb	14.5	2271	468	200	11.3x
Hyperlink2014	25.6	3776	782	363	10.4x
Hyperlink2012	53.1	6889	1449	702	9.81x

Table 10.2: Statistics about the memory usage using different formats in Aspen. **Flat Snap.** shows the amount of memory in GBs required to represent a flat snapshot of the graph. **Aspen Uncomp.**, **Aspen (No DE)**, and **Aspen (DE)** show the amount of memory in GBs required to represent the graph using uncompressed trees, Aspen without difference encoding of chunks, and Aspen with difference encoding of chunks, respectively. **Savings** shows the factor of memory saved by using Aspen (DE) over the uncompressed representation.

EDGEMAP. As argued in Section 10.2, the cost of creating the snapshot does not asymptotically affect the work or depth of our implementations. The work and depth of our implementations of BFS, BC, and MIS are identical to the implementations in [117]. Our local algorithms are 2-***hop*** and ***Local-Cluster***. 2-hop computes the set of vertices that are at most 2 hops away from the vertex using EDGEMAP. The worst-case work is $O(m + n \lg n)$ and the depth is $O(\lg n)$. Local-Cluster is a sequential implementation of the Nibble-Serial graph clustering algorithm (see [325, 335]), run using $\epsilon = 10^{-6}$ and $T = 10$.

In our experiments, we run the global queries one at a time due to their large memory usage and significant internal parallelism, and run the local queries concurrently (many at the same time).

Experimental Setup. Our experiments are performed on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4 × 2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use a work-stealing scheduler that we implemented. The scheduler is implemented similarly to Cilk for parallelism. Our programs are compiled with the g++ compiler (version 7.3.0) with the `-O3` flag. All experiments involving balanced-binary trees use weight-balanced trees as the underlying balanced tree implementation [63, 340]. We use Aspen to refer to the system using C-trees and difference encoding within each chunk and explicitly specify other configurations of the system if necessary.

Graph Data. Table 10.1 lists the graphs we use. ***LiveJournal*** is a directed graph of the LiveJournal social network [81]. ***com-Orkut*** is an undirected graph of the Orkut social network. ***Twitter*** is a directed graph of the Twitter network, where edges represent the follower relationship [208]. ***ClueWeb*** is a Web graph from the Lemur project at CMU [81]. ***Hyperlink2012*** and ***Hyperlink2014*** are directed hyperlink graphs obtained from the WebDataCommons dataset where nodes represent web pages [241]. Hyperlink2012 is the *largest publicly-available graph*, and we show that *Aspen is able to process it on a single*

Application	LiveJournal			com-Orkut			Twitter		
	(1)	**(72h)**	**(SU)**	**(1)**	**(72h)**	**(SU)**	**(1)**	**(72h)**	**(SU)**
BFS	0.981	0.021	46.7	0.690	0.015	46.0	7.26	0.138	52.6
BC	4.66	0.075	62.1	4.58	0.078	58.7	81.2	1.18	68.8
MIS	3.38	0.054	62.5	4.19	0.069	60.7	71.5	0.99	72.2
2-hop	4.36e-3	1.06e-4	41.1	2.95e-3	6.82e-5	43.2	0.036	8.70e-4	41.3
Local-Cluster	0.075	1.64e-3	45.7	0.122	2.50e-3	48.8	0.127	2.59e-3	49.0

Table 10.3: Running times (in seconds) of algorithms in Aspen for symmetric graph inputs where **(1)** is the single threaded time **(72h)** is the 72-core time (with hyper-threading, i.e., 144 threads), and **(SU)** is the self-relative speedup.

multicore machine. We symmetrized the graphs in our experiments, as the running times for queries like BFS and BC are more consistent on undirected graphs due to the majority of vertices being in a single large component.

Overview of Results. We show the following experimental results in this section.

- The most memory-efficient representation of *C*-trees saves between 4–11x memory over using uncompressed trees, and improves performance by 2.5–2.8x compared to using uncompressed trees (Section 10.4.1).
- Algorithms implemented using Aspen are scalable, achieving between 32–78x speedup across inputs (Section 10.4.2).
- Updates and queries can be run concurrently in Aspen with only a slight increase in latency (Section 10.4.3).
- Parallel batch updates in Aspen are efficient, achieving between 105–442M updates/sec for large batches (Section 10.4.4).
- Aspen outperforms Stinger by 1.8–10.2x while using 8.5–11.4x less memory (Section 10.4.5).
- Aspen outperforms LLAMA by 2.8–7.8x while using 1.9–3.5x less memory (Section 10.4.6).
- Aspen is competitive with state-of-the-art static graph processing systems, ranging from being 1.4x slower to 30x faster (Section 10.4.7).

10.4.1 Chunking and Compression in Aspen

Memory Usage. Table 10.2 shows the amount of memory required to represent real-world graphs in Aspen without compression, using *C*-trees, and finally using *C*-trees with difference encoding. In the uncompressed representation, the size of a vertex-tree node is 48 bytes, and the size of an edge-tree node is 32 bytes. On the other hand, in the compressed representation, the size of a vertex-tree node is 56 bytes (due to padding and extra pointers for the prefix) and the size of an edge-tree node is 48 bytes. We calculated the memory footprint of graphs that require more than 1TB of memory in the uncompressed format by hand, using the sizes of nodes in the uncompressed format.

Application	ClueWeb			Hyperlink2014			Hyperlink2012		
	(1)	**(72h)**	**(SU)**	**(1)**	**(72h)**	**(SU)**	**(1)**	**(72h)**	**(SU)**
BFS	186	3.69	50.4	362	6.19	58.4	1001	14.1	70.9
BC	1111	21.8	50.9	1725	24.5	70.4	4581	58.1	78.8
MIS	955	12.1	78.9	1622	22.2	73.0	3923	50.8	77.2
2-hop	0.883	0.021	42.0	1.61	0.038	42.3	3.24	0.0755	42.9
Local-Cluster	0.016	4.45e-4	35.9	0.022	6.75e-4	32.5	0.028	6.82e-4	41.0

Table 10.4: Running times (in seconds) of our algorithms over symmetric graph inputs where **(1)** is the single threaded time **(72h)** is the 72-core time (with hyper-threading, i.e., 144 threads), and **(SU)** is the self-relative speedup.

We observe that by using C-trees and difference encoding to represent the edge trees, we reduce the memory footprint of the dynamic graph representation by 4.7–11.3x compared to the uncompressed format. Using difference encoding provides between 1.2–2.3x reduction in memory usage compared to storing the chunks in an uncompressed format. We observe that both using C-trees and compressing within the chunks is crucial for storing and processing our largest graphs in a reasonable amount of memory.

Application	**Graph**	**Aspen Uncomp.**	**Aspen**	**(S)**
BFS	LiveJournal	0.055	0.021	2.6x
	com-Orkut	0.042	0.015	2.8x
	Twitter	0.348	0.138	2.5x

Table 10.5: Aspen Uncomp. is the parallel time using Aspen with uncompressed trees, and **Aspen** is the parallel time of Aspen with C-trees and difference encoding. **(S)** is the speedup obtained by Aspen over the uncompressed format. All times are measured on 72 cores using hyper-threading.

Comparison with Uncompressed Trees. Next, we study the performance improvement gained by the improved locality of the C-tree data structure. Due to the memory overheads of representing large graphs using the uncompressed format (see Table 10.2), we are only able to report results for our three smallest graphs, LiveJournal, com-Orkut, and Twitter, as we cannot store the larger graphs even with 1TB of RAM in the uncompressed format. We ran BFS on both the uncompressed and C-tree formats (using difference encoding) and show the results in Table 10.5. The results show that using the compressed representation improves the running times of these applications from between 2.5–2.8x across these graphs.

Choice of Chunk Size. Next, we consider how Aspen performs as a function of the expected chunk size, b. Table 10.6 reports the amount of memory used, and the BFS, BC, and MIS running times as a function of b. In the rest of the paper, we fixed $b = 2^8$, which

b (Exp. Chunk Size)	Memory	BFS (72h)	BC (72h)	MIS (72h)
2^1	68.83	0.309	2.72	2.17
2^2	41.72	0.245	2.09	1.71
2^3	26.0	0.217	1.68	1.41
2^4	17.7	0.172	1.45	1.24
2^5	13.3	0.162	1.32	1.14
2^6	11.1	0.152	1.25	1.07
2^7	9.97	0.142	1.22	1.01
2^8	9.42	**0.138**	**1.18**	0.99
2^9	9.17	0.141	1.20	0.99
2^{10}	9.03	0.152	1.19	0.98
2^{11}	8.96	0.163	1.20	0.98
2^{12}	**8.89**	0.170	1.21	**0.98**

Table 10.6: Memory usage (gigabytes) and performance (seconds) for the Twitter graph as a function of the (expected) chunk size. All times are measured on 72 cores using hyper-threading. Bold-text marks the best value in each column. We use 2^8 in the other experiments.

.

we found gave the best tradeoff between the amount of memory consumed (it requires 5% more memory than the most memory-efficient configuration) while enabling good parallelism across different applications.

10.4.2 Parallel Scalability of Aspen

Algorithm Performance. Tables 10.3 and 10.4 report experimental results including the single-threaded time and 72-core time (with hyper-threading) for Aspen using compressed *C*-trees. For BFS, we achieve between 46–70x speedup across all inputs. For BC, our implementations achieve between 50–78x speedup across all inputs. Finally, for MIS, our implementations achieve between 60x–78x speedup across all inputs. We observe that the experiments in [117] report similar speedups for the same graphs. For local algorithms, we report the average running time for performing 2048 queries sequentially and in parallel. We achieve between 41–43x speedup for 2-hop, and between 35–49x speedup for Local-Cluster.

Flat Snapshots. Table 10.7 shows the running times of BFS with and without the use of a flat snapshot. Our BFS implementation is between 1.12–1.34x faster using a flat snapshot, including the time to compute a flat snapshot. The table also reports the time to acquire a flat snapshot, which is between 15–24% of the overall BFS time across all graphs. We observe that acquiring a flat snapshot is already an improvement for a single run of an algorithm, and quickly becomes more profitable as multiple algorithms are run over a single snapshot of the graph (e.g., multiple BFS's or betweenness centrality computations).

Graph	Without FS	With FS	Speedup	FS Time
LiveJournal	0.028	0.021	1.33	3.8e-3
com-Orkut	0.018	0.015	1.12	2.3e-3
Twitter	0.184	0.138	1.33	0.034
ClueWeb	4.98	3.69	1.34	0.779
Hyperlink2014	7.51	6.19	1.21	1.45
Hyperlink2012	18.3	14.1	1.29	3.03

Table 10.7: 72-core with hyper-threading running times (in seconds) comparing the performance of BFS without flat snapshots (**Without FS**) and with flat snapshots (**With FS**), as well as the running time for computing the flat snapshot (**FS Time**).

Graph	Update		Query (BFS)	
	Edges/sec	Latency	Latency (C)	Latency (I)
LiveJournal	7.86e4	1.27e-5	0.0190	0.0185
com-Orkut	6.02e4	1.66e-5	0.0179	0.0176
Twitter	4.44e4	1.73e-5	0.155	0.155
ClueWeb	2.06e4	4.83e-5	4.83	4.82
Hyperlink2014	1.42e4	7.04e-5	6.17	6.15
Hyperlink2012	1.16e4	8.57e-5	15.8	15.5

Table 10.8: Throughput and average latency achieved by Aspen when concurrently processing a sequential stream of edge updates along with a sequential stream of breadth-first search queries (each BFS is internally parallel). **Latency (C)** reports the average latency of the query when running the updates and queries concurrently, while **Latency (I)** reports the average latency when running queries in isolation on the modified graph.

10.4.3 Simultaneous Updates and Queries

In this sub-section, we experimentally verify that Aspen can support low-latency queries and updates running concurrently. In these experiments, we generate an update stream by randomly sampling 2 million edges from the input graph to use as updates. We sub-sample 90% of the sample to use as edge insertions, and immediately delete them from the input graph. The remaining 10% are kept in the graph, as we will delete them over the course of the update stream. The update stream is a random permutation of these insertions and deletions. We believe that sampling edges from the input graph better preserves the properties of the graph and ensures that edge deletions perform non-trivial work, compared to using random edge updates.

After constructing the update stream, we spawn two parallel jobs, one which performs the updates sequentially and one which performs global queries. We maintain the undirectedness of the graph by inserting each edge as two directed edge updates, within a single batch. For global queries, we run a stream of BFS's from random sources one after the other and measure the average latency. We note that for the BFS queries, as our inputs

are symmetrized, a random vertex is likely to fall in the giant connected component which exists in all of our input graphs. The global queries therefore process nearly all of the vertices and edges.

Table 10.8 shows the throughput in terms of directed edge updates per second, the average latency to make an undirected edge visible, and the latency of global queries both when running concurrently with updates and when running in isolation. We note that when running global queries in isolation, we use all of the threads in the system (72-cores with hyper-threading). We observe that our data structure achieves between 22–157 thousand directed edge updates per second, which is achieved while concurrently running a parallel query on all remaining threads. We obtain higher update rates on smaller graphs, where the small size of the graph enables it to utilize the caches better. In all cases, the average latency for making an edge visible is at most 86 microseconds, and is as low as 12.7 microseconds on the smallest graph.

The last two columns in Table 10.8 show the average latency of BFS queries from random sources when running queries concurrently with updates, and when running queries in isolation. We see that the performance impact of running updates concurrently with queries is less than 3%, which could be due to having one fewer thread. We ran a similar experiment, where we ran updates on 1 core and ran multiple concurrent local queries (Local-Cluster) on the remaining cores, and found that the difference in average query times is even lower than for BFS.

10.4.4 Performance of Batch Updates

In this sub-section, we show that the batch versions of our primitives achieve high throughput when updating the graph, even on very large graphs and for very large batches. As there are insufficient edges on our smaller graphs for applying the methodology from Section 10.4.3, we sample directed edges from an rMAT generator [95] with $a = 0.5, b = c = 0.1, d = 0.3$ to perform the updates. To evaluate our performance on a batch of size B, we generate B directed edge updates from the stream (note that there can be duplicates), repeatedly call InsertEdges and DeleteEdges on the batch, and report the median of three such trials. The costs that we report *include* the time to sort the batch and combine duplicates.

Table 10.9 shows the throughput (the number of edges processed per second) of performing batch edge insertions in parallel on varying batch sizes. The throughput for edge deletions are within 10% of the edge insertion times, and are usually faster (see Figure 10.2). The running time can be calculated by dividing the batch size by the throughput. We illustrate the throughput obtained for both insertions and deletions in Figure 10.2 for the largest and smallest graph, and note that the lines for other graphs are sandwiched between these two lines. The only exception of com-Orkut, where batch

Graph	Batch Size					
	10	10^3	10^5	10^7	10^9	$2 \cdot 10^9$
LiveJournal	8.26e4	2.88e6	2.29e7	1.56e8	4.13e8	4.31e8
com-Orkut	7.14e4	2.79e6	2.22e7	1.51e8	4.21e8	4.42e8
Twitter	6.32e4	2.63e6	1.23e7	5.68e7	3.04e8	3.15e8
ClueWeb	6.57e4	2.38e6	7.19e6	2.64e7	1.33e8	1.69e8
Hyperlink2014	6.17e4	2.12e6	6.66e6	2.28e7	9.90e7	1.39e8
Hyperlink2012	6.45e4	2.04e6	4.97e6	1.84e7	8.26e7	1.05e8

Table 10.9: Throughput (directed edges/second) obtained when performing parallel batch edge insertions on different graphs with varying batch sizes, where inserted edges are sampled from an rMAT graph generator. We note that the times for batch deletions are similar to the time for insertions. All times are on 72 cores with hyper-threading.

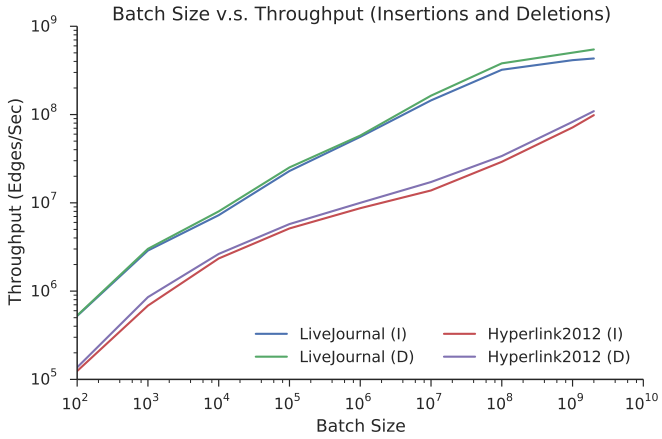


Figure 10.2: Throughput (edges/sec) when performing batches of insertions (I) and deletions (D) with varying batch sizes on Hyperlink2012 and LiveJournal in a log-log scale. All times are on 72 cores with hyper-threading.

insertions achieve about 2% higher throughput than soc-LiveJournal at the two largest batch sizes.

We observe that Aspen's throughput seems to vary depending on the graph size. We achieve a maximum throughput of 442M updates per second on com-Orkut when processing batches of 2B updates. On the other hand, on the Hyperlink2012 graph, the largest graph that we tested on, we achieve 105M updates per second for this batch size. We believe that the primary reason that small graphs achieve much better throughput at the largest batch size is that nearly all of the vertices in the tree are updated for the small graphs. In this case, due to the asymptotic work bound for the update algorithm, the work for our updates become essentially linear in the tree size.

Graph	ST	LL	Ligra+	Aspen	ST/Asp.	LL/Asp.	L+/Asp.
LiveJournal	4.98	1.12	0.246	0.582	8.55x	1.92x	0.422x
com-Orkut	10.2	3.13	0.497	0.893	11.4x	3.5x	0.55x
Twitter	81.8	31.4	5.1	9.42	8.6x	3.3x	0.54x
ClueWeb	–	–	100	200	–	–	0.50x
Hyperlink2014	–	–	184	363	–	–	0.50x
Hyperlink2012	–	–	351	702	–	–	0.50x

Table 10.10: The first four columns show the memory in gigabytes required to represent the graph using Stinger (**ST**), LLAMA (**LL**), Ligra+, and Aspen respectively. **ST/A**, **LL/A**, and **L+/A** is the amount of memory used by Stinger, LLAMA, and Ligra+ divided by the memory used by Aspen respectively. Stinger and LLAMA do not support compression and were not able to store the largest graphs used in our experiments.

10.4.5 Comparison with Stinger

In this sub-section, we compare Aspen to Stinger [130], a state-of-the-art graph-streaming system.

Stinger Design. Stinger's data structure for processing streaming graphs is based on adapting the CSR format to support dynamic updates. Instead of storing all edges of a vertex contiguously, it chunks the edges into a number of blocks, which are chained together as a linked list. Updates traverse the list to find an empty slot for a new edge, or to determine whether an edge exists. Therefore, updates take $O(deg(v))$ work and depth for a vertex v that is updated. Furthermore, updates use fine-grained locking to perform edge insertions, which may result in contention when updating very high degree vertices. As Stinger does not support compressed graph inputs, we were unable to run the system on our input graphs that are larger than Twitter.

Memory Usage. We list the sizes of the three graphs that Stinger was able to process in Table 10.10. The Stinger interface supports a function which returns the size of its in-memory representation in bytes, which is what we use to report the numbers in this chapter.

We found that Stinger has a high memory usage, even in the memory-efficient settings used in our experiments. The memory usage we observed appears to be consistent with [130], which reports that the system requires 313GB of memory to store a scale-free (RMAT) graph with 268 million vertices and 2.15 billion edges, making the cost 145 bytes per edge. This number is on the same order of magnitude as the numbers we report in Table 10.10. We found that Aspen is between 8.5–11.4x more memory efficient than Stinger.

Batch Update Performance. We measure the batch update performance of Stinger by using an rMAT generator provided in Stinger to generate the directed updates. We set $n = 2^{30}$ for updates in the stream. The largest batch size supported by Stinger is 2M

Batch Size	Stinger	Updates/sec	Aspen	Updates/sec
10	0.0232	431	9.74e-5	102,669
10^2	0.0262	3,816	2.49e-4	401,606
10^3	0.0363	27,548	6.98e-4	1.43M
10^4	0.171	58,479	2.01e-3	4.97M
10^5	0.497	201,207	9.53e-3	10.4M
10^6	3.31	302,114	0.0226	44.2M
$2 \cdot 10^6$	6.27	318,979	0.0279	71.6M

Table 10.11: Running times and update rates (directed edges/second) for Stinger and Aspen when performing batch edge updates on an empty graph with varying batch sizes. Inserted edges are sampled from the RMAT graph generator. All times are on 72 cores with hyper-threading.

directed updates. The update times for Stinger were fastest when inserting into nearly-empty graphs. For each batch size, we insert 10 batches of edges of that size into the graph, and report the median time.

The results in Table 10.11 show the update rates for inserting directed edge updates in Stinger and Aspen. We observe that the running time for Stinger is reasonably high, even on very small batches, and grows linearly with the size of the batch. The Aspen update times also grow linearly, but are very fast for small batches. Perhaps surprisingly, our update time on a batch of 1M updates is faster than the update time of Stinger on a batch of 10 edges.

Algorithm Performance. Lastly, we show the performance of graph algorithms implemented using the Stinger data structures. We use the BFS implementation for Stinger developed in McColl et al. [234]. We used a BC implementation that is available in the Stinger code base. Unfortunately, this implementation is entirely sequential, and so we compare Stinger's BC time to our single-threaded time. Neither of the Stinger implementations perform direction-optimization, so to perform a fair comparison, we used an implementation of BFS and BC in Aspen that disables direction-optimization. Table 10.12 shows the parallel running times of of Stinger and Aspen for these problems. For BFS, which is run in parallel, we achieve between 6.7–10.2x speedup over Stinger. For BC, which is run sequentially, we achieve between 1.8–4.2x speedup over Stinger. A likely reason that Aspen's BFS is significantly faster than Stinger's is that it can process edges incident to high-degree vertices in parallel, whereas traversing a vertex's neighbors in Stinger requires sequentially traversing a linked list of blocks.

10.4.6 Comparison with LLAMA

In this sub-section, we compare Aspen to LLAMA [226], another state-of-the-art graph-streaming system.

App.	Graph	ST	LL	A	A(1)	A†	ST/A	LL/A
BFS	LiveJournal	0.478	0.161	0.047	–	0.021	10.2	3.42
	com-Orkut	0.548	0.192	0.067	–	0.015	8.18	2.86
	Twitter	6.99	8.09	1.03	–	0.138	6.79	7.85
BC	LiveJournal	18.7	0.408	0.105	5.45	0.075	3.43	3.88
	com-Orkut	32.8	1.32	0.160	7.74	0.078	4.23	8.25
	Twitter	223	53.1	3.52	122	1.18	1.82	15.1

Table 10.12: Running times (in seconds) comparing the performance of algorithms implemented in Stinger (**ST**), LLAMA (**LL**), and Aspen. **A** is the parallel time using Aspen *without direction-optimization*. **(A(1))** is the one-thread time of Aspen, which is only relevant for comparing with Stinger's BC implementation. **A**† is the parallel time using Aspen *with direction-optimization*. **(ST/A)** is Aspen's speedup over Stinger and **(LL/A)** is Aspen's speedup over LLAMA.

LLAMA Design. Like Stinger, LLAMA's streaming graph data structure is motivated by the CSR format. However, like Aspen, LLAMA is designed for batch-processing in the single-writer multi-reader setting and can provide serializable snapshots. In LLAMA, a batch of size k generates a new snapshot which uses $O(n)$ space to store a vertex array, and $O(k)$ space to store edge updates in a dynamic CSR structure. The structure creates a linked list over the edges incident to a vertex that is linked over multiple snapshots. This design can cause the depth of iterating over the neighbors of a vertex to be large if the edges are spread over multiple snapshots.

Unfortunately, the publicly-available code for LLAMA does not provide support for evaluating streaming graph algorithms or batch updates. However, we we were able to load static graphs and run several implementations of algorithms in LLAMA for which we report times in this section. As LLAMA does not support compressed graph inputs, we were unable to run the system on our input graphs that are larger than Twitter.

Memory Usage. Unfortunately, we were not able to get LLAMA's internal allocator to report correct memory usage statistics for its internal allocations. Instead, we measured the lifetime memory usage of the process and use this as an estimate for the size of the in-memory data structure built by LLAMA. The memory usage in bytes for the three graphs that LLAMA was able to process is shown in Table 10.10. The cost in terms of bytes/edge for LLAMA appears to be consistent, which matches the fact that the internal representation is a flat CSR, since there is a single snapshot. Overall, Aspen is between 1.9–3.5x more memory efficient than LLAMA.

Algorithm Performance. We measured the performance of a parallel breadth-first search (BFS) and single-source betweenness centrality (BC) algorithms in LLAMA. The same source is used for both LLAMA and Aspen for both BFS and BC. BFS and BC in LLAMA do not use direction-optimization, and so we report our times for these algorithms without using direction-optimization to ensure a fair comparison.

Application	LiveJournal			com-Orkut			Twitter		
	L	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$	**L**	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$	**L**	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$
BFS	0.015	0.021	1.40x	0.012	0.015	1.25x	0.081	0.138	1.70x
BC	0.052	0.075	1.44x	0.062	0.078	1.25x	0.937	1.18	1.25x
MIS	0.032	0.054	1.68x	0.044	0.069	1.56x	0.704	0.99	1.40x
2-hop	3.06e-4	3.45e-4	1.13x	2.12e-4	2.52e-4	1.18x	2.79e-3	7.79e-3	2.79x
Local-Cluster	0.031	0.058	1.87x	0.046	0.097	2.10x	0.037	0.094	2.54x

Table 10.13: Running times (in seconds) of algorithms over small symmetric graph inputs on a 72-core machine (with hyper-threading) where **L** is the parallel time using Ligra+, **A** is the parallel time using Aspen, and $\frac{\mathbf{A}}{\mathbf{L}}$ is the slowdown incurred by Aspen. All times are measured using 72 cores using hyper-threading.

Application	ClueWeb			Hyperlink2014			Hyperlink2012		
	textbfL	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$	**L**	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$	**L**	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$
BFS	1.68	3.69	2.19x	3.44	6.19	1.79x	8.48	14.1	1.66x
BC	14.7	21.8	1.48x	17.8	24.5	1.37x	37.1	58.1	1.56x
MIS	8.14	12.1	1.48x	14.2	22.2	1.56x	32.2	50.8	1.57x
2-hop	0.024	0.028	1.16x	0.036	0.038	1.05x	0.072	0.075	1.04x
Local-Cluster	0.013	0.020	1.53x	0.013	0.021	1.61x	0.016	0.024	1.50x

Table 10.14: Running times (in seconds) of our algorithms over large symmetric graph inputs on a 72-core machine (with hyper-threading) where **L** is the parallel time using Ligra+, **A** is the parallel time using Aspen, and $\frac{\mathbf{A}}{\mathbf{L}}$ is the slowdown incurred by Aspen. All times are measured using 72 cores using hyper-threading.

Table 10.12 shows the running times for BFS and BC. We achieve between 2.8–7.8x speedup over LLAMA for BFS and between 3.8–15.1x speedup over LLAMA for BC. LLAMA's poor performance on these graphs, especially Twitter, is likely due to sequentially exploring the out-edges of a vertex in the search, which is slow on graphs with high degrees.

10.4.7 Static Graph Processing Systems

We compared Aspen to Ligra+, a state-of-the-art shared-memory graph processing system, GAP, a state-of-the-art graph processing benchmark [45], and Galois, a shared-memory parallel programming library for C++ [259].

Ligra+. Table 10.15 the parallel running times of our three global algorithms expressed using Aspen and Ligra+. The results show that Ligra is 1.43x faster than Aspen for global algorithms on our small inputs. We also performed a more extensive experimental comparison between Aspen and Ligra+, comparing the parallel running times of all of

App.	Graph	GAP	Galois	Ligra+	Aspen	$\frac{\text{GAP}}{\text{A}}$	$\frac{\text{GAL}}{\text{A}}$	$\frac{\text{L+}}{\text{A}}$
BFS	LiveJ	0.0238	0.0761	0.015	0.021	1.1x	3.6x	0.71x
	Orkut	0.0180	0.0661	0.012	0.015	1.2x	4.4x	0.80x
	Twitter	0.139	0.461	0.081	0.138	1.0x	3.3x	0.58x
BC	LiveJ	0.0930	–	0.052	0.075	1.24x	–	0.69x
	Orkut	0.107	–	0.062	0.078	1.72x	–	0.79x
	Twitter	2.62	–	0.937	1.18	2.22x	–	0.79x
MIS	LiveJ	–	1.65	0.032	0.054	–	30x	0.59x
	Orkut	–	1.52	0.044	0.069	–	22x	0.63x
	Twitter	–	8.92	0.704	0.99	–	9.0x	0.71x

Table 10.15: Running times (in seconds) comparing the performance of algorithms implemented in GAP, Galois, Ligra+, and Aspen. $\frac{\text{GAP}}{\text{A}}$, $\frac{\text{GAL}}{\text{A}}$, and $\frac{\text{L+}}{\text{A}}$ are Aspen's speedups over GAP, Galois, and Ligra+ respectively.

our algorithms on all of our inputs (Tables 10.13 and 10.14). Compared to Ligra+, across all inputs, algorithms in Aspen are 1.51x slower on average (between 1.2x–1.7x) for the global algorithms, and 1.45x slower on average (between 1.0–2.1x) for the local algorithms. We report the local times in Tables 10.13 and 10.14. The local algorithms have a modest slowdown compared to their Ligra+ counterparts, due to logarithmic work vertex accesses being amortized against the relative high average degrees (see Table 10.1).

GAP. Table 10.15 shows the parallel running times of the BFS and BC implementations from GAP. On average, our implementations in Aspen are 1.4x faster than the implementations from GAP over all problems and graphs. We note that the code in GAP has been hand-optimized using OpenMP scheduling primitives. As a result, the GAP code is significantly more complex than our code, which only uses the high-level primitives defined by Ligra+.

Galois. Table 10.15 shows the running times of using Galois, a shared-memory parallel programming library that provides support for graph processing [259]. Galois' algorithms (e.g., for BFS and MIS) come with several versions. In our experiments, we tried all versions of their algorithms, and report times for the fastest one. On average, our implementations in Aspen are 12x faster than Galois. For BFS, Aspen is between 3.3–4.4x faster than Galois. We note that the Galois BFS implementation is synchronous, and does not appear to use Beamer's direction-optimization. We omit BC as we were not able to obtain reasonable numbers on our inputs using their publicly-available code (the numbers we obtained were much worse than the ones reported in [259]). For MIS, our implementations are between 9–30x faster than Galois.

10.5 Related Work

We have mentioned some other schemes for chunking in Section 9.2.3. Although we use functional trees to support snapshots, many other systems for supporting persistence and snapshots use version lists [46, 286, 127]. The idea is for each mutable value or pointer to keep a timestamped list of versions, and reading a structure to go through the list to find the right one (typically the most current is kept first). LLAMA [226] uses a variation of this idea. However, it seems challenging to achieve the low space that we achieve using such systems since the space for such a list is large.

10.5.1 Graph Processing Frameworks

Many processing frameworks have been designed to process static graphs (e.g. [111, 281, 269, 363, 229, 154, 223, 259, 319], among many others). We refer the reader to [235, 377] for surveys of existing frameworks. Similar to Ligra+ [322], Log(Graph) [56] supports running parallel algorithms on compressed graphs. Their experiments show that they have a moderate performance slowdown on real-world graphs, but sometimes get improved performance on synthetic graphs [56].

Existing dynamic graph streaming frameworks can be divided into two categories based on their approach to ingesting updates. The first category processes updates and queries in phases, i.e., updates wait for queries to finish before updating the graph, and queries wait for updates to finish before viewing the graph. Most existing systems take this approach, as it allows updates to mutate the underlying graph without worrying about the consistency of queries [130, 139, 156, 371, 22, 309, 307, 306, 253, 91, 359, 344, 314, 89]. Hornet [89], one of the most recent systems in this category, reports a throughput of up to 800 million edges per second on a GPU with 3,840 cores (about twice our throughput using 72 CPU cores for similarly-sized graphs); however the graphs used in Hornet are much smaller that what Aspen can handle due to memory limitations of GPUs. The second category enables queries and updates to run concurrently by isolating queries to run on snapshots and periodically have updates generate new snapshots [100, 226, 185, 184].

GraphOne [206] is a system developed concurrently with our work that can handle queries running on the most recent version of the graph while updates are running concurrently by using a combination of an adjacency list and an edge list. They report an update rate of about 66.4 million edges per second on a Twitter graph with 2B edges using 28 cores; Aspen is able to ingest 94.5 million edges per second on a larger Twitter graph using 28 cores. However, GraphOne also backs up the update data to disk for durability.

There are also many systems that have been built for analyzing graphs over time [200, 164, 201, 244, 245, 168, 145, 291, 348, 360]. These systems are similar to processing dynamic graph streams in that updates to the graph must become visible to new queries, but are different in that queries can performed on the graph as it appeared at any point in time.

Although we do not explore historical queries in this thesis, functional data structures are particularly well-suited for this scenario since it is easy to keep any number of persistent versions simply by keeping their roots.

10.5.2 Graph Databases

There has been significant research on graph databases (e.g., [88, 198, 310, 280, 205, 129, 257]). The main difference between processing dynamic graph-streams and graph databases is that graph databases support transactions, i.e., multi-writer concurrency. A graph database running with snapshot isolation could be used to solve the same problem we solve. However, due to their need to support transactions, graph databases have significant overhead even for graph analytic queries such as PageRank and shortest paths. McColl et al. [234] show that Stinger is orders of magnitude faster than state-of-the-art graph databases.

10.6 Discussion

We have presented a compressed fully-functional tree data structured called the *C*-tree that has theoretically-efficient operations, low space usage, and good cache locality. We use *C*-trees to represent graphs, and design a graph-streaming framework called Aspen that is able to support concurrent queries and updates to the graph with low latency. Experiments show that Aspen outperforms state-of-the-art graph-streaming frameworks, STINGER and LLAMA, and only incurs a modest overhead over state-of-the-art static graph processing frameworks. Future work includes designing incremental graph algorithms and historical queries using Aspen, and using *C*-trees in other applications.

Although our original motivation for designing *C*-trees was for representing compressed graphs, we believe that they are of independent interest and can be used in applications where ordered sets of integers are dynamically maintained, such as compressed inverted indices in search engines.

Part IV

Conclusion and Future Directions

11 Conclusion and Future Work

11.1 Conclusion

The use of graphs and graph algorithms to model and reason about data has seen a huge investment of effort from both theoretical and practical communities over the past decade. This thesis contributes to this research effort in Chapters 3,4, and 5 by developing graph processing tools and algorithms that enable users to easily, quickly, and cost-effectively process massive real-world graphs using multicore machines. Our results include a number of new practical and theoretically-efficient graph algorithms for a broad set of fundamental graph problems, ranging from shortest path problems, connectivity problems, covering problems, and substructure problems. All of the parallel graph algorithm implementations developed in this thesis have been made publicly available as part of the Graph Based Benchmark Suite (GBBS). Then, in Chapter 6, we discussed how to extend our approach to graphs stored on non-volatile memory, and designed a new graph processing system called Sage for this setting. Taken together, these results have illustrated the power of using theoretically-efficient shared-memory algorithms by showing that both our shared-memory and non-volatile memory implementations can solve a broad class of problems on the largest publicly-available graph, the WebDataCommons hyperlink graph, with over 200 billion edges, in just seconds to minutes.

This thesis also developed some of the first provably-efficient results for parallel graph processing on evolving graphs which change over time. Specifically, we have designed efficient parallel algorithms for dynamic settings which take advantage of batching. In Chapter 7, we designed theoretically and practically efficient parallel batch-dynamic algorithm for the fundamental forest connectivity problem by adapting the classic Euler tour tree data structure to the batch-dynamic setting. Our Euler tour tree structure was based on using a new phase-concurrent data structure for batch-dynamic sequences based on skip-lists which may be of independent interest. Finally, in Chapter 8, we designed the first theoretically-efficient parallel batch-dynamic algorithm for connectivity. Our algorithm adapts a classic dynamic graph algorithm by Holm, de Lichtenberg, and Thorup, and utilizes our batch-dynamic algorithm for forest connectivity as a crucial sub-routine.

Finally, in this thesis we have designed new provably-efficient data structures and the Aspen graph-streaming system for representing and processing evolving graphs. In Chapter 9, we first presented an approach to streaming graph processing based on representing graphs using purely-functional trees. To address the space-inefficiency and cache-inefficiencies of simply using nested purely-functional trees, we designed the C-tree data structure, which is a new type of compressed purely-functional tree. We showed how

to implement a broad set of useful primitives on C-trees, and designed parallel batch-update algorithms for C-trees that have strong provable bounds on their work and depth and are also fast in practice. Then, in Chapter 10 we introduced the using Aspen graph-streaming system, which uses a nested purely-functional graph data structure based on C-trees. We studied the cost of parallel batch-updates to this graph representation and showed that our update algorithms achieve good bounds in theory and achieve update rates of up to several hundred million edges per second in practice. Using Aspen, we showed that we can update massive evolving graphs, including the WebDataCommons hyperlink graph with over 200 billion edges in the main memory of a single multicore machine while concurrently running graph analytics on the graph.

11.2 Future Work

There are many interesting directions for future work stemming from the work in this thesis.

Compilation-Based and Optimizing Graph Processing Frameworks. An interesting direction stemming from this work is to integrate algorithms and techniques designed in this thesis into *compilation-based* frameworks such as GraphIt [383], and to also to explore the parameter space of algorithm design decisions for these algorithms in *optimizing* frameworks. For example, since the publication of our work on Julienne, we showed how to implement ordered graph algorithms such as k-core, Δ-stepping and parallel approximate set-cover using a compilation-based approach in the GraphIt system [384]. Implementing the remaining algorithms developed in this thesis in GraphIt would enable these implementations to take advantage of the NUMA optimizations and scheduling optimizations implemented by GraphIt.

In terms of optimizing frameworks, in recent work we have designed a system called ConnectIt which allows users to explore the space of design decisions for parallel algorithms for the connected components problem and related problems [118]. Exploring this space enabled us to obtain a parallel connectivity algorithm which processes the WebDataCommons hyperlink graph in *under 10 seconds*, improving the previous state-of-the-art result (from this thesis) by 3.1x. In future work, it could be interesting to design similar frameworks for other problems studied in this thesis.

Graph Clustering. One important application area of graph algorithms is graph clustering. Graph clustering algorithms take as input a graph where the vertices represent objects to be clustered and group vertices that are deemed similar based on properties of the graph together into the same cluster. Example algorithms include the Louvain and Leiden algorithms which maximize modularity [352], correlation clustering [40, 272], many variants of hierarchical agglomerative clustering (HAC) [230], among many other graph clustering algorithms [376, 368, 33].

Despite significant real-world interest in this area, existing implementations of parallel graph clustering algorithms suffer from the same kinds of problems faced by graph algorithm implementations described in this work, namely they do not scale well to large datasets with billions of vertices and hundreds of billions of edges and they usually do not have good provable bounds on their theoretical costs.

Therefore, an interesting direction is to study parallel clustering algorithms both in theory and in practice. Certain agglomerative methods in this area seem inherently difficult to parallelize (and many papers have informally stated that this is the case). It would be interesting to study these problems to determine their parallel complexity, and in cases where they are P-complete (or fall into a similar class outside of NC), derive practical approximation algorithms for these problems with low work and depth.

Optimal Graph Algorithms in the Binary-Forking Model. The binary-forking (BF) model [72, 62], was being developed at CMU during the time this thesis was being written, and all of the algorithms in this thesis are primarily analyzed in this model. Although this thesis has developed a large body of practical work-efficient parallel algorithms in the BF model, there are still many interesting open questions that remain for future work.

First, although many *optimal* graph algorithms have been developed for different PRAM models, perhaps most notably for the connectivity problem [148, 162, 163], these algorithms all require $O(\lg^2 n)$ depth in the BF model. Thus, a natural question is whether we can design work-efficient $O(\lg n)$ depth BF algorithms for graph problems such as connectivity and spanning forest. Even ignoring work-efficiency, this is an interesting question, since nearly every parallel connectivity algorithm seems to require super-constant (usually $O(\lg n)$) rounds of synchronization which leads to super-logarithmic depth in the BF model. Can we solve the approximate densest subgraph problem, which uses $O(\lg n)$ rounds of peeling in Algorithm 19 from Chapter 5 in $O(\lg n)$ depth in the BF model? Can we develop a theory that explains when synchronization may be necessary?

Second, can we prove work-depth tradeoffs for problems suffering from the transitive closure bottleneck such as directed reachability and shortest path problems in the BF model, or in a space-restricted version of the model? This is a longstanding open problem in parallel algorithms, but recently there has been a flurry of progress on nearly-work efficient algorithms for both reachability [140, 92, 93] and approximate SSSP problems [24, 216, 93]. Yet, the depth bounds for exact solutions to these problems in NC are all highly work-inefficient. Can we develop hardness results (even conditional ones) that rule out exact nearly-linear work, poly-logarithmic depth reachability and shortest path algorithms?

Implementing Parallel Batch-Dynamic Algorithms. Another important next step stemming from this work is to implement and experimentally evaluate parallel batch-dynamic algorithms for problems. A significant issue in this setting is the lack of high quality dynamic graph datasets, at least compared to the relative abundance of medium and large-sized static graph datasets (e.g., see the SNAP [215] and LAW datasets [81]).

There is a historic dearth of good experimental results on dynamic graph algorithms with most of the work in the area focusing primarily in theoretical results, although in recent years some groups have been making inspiring progress on the experimental front [171, 166, 167].

In terms of future directions, it would be interesting to understand whether the parallel batch-dynamic connectivity algorithm from Chapter 8 is practical. If this algorithm is impractical, one approach may be to show that existing heuristics for dynamic connectivity like that of McColl et al. [233] are actually theoretically-efficient for certain input distributions (e.g., graphs whose update sequences follow a power-law distribution), or to design practical algorithms that are specifically designed for graphs with power-law distributions, or which admit good clustering structure. This approach would yield beyond worst-case analyses of dynamic graph connectivity, which would be interesting even in the sequential setting without considering parallelism or batching.

Finally, it would be interesting to study parallel batch-dynamic algorithms for other problems, including triangle counting, k-clique counting[1], maximal independent set, maximal matching. It would also be interesting to study practically-motivated problems, such as batch-dynamic graph clustering and understand whether agglomerative clustering, or some suitable approximation of agglomerative clustering can be efficiently maintained under dynamic updates.

Graph Streaming Systems and Databases. There are several important next steps that stem from our work on graph-streaming systems. Firstly, can we obtains bounds similar to those we showed for updates to C-trees for *deterministic* update algorithms? Since a C-tree is randomized by design, such an approach would require rethinking how compression is done to obtain deterministic bounds. Secondly, can we improve the constant factors in the bounds? Currently our results have a quadratic dependence on the chunking parameter, b, but only a linear dependence seems necessary. It would be interesting to both show this upper-bound, and prove a matching lower-bound in a reasonable model of computation, such as a pointer-machine [49]. Thirdly, outside of faster update algorithms, we should extend Aspen to support a broad set of features, including arbitrary edge and node attributes. Although these features may seem like bells-and-whistles that only complicate the implementation, building this kind of functionality is important for the system to be useful to a broader set of users, hobbyists, and also industrial applications.

Looking forwards, developing features such as arbitrary attributes would be the first step towards developing a graph database based on Aspen. An important challenge facing this work will be to efficiently support transactions. Although this problem been very well studied in the database literature [155, 250, 251], there may be new opportunities in the context of graphs: for example, there is an intriguing prospect of designing efficient transaction-processing algorithms specially designed for graph update streams. A hopeful

[1]Some of our ongoing work [119] makes progress for triangle and k-clique counting.

vision in this direction is a hybrid transaction and analytics graph database system that can efficiently apply graph algorithms on read-only snapshots, while enabling fast (potentially batched) transactions, all with strong provable-bounds on their costs. I believe that the algorithms and systems developed in this thesis are a promising first step toward building such a system.

Bibliography

[1] James Abello, Adam L. Buchsbaum, and Jeffery R. Westbrook. "A Functional Approach to External Graph Algorithms". In: *Algorithmica* 32.3 (2002), pp. 437–458.

[2] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996. ISBN: 0-262-01153-0.

[3] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. "EmptyHeaded: A Relational Engine for Graph Processing". In: *ACM Trans. Database Syst.* 42.4 (2017), 20:1–20:44.

[4] Umut A Acar, Vitaly Aksenov, and Sam Westrick. "Brief Announcement: Parallel Dynamic Tree Contraction via Self-Adjusting Computation". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2017.

[5] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. "Theory and Practice of Chunked Sequences". In: *European Symposium on Algorithms (ESA)*. 2014, pp. 25–36.

[6] Umut A Acar, Guy Edward Blelloch, Robert Endre Harper, Jorge Vittes, and Shan Leung Maverick Woo. "Dynamizing Static Algorithms, with Applications to Dynamic Trees and History Independence". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2004, pp. 531–540.

[7] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. "Parallel Batch-Dynamic Graph Connectivity". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2019, pp. 381–392.

[8] Umut A Acar, Daniel Anderson, Guy E Blelloch, and Laxman Dhulipala. "Parallel Batch-Dynamic Graph Connectivity". In: *CoRR* abs/1903.08794 (2019).

[9] Umut A Acar, Andrew Cotter, Benoit Hudson, and Duru Türkoglu. "Parallelism in dynamic well-spaced point sets". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2011.

[10] Dimitris Achlioptas and Michael Molloy. "The Solution Space Geometry of Random Linear Equations". In: *Random Structures & Algorithms* 46.2 (2015).

[11] Alok Aggarwal, Richard J Anderson, and M-Y Kao. "Parallel Depth-First Search in General Directed Graphs". In: *ACM Symposium on Theory of Computing (STOC)*. 1989, pp. 297–308.

[12] Alok Aggarwal and Jeffrey S. Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Commun. ACM* 31.9 (1988), pp. 1116–1127.

[13] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. "Crono: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores". In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2015, pp. 44–55.

[14] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. "Analyzing Graph Structure via Linear Measurements". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2012, pp. 459–467.

[15] Yaroslav Akhremtsev and Peter Sanders. "Fast Parallel Operations on Search Trees". In: *IEEE International Conference on High-Performance Computing (HiPC)*. 2016, pp. 291–300.

[16] N. Alon, R. Yuster, and U. Zwick. "Finding and Counting Given Length Cycles". In: *Algorithmica* 17.3 (1997), pp. 209–223.

[17] Noga Alon, László Babai, and Alon Itai. "A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem". In: *J. Algorithms* 7.4 (1986), pp. 567–583.

[18] Mohammad Alshboul, Hussein Elnawawy, Reem Elkhouly, Keiji Kimura, James Tuck, and Yan Solihin. "Efficient Checkpointing with Recompute Scheme for Non-volatile Main Memory". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 16.2 (2019), p. 18.

[19] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Maintaining Information in Fully Dynamic Trees with Top Trees". In: *ACM Transactions on Algorithms* 1.2 (2005), pp. 243–264.

[20] J. I. Alvarez-Hamelin, Luca Dall'asta, Alain Barrat, and Alessandro Vespignani. "Large Scale Networks Fingerprinting and Visualization using the k-core Decomposition". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2005, pp. 41–50.

[21] Khaled Ammar and M Tamer Özsu. "Experimental Analysis of Distributed Graph Systems". In: *Proc. VLDB Endow.* 11.10 (2018), pp. 1151–1164.

[22] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. "Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-memory Dataflows". In: *Proc. VLDB Endow.* 11.6 (2018), pp. 691–704.

[23] Richard Anderson and Ernst W. Mayr. *A* P*-complete Problem and Approximations to It.* Tech. rep. Stanford University, 1984.

[24] Alexandr Andoni, Clifford Stein, and Peilin Zhong. "Parallel Approximate Undirected Shortest Paths via Low Hop Emulators". In: *ACM Symposium on Theory of Computing (STOC)*. ACM, 2020, pp. 322–335.

[25] Alexandr Andoni, Clifford Stein, Zhao Song, Zhengyu Wang, and Peilin Zhong. "Parallel Graph Connectivity in Log Diameter Rounds". In: *IEEE Symposium on Foundations of Computer Science (FOCS)*. 2018, pp. 674–685.

[26] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. "Thread scheduling for multiprogrammed multiprocessors". In: *Theory of Computing Systems (TOCS)* 34.2 (2001), pp. 115–144.

[27] Joy Arulraj and Andrew Pavlo. "How to Build a Non-Volatile Memory Database Management System". In: *ACM International Conference on Management of Data*. 2017, pp. 1753–1758.

[28] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. "BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory". In: *Proc. VLDB Endow.* 11.5 (2018), pp. 553–565.

[29] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. "Tracking in Order to Recover: Recoverable Lock-Free Data Structures". In: *arXiv preprint arXiv:1905.13600* (2019).

[30] Baruch Awerbuch. "Complexity of Network Synchronization". In: *J. ACM* 32.4 (1985), pp. 804–823.

[31] Baruch Awerbuch and Y. Shiloach. "New Connectivity and MSF Algorithms for Ultracomputer and PRAM". In: *International Conference on Parallel Processing (ICPP)*. 1983, pp. 175–179.

[32] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. "Low-Diameter Graph Decomposition is in NC". In: *Scandinavian Workshop on Algorithm Theory*. 1992, pp. 83–93.

[33] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. "HipMCL: a High-performance Parallel Implementation of the Markov Clustering Algorithm for Large-scale Networks". In: *Nucleic acids research* 46.6 (2018), e33.

[34] David A Bader and Guojing Cong. "Fast Shared-Memory Algorithms for Computing the Minimum Spanning Forest of Sparse Graphs". In: *J. Parallel Distrib. Comput.* 66.11 (2006), pp. 1366–1378.

[35] David A Bader and Kamesh Madduri. "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors". In: *IEEE International Conference on High-Performance Computing (HiPC)*. 2005, pp. 465–476.

[36] David A Bader and Kamesh Madduri. "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2". In: *International Conference on Parallel Processing (ICPP)*. 2006, pp. 523–530.

[37] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. "Efficient Primal-Dual Graph Algorithms for MapReduce". In: *International Workshop on Algorithms and Models for the Web-Graph*. 2014, pp. 59–78.

[38] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. "Densest Subgraph in Streaming and MapReduce". In: *Proc. VLDB Endow.* 5.5 (2012), pp. 454–465.

[39] Georg Baier, Ekkehard Köhler, and Martin Skutella. "The k-Splittable Flow Problem". In: *Algorithmica* 42.3-4 (2005), pp. 231–248.

[40] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. "Correlation Clustering". In: *Machine learning* 56.1-3 (2004), pp. 89–113.

[41] Vladimir Batagelj and Matjaz Zaversnik. "An $O(m)$ Algorithm for Cores Decomposition of Networks". In: *CoRR* cs.DS/0310049 (2003).

[42] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. "Affinity Clustering: Hierarchical Clustering at Scale". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017, pp. 6864–6874.

[43] R. Bayer and E. M. McCreight. "Organization and Maintenance of Large Ordered Indexes". In: *Acta Informatica* 1.3 (1972), pp. 173–189.

[44] Scott Beamer, Krste Asanović, and David Patterson. "Direction-Optimizing Breadth-First Search". In: *Scientific Programming* 21.3-4 (2013), pp. 137–148.

[45] Scott Beamer, Krste Asanovic, and David A. Patterson. "The GAP Benchmark Suite". In: *CoRR* abs/1508.03619 (2015).

[46] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. "An Asymptotically Optimal Multiversion B-Tree". In: *The VLDB Journal* 5.4 (1996), pp. 264–275.

[47] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Łącki, Vahab Mirrokni, and Warren Schudy. "Massively Parallel Computation via Remote Memory Access". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2019, pp. 59–68.

[48] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab Mirrokni. "Near-optimal massively parallel graph connectivity". In: *IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2019, pp. 1615–1636.

[49] Amir M Ben-Amram. "What is a Pointer Machine?" In: *ACM SIGACT News* 26.2 (1995), pp. 88–95.

[50] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. "Delay-Free Concurrency on Faulty Persistent Memory". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2019, pp. 253–264.

[51] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. "Implicit Decomposition for Write-Efficient Connectivity Algorithms". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 711–722.

[52] Naama Ben-David, Guy E Blelloch, Yihan Sun, and Yuanhao Wei. "Multiversion Concurrency with Bounded Delay and Precise Garbage Collection". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2019, pp. 241–252.

[53] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. "Parallel Algorithms for Asymmetric Read-Write Costs". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2016.

[54] Bonnie Berger, John Rompel, and Peter W. Shor. "Efficient NC Algorithms for Set Cover with Applications to Learning and Geometry". In: *J. Computer and System Sciences* 49.3 (1994), pp. 454–477.

[55] Jean-Philippe Bernardy. *The Haskell Yi package*. 2008.

[56] Maciej Besta, Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh, Maurice Hoerold, and Torsten Hoefler. "Log(Graph): A Near-optimal High-performance Graph Representation". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2018, 7:1–7:13.

[57] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. "Efficient Parallel and External Matching". In: *European Conference on Parallel Processing (Euro-Par)*. 2013, pp. 659–670.

[58] Daniel K. Blandford and Guy E. Blelloch. "Compact Representations of Ordered Sets". In: *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2004, pp. 11–19.

[59] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. "An Experimental Analysis of a Compact Graph Representation". In: *Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2004, pp. 49–61.

[60] Guy E. Blelloch. "Prefix Sums and Their Applications". In: *Synthesis of Parallel Algorithms*. Ed. by John Reif. Morgan Kaufmann, 1993.

[61] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. "ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2020, 507âĂŞ–509.

[62] Guy E. Blelloch and Laxman Dhulipala. *Introduction to Parallel Algorithms*. http://www.cs.cmu.edu/~realworld/slidesS18/parallelChap.pdf. Carnegie Mellon University. 2018.

[63] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. "Just Join for Parallel Ordered Sets". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2016, pp. 253–264.

[64] Guy E Blelloch, Jeremy T Fineman, and Julian Shun. "Greedy sequential maximal independent set and matching are parallel on average". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2012, pp. 308–317.

[65] Guy E Blelloch and Yan Gu. "Improved Parallel Cache-Oblivious Algorithms for Dynamic Programming". In: *SIAM/ACM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 2020, pp. 105–119.

[66] Guy E. Blelloch, Yan Gu, and Yihan Sun. "Efficient Construction on Probabilistic Tree Embeddings". In: *Intl. Colloq. on Automata, Languages and Programming (ICALP)*. 2017, 26:1–26:14.

[67] Guy E. Blelloch, Richard Peng, and Kanat Tangwongsan. "Linear-Work Greedy Parallel Approximate Set Cover and Variants". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2011.

[68] Guy E. Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. "Parallel and I/O Efficient Set Covering Algorithms". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2012, pp. 82–90.

[69] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. "Efficient Algorithms with Asymmetric Read and Write Costs". In: *European Symposium on Algorithms (ESA)*. 2016.

[70] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. "Internally Deterministic Algorithms Can Be Fast". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2012, pp. 181–192.

[71] Guy E Blelloch, Anupam Gupta, Ioannis Koutis, Gary L Miller, Richard Peng, and Kanat Tangwongsan. "Nearly-Linear Work Parallel SDD Solvers, Low-Diameter Decomposition, and Low-Stretch Subgraphs". In: *Theory of Computing Systems* 55.3 (2014), pp. 521–554.

[72] Guy E Blelloch, Jeremy T Fineman, Yan Gu, and Yihan Sun. "Optimal Parallel Algorithms in the Binary-Forking Model". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2020, pp. 89–102.

[73] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. "Parallel Shortest Paths Using Radius Stepping". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2016, pp. 443–454.

[74] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. "Parallel Write-Efficient Algorithms and Data Structures for Computational Geometry". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2018.

[75] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. "Parallelism in Randomized Incremental Algorithms". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2016, pp. 467–478.

[76] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. "Sorting with Asymmetric Read and Write Costs". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2015.

[77] Robert D. Blumofe and Charles E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *J. ACM* 46.5 (1999), pp. 720–748.

[78] Robert D. Blumofe and Charles E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *J. ACM* 46.5 (1999), pp. 720–748.

[79] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 1995.

[80] Hans-J. Boehm, Russ Atkinson, and Michael Plass. "Ropes: An Alternative to Strings". In: *Softw. Pract. Exper.* 25.12 (1995).

[81] Paolo Boldi and Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques". In: *International World Wide Web Conference (WWW)*. 2004, pp. 595–601.

[82] Francesco Bonchi, Arijit Khan, and Lorenzo Severini. "Distance-generalized Core Decomposition". In: *ACM International Conference on Management of Data (SIGMOD)*. 2019, pp. 1006–1023.

[83] Otakar Borůvka. "O jistém problému minimálním". In: *Práce Mor. Přírodověd. Spol. v Brně III* 3 (1926), pp. 37–58.

[84] Ulrik Brandes. "A Faster Algorithm for Betweenness Centrality". In: *Journal of Mathematical Sociology* 25.2 (2001), pp. 163–177.

[85] Sergey Brin and Lawrence Page. "The Anatomy of a Large-scale Hypertextual Web Search Engine". In: *International World Wide Web Conference (WWW)*. 1998, pp. 107–117.

[86] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. "A Parallel Priority Queue with Constant Time Operations". In: *J. Parallel Distrib. Comput.* 49.1 (1998).

[87] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. "Graph Structure in the Web". In: *Computer Networks* 33.1-6 (2000), pp. 309–320.

[88] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *USENIX Annual Technical Conference (ATC)*. 2013, pp. 49–60.

[89] F. Busato, O. Green, N. Bombieri, and D. A. Bader. "Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs". In: *IEEE High Performance extreme Computing Conference (HPEC)*. 2018, pp. 1–7.

[90] Tao Cai, Fuli Chen, Qingjian He, Dejiao Niu, and Jie Wang. "The Matrix KV Storage System Based on NVM Devices". In: *Micromachines* 10.5 (2019), p. 346.

[91] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. "Facilitating Real-time Graph Mining". In: *International Workshop on Cloud Data Management (CloudDB)*. 2012, pp. 1–8.

[92] Nairen Cao, Jeremy T. Fineman, and Katina Russell. "Improved Work Span Tradeoff for Single Source Reachability and Approximate Shortest Paths". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2020, 511âĂŞ513.

[93] Nairen Cao, Jeremy T Fineman, and Katina Russell. "Improved Work Span Tradeoff for Single Source Reachability and Approximate Shortest Paths". In: *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 2020, pp. 511–513.

[94] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simhadri. "Write-Avoiding Algorithms". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016.

[95] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. "R-MAT: A Recursive Model for Graph Mining". In: *SIAM International Conference on Data Mining (SDM)*. 2004, pp. 442–446.

[96] Moses Charikar. "Greedy Approximation Algorithms for Finding Dense Components in a Graph". In: *International Workshop on Approximation Algorithms for Combinatorial Optimization*. 2000, pp. 84–95.

[97] Qichen Chen, Hyojeong Lee, Yoonhee Kim, Heon Young Yeom, and Yongseok Son. "Design and Implementation of Skiplist-Based Key-Value Store on Non-Volatile Memory". In: *Cluster Computing* 22.2 (2019), pp. 361–371.

[98] Shimin Chen, Phillip B. Gibbons, and Suman Nath. "Rethinking Database Algorithms for Phase Change Memory". In: *Conference on Innovative Data Systems Research (CIDR)*. 2011.

[99] Shimin Chen and Qin Jin. "Persistent B+-Trees in Non-Volatile Main Memory". In: *Proc. VLDB Endow.* 8.7 (2015), pp. 786–797.

[100] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. "Kineograph: Taking the Pulse of a Fast-Changing and Connected World". In: *European Conference on Computer Systems (EuroSys)*. 2012, pp. 85–98.

[101] Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. "Max-cover in Map-reduce". In: *International World Wide Web Conference (WWW)*. 2010.

[102] Edith Cohen. "Using Selective Path-Doubling for Parallel Shortest-Path Computations". In: *J. Algorithms* 22.1 (1997).

[103] Nachshon Cohen, Rachid Guerraoui, and Mihail Igor Zablotchi. "The Inherent Cost of Remembering Consistently". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2018.

[104] Richard Cole, Philip N. Klein, and Robert E. Tarjan. "Finding Minimum Spanning Forests in Logarithmic Time and Linear Work Using Random Sampling". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1996, pp. 243–250.

[105] Guojing Cong and David A Bader. "An Experimental Study of Parallel Biconnected Components Algorithms on Symmetric Multiprocessors (SMPs)". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2005, pp. 9–18.

[106] Guojing Cong and Ilie Gabriel Tanase. "Composable Locality Optimizations for Accelerating Parallel Forest Computations". In: *IEEE International Conference on High Performance Computing and Communications (HPCC)*. 2016, pp. 190–197.

[107] Don Coppersmith, Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. *A Divide-and-Conquer Algorithm for Identifying Strongly Connected Components*. Tech. rep. RC23744. IBM Research, 2003.

[108] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)* MIT Press, 2009.

[109] Andreia Correia, Pascal Felber, and Pedro Ramalhete. "Romulus: Efficient Algorithms for Persistent Transactional Memory". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2018, pp. 271–282.

[110] Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. "ParK: An Efficient Algorithm for k-Core Decomposition on Multicore Processors". In: *IEEE International Conference on Big Data (BigData)*. 2014, pp. 9–16.

[111] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. "Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2018, pp. 752–768.

[112] Andrew A. Davidson, Sean Baxter, Michael Garland, and John D. Owens. "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths". In: *IPDPS*. 2014.

[113] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (2011).

[114] Erik Demaine and Shafi Goldwasser. "6.046J/18.410J Lecture Notes on Skip Lists". University Lecture. 2004. URL: https://courses.csail.mit.edu/6.046/spring04/handouts/skiplists.pdf.

[115] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. "Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2017, pp. 293–304.

[116] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. "Low-Latency Graph Streaming using Compressed Purely-Functional Trees". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2019, pp. 918–934.

[117] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. "Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2018, pp. 293–304.

[118] Laxman Dhulipala, Changwan Hong, and Julian Shun. "ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms". In: *CoRR* abs/2008.03909 (2020).

[119] Laxman Dhulipala, Quanquan C Liu, and Julian Shun. "Parallel Batch-Dynamic k-Clique Counting". In: *arXiv preprint arXiv:2003.13585* (2020).

[120] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. "Compressing Graphs and Indexes with Recursive Graph Bisection". In: *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 2016, pp. 1535–1544.

[121] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. "Parallel Batch-Dynamic Graphs: Algorithms and Lower Bounds". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2020, pp. 1300–1319.

[122] Laxman Dhulipala, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E Blelloch, Phillip B Gibbons, and Julian Shun. "Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs". In: *arXiv preprint arXiv:1910.12310* (2020).

[123] Laxman Dhulipala, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E Blelloch, Phillip B Gibbons, and Julian Shun. "Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs". In: *Proc. VLDB Endow.* 13.9 (2020), pp. 1598–1613.

[124] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. "The Graph Based Benchmark Suite (GBBS)". In: *International Workshop on Graph Data Management Experiences and Systems (GRADES) and Network Data Analytics (NDA).* 2020, 11:1–11:8.

[125] Robert B. Dial. "Algorithm 360: Shortest-path Forest with Topological Ordering [H]". In: *Commun. ACM* 12.11 (1969).

[126] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numer. Math.* 1.1 (1959).

[127] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. "Making Data Structures Persistent". In: *J. Comput. Syst. Sci.* 38.1 (1989), pp. 86–124.

[128] Ran Duan, Kaifeng Lyu, and Yuanhang Xie. "Single-Source Bottleneck Path Algorithm Faster than Sorting for Sparse Graphs". In: *Intl. Colloq. on Automata, Languages and Programming (ICALP).* 2018, 43:1–43:14.

[129] Ayush Dubey, Greg D Hill, Robert Escriva, and Emin Gün Sirer. "Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps". In: *Proc. VLDB Endow.* 9.11 (2016), pp. 852–863.

[130] David Ediger, Robert McColl, Jason Riedy, and David A Bader. "STINGER: High Performance Data Structure for Streaming Graphs". In: *IEEE Conference on High Performance Extreme Computing (HPEC).* 2012, pp. 1–5.

[131] James A. Edwards and Uzi Vishkin. "Better Speedups Using Simpler Parallel Programming for Graph Connectivity and Biconnectivity". In: *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM).* 2012, pp. 103–114.

[132] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. "Grami: Frequent Subgraph and Pattern Mining in a Single Large Graph". In: *Proc. VLDB Endow.* 7.7 (2014), pp. 517–528.

[133] B. Elser and A. Montresor. "An Evaluation Study of BigData Frameworks for Graph Processing". In: *IEEE International Conference on Big Data (BigData).* 2013.

[134] David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Manuel R. Torres. "2-3 Cuckoo Filters for Faster Triangle Listing and Set Intersection". In: *pods.* 2017, pp. 247–260.

[135] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. "Sparsification–a Technique for Speeding up Dynamic Graph Algorithms". In: *J. ACM* 44.5 (1997), pp. 669–696.

[136] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling". In: *ACM International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376.

[137] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 1996, pp. 226–231.

[138] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. "On Graph Problems in a Semi-Streaming Model". In: *Theoretical Computer Science* 348.2-3 (2005), pp. 207–216.

[139] Guoyao Feng, Xiao Meng, and Khaled Ammar. "DISTINGER: A Distributed Graph Data Structure for Massive Dynamic Graph Processing". In: *IEEE International Conference on Big Data (BigData)*. 2015, pp. 1814–1822.

[140] Jeremy T. Fineman. "Nearly Work-Efficient Parallel Algorithm for Digraph Reachability". In: *ACM Symposium on Theory of Computing (STOC)*. 2018, pp. 457–470.

[141] Manuela Fischer and Andreas Noever. "Tight Analysis of Parallel Randomized Greedy MIS". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2018, pp. 2152–2160.

[142] Lisa K Fleischer, Bruce Hendrickson, and Ali Pinar. "On Identifying Strongly Connected Components in Parallel". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2000, pp. 505–511.

[143] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. "Implicitly Threaded Parallelism in Manticore". In: *J. Funct. Program.* 20.5-6 (2010), pp. 537–576.

[144] Lester Randolph Ford and Delbert R Fulkerson. "Maximal Flow Through a Network". In: *Classic Papers in Combinatorics*. Springer, 2009, pp. 243–248.

[145] François Fouquet, Thomas Hartmann, Sébastien Mosser, and Maxime Cordy. "Enabling Lock-Free Concurrent Workers over Temporal Graphs Composed of Multiple Time-Series". In: *ACM Symposium on Applied Computing (SAC)*. Vol. 8. 2018, pp. 1054–1061.

[146] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". In: *J. ACM* 34.3 (1987).

[147] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. "Cache-Oblivious Algorithms". In: *focs*. 1999, pp. 285–298.

[148] Hillel Gazit. "An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph". In: *SIAM J. on Computing* 20.6 (1991), pp. 1046–1067.

[149] Hillel Gazit and Gary L Miller. "An Improved Parallel Algorithm that Computes the BFS Numbering of a Directed Graph". In: *Information Processing Letters* 28.2 (1988), pp. 61–65.

[150] Joseph Gil, Yossi Matias, and Uzi Vishkin. "Towards a Theory of Nearly Constant Time Parallel Algorithms". In: *IEEE Symposium on Foundations of Computer Science (FOCS)*. 1991.

[151] Seth Gilbert and Lawrence Li Er Lu. "How Fast Can You Update Your MST?" In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2020, 531âĂŞ–533.

[152] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. "Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory". In: *Proc. VLDB Endow.* 13.8 (2020), pp. 1304–13.

[153] A. V. Goldberg. *Finding a Maximum Density Subgraph*. Tech. rep. UCB/CSD-84-171. Berkeley, CA, USA, 1984.

[154] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2012, pp. 17–30.

[155] Jim N Gray, Raymond A Lorie, and Gianfranco R Putzolu. "Granularity of Locks in a Shared Data Base". In: *Proceedings of the 1st International Conference on Very Large Data Bases*. 1975, pp. 428–451.

[156] Oded Green and David A Bader. "cuSTINGER: Supporting Dynamic Graph Algorithms for GPUs". In: *IEEE Conference on High Performance Extreme Computing (HPEC)*. 2016, pp. 1–6.

[157] Oded Green, Luis M. Munguia, and David A. Bader. "Load Balanced Clustering Coefficients". In: *Workshop on Parallel programming for Analytics Applications (PPAA)*. 2014, pp. 3–10.

[158] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-completeness Theory*. Oxford University Press, Inc., 1995. ISBN: 0-19-508591-4.

[159] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. "Making Pull-based Graph Processing Performant". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2018, pp. 246–260.

[160] Yan Gu, Yihan Sun, and Guy E. Blelloch. "Algorithmic Building Blocks for Asymmetric Memories". In: *European Symposium on Algorithms (ESA)*. 2018, 44:1–44:15.

[161] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. "A Top-Down Parallel Semisort". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2015, pp. 24–34.

[162] Shay Halperin and Uri Zwick. "An Optimal Randomized Logarithmic Time Connectivity Algorithm for the EREW PRAM". In: *J. Comput. Syst. Sci.* 53.3 (1996), pp. 395–416.

[163] Shay Halperin and Uri Zwick. "Optimal Randomized EREW PRAM Algorithms for Finding Spanning Forests". In: 39.1 (2001), pp. 1–46.

[164] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. "Chronos: a Graph Engine for Temporal Graph Analysis". In: *European Conference on Computer Systems (EuroSys)*. 2014, 1:1–1:14.

[165] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. "Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases". In: *ACM International Conference on Management of Data (SIGMOD)*. 2013, pp. 337–348.

[166] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. "Faster Fully Dynamic Transitive Closure in Practice". In: *ACM Symposium on Experimental and Efficient Algorithms (SEA)*. Vol. 160. 2020, 14:1–14:14.

[167] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. "Fully Dynamic Single-Source Reachability in Practice: An Experimental Study". In: *Algorithm Engineering and Experiments (ALENEX)*. 2020, pp. 106–119.

[168] Thomas Hartmann, Francois Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. "Analyzing Complex Data in Motion at Scale with Temporal Graphs". In: *International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 2017, pp. 596–601.

[169] William Hasenplaugh, Tim Kaler, Tao B Schardl, and Charles E Leiserson. "Ordering Heuristics for Parallel Graph Coloring". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2014, pp. 166–177.

[170] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. "Ordered vs. Unordered: A Comparison of Parallelism and Work-efficiency in Irregular Algorithms". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2011.

[171] Monika Henzinger, Shahbaz Khan, Richard Paul, and Christian Schulz. "Dynamic Matching Algorithms in Practice". In: *CoRR* abs/2004.09099 (2020).

[172] Monika R Henzinger and Valerie King. "Maintaining Minimum Spanning Forests in Dynamic Graphs". In: *SIAM J. on Computing* 31.2 (2001), pp. 364–374.

[173] Monika Rauch Henzinger and Valerie King. "Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation". In: *ACM Symposium on Theory of Computing (STOC)*. 1995.

[174] Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, Bozhi You, Keshav Pingali, and Vijaya Ramachandran. "A Round-Efficient Distributed Betweenness Centrality Algorithm". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2019, pp. 272–286.

[175] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity". In: *J. ACM* 48.4 (2001), pp. 723–760.

[176] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. "On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-world Graphs". In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2013, 92:1–92:11.

[177] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. "Green-Marl: a DSL for Easy and Efficient Graph Analysis". In: *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2012, pp. 349–362.

[178] John Hopcroft and Robert Tarjan. "Algorithm 447: efficient algorithms for graph manipulation". In: *Communications of the ACM* 16.6 (1973), pp. 372–378.

[179] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. "Fully Dynamic Connectivity in O(Log N(Log Log N)2) Amortized Expected Time". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2017, pp. 510–520.

[180] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. "Querying k-truss Community in Large and Dynamic Graphs". In: *ACM International Conference on Management of Data (SIGMOD)*. 2014, pp. 1311–1322.

[181] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. "LDBC Graphalytics: A Benchmark for Large-scale Graph Analysis on Parallel and Distributed Platforms". In: *Proc. VLDB Endow.* 9.13 (2016), pp. 1317–1328.

[182] Amos Israeli and Y. Shiloach. "An Improved Parallel Algorithm for Maximal Matching". In: *Information Processing Letters* 22.2 (1986), pp. 57–60.

[183] Alon Itai and Michael Rodeh. "Finding a Minimum Circuit in a Graph". In: *ACM Symposium on Theory of Computing (STOC)*. 1977, pp. 1–10.

[184] Anand Iyer, Li Erran Li, and Ion Stoica. "CellIQ : Real-Time Cellular Network Analytics at Scale". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015, pp. 309–322.

[185] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. "Time-evolving Graph Processing at Scale". In: *International Workshop on Graph Data Management Experiences and Systems (GRADES)*. 2016, 5:1–5:6.

[186] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module". In: *arXiv preprint arXiv:1903.05714* (2019).

[187] Rico Jacob and Nodari Sitchinava. "Lower Bounds in the Asymmetric External Memory Model". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2017, pp. 247–254.

[188] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[189] Jiayang Jiang, Michael Mitzenmacher, and Justin Thaler. "Parallel Peeling Algorithms". In: *ACM Trans. Parallel Comput.* 3.1 (2017).

[190] David S. Johnson. "Approximation Algorithms for Combinatorial Problems". In: *J. Comput. Syst. Sci.* 9.3 (1974).

[191] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. "GraFBoost: Using Accelerated Flash Storage for External Graph Analytics". In: *ACM International Symposium on Computer Architecture (ISCA)*. 2018, pp. 411–424.

[192] H. Kabir and K. Madduri. "Parallel k-Core Decomposition on Multicore Platforms". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 1482–1491.

[193] Bruce Kapron, Valerie King, and Ben Mountjoy. "Dynamic Graph Connectivity in Polylogarithmic Worst Case Time". In: *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms*. 2013, pp. 1131–1142.

[194] David R. Karger, Philip N. Klein, and Robert E. Tarjan. "A Randomized Linear-time Algorithm to Find Minimum Spanning Trees". In: *J. ACM* 42.2 (1995), pp. 321–328.

[195] Richard M. Karp and Vijaya Ramachandran. "Parallel Algorithms for Shared-memory Machines". In: *Handbook of Theoretical Computer Science (Vol. A)*. Ed. by Jan van Leeuwen. Cambridge, MA, USA: MIT Press, 1990, pp. 869–941.

[196] Richard M. Karp and Avi Wigderson. "A Fast Parallel Algorithm for the Maximal Independent Set Problem". In: *ACM Symposium on Theory of Computing (STOC)*. 1984, pp. 266–272.

[197] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. "Faster Worst Case Deterministic Dynamic Connectivity". In: *European Symposium on Algorithms (ESA)*. 2016, 53:1âĂŞ53:15.

[198] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. "ZipG: A Memory-efficient Graph Store for Interactive Queries". In: *ACM International Conference on Management of Data (SIGMOD)*. 2017, pp. 1149–1164.

[199] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. "K-core Decomposition of Large Networks on a Single PC". In: *Proc. VLDB Endow.* 9.1 (2015), pp. 13–23.

[200] Udayan Khurana and Amol Deshpande. "Efficient Snapshot Retrieval over Historical Graph Data". In: *International Conference on Data Engineering (ICDE)*. 2013, pp. 997–1008.

[201] Udayan Khurana and Amol Deshpande. "Storing and Analyzing Historical Graph Data at Scale". In: *International Conference on Extending Database Technology (EDBT)*. 2016, pp. 65–76.

[202] Jinha Kim, Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, and Hwanjo Yu. "OPT: A New Framework for Overlapped and Parallel Triangulation in Large-scale Graphs". In: *ACM International Conference on Management of Data (SIGMOD)*. 2014, pp. 637–648.

[203] Philip N Klein and Sairam Subramanian. "A Randomized Parallel Algorithm for Single-Source Shortest Paths". In: *J. Algorithms* 25.2 (1997), pp. 205–220.

[204] Lasse Kliemann. "Engineering a Bipartite Matching Algorithm in the Semi-Streaming Model". In: *Algorithm Engineering - Selected Results and Surveys*. Vol. 9220. Lecture Notes in Computer Science. 2016, pp. 352–378.

[205] P. Kumar and H. H. Huang. "G-Store: High-Performance Graph Store for Trillion-Edge Processing". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2016, pp. 830–841.

[206] Pradeep Kumar and H. Howie Huang. "GraphOne: A Data Store for Real-time Analytics on Evolving Graphs". In: *USENIX Conference on File and Storage Technologies (FAST)*. 2019, pp. 249–263.

[207] Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. "Fast Greedy Algorithms in MapReduce and Streaming". In: *ACM Trans. Parallel Comput.* 2.3 (2015), 14:1–14:22.

[208] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. "What is Twitter, a Social Network or a News Media?" In: *International World Wide Web Conference (WWW)*. 2010, pp. 591–600.

[209] Jakub Łącki, Vahab Mirrokni, and Michał Włodarczyk. "Connected Components at Scale via Local Contractions". In: *arXiv preprint arXiv:1807.10727* (2018).

[210] Matthieu Latapy. "Main-memory Triangle Computations for Very Large (Sparse (Power-law)) Graphs". In: *Theor. Comput. Sci.* 407.1-3 (2008), pp. 458–473.

[211] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. "WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems." In: *USENIX Conference on File and Storage Technologies (FAST)*. 2017.

[212] Charles E Leiserson and Tao B Schardl. "A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers)". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2010, pp. 303–314.

[213] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. "There's Plenty of Room at the Top: What Will Drive Computer Performance After Moore's law?" In: *Science* 368.6495 (2020).

[214] Lucas Lersch, Wolfgang Lehner, and Ismail Oukid. "Persistent Buffer Management with Optimistic Consistency". In: *International Workshop on Data Management on New Hardware*. 2019, 14:1–14:3.

[215] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. 2014.

[216] Jason Li. "Faster Parallel Algorithm for Approximate Shortest Path". In: *ACM Symposium on Theory of Computing (STOC)*. 2020, pp. 308–321.

[217] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. "Efficient Core Maintenance in Large Dynamic Graphs". In: *IEEE Transactions on Knowledge and Data Engineering* (2013), pp. 2453–2465.

[218] Jeff W Lichtman, Hanspeter Pfister, and Nir Shavit. "The Big Data Challenges of Connectomics". In: *Nature neuroscience* 17.11 (2014), pp. 1448–1454.

[219] Jihang Liu and Shimin Chen. "Initial Experience with 3D XPoint Main Memory". In: *IEEE International Conference on Data Engineering Workshops (ICDEW)*. 2019, pp. 300–305.

[220] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. "iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory". In: *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 258–270.

[221] Xinxin Liu, Yu Hua, Xuan Li, and Qifan Liu. "Write-Optimized and Consistent RDMA-based NVM Systems". In: *arXiv preprint arXiv:1906.08173* (2019).

[222] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proc. VLDB Endow.* 5.8 (2012).

[223] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. "GraphLab: A New Parallel Framework for Machine Learning". In: *Conference on Uncertainty in Artificial Intelligence (UAI)*. 2010, pp. 340–349.

[224] Michael Luby. "A Simple Parallel Algorithm for the Maximal Independent Set Problem". In: *SIAM J. Comput.* (1986), pp. 1036–1053.

[225] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. "Mosaic: Processing a Trillion-Edge Graph on a Single Machine". In: *European Conference on Computer Systems (EuroSys)*. 2017, pp. 527–543.

[226] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. "LLAMA: Efficient Graph Analytics using Large Multiversioned Arrays". In: *IEEE International Conference on Data Engineering (ICDE)*. 2015, pp. 363–374.

[227] Kamesh Madduri, David A. Bader, Jonathan W. Berry, and Joseph R. Crobak. "An Experimental Study of A Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances". In: *ALENEX*. 2007, pp. 23–35.

[228] Saeed Maleki, Donald Nguyen, Andrew Lenharth, María Garzarán, David Padua, and Keshav Pingali. "DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem". In: *Proceedings of the 2016 International Conference on Supercomputing (ICS)*. 2016, 32:1–32:14.

[229] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-scale Graph Processing". In: *ACM International Conference on Management of Data (SIGMOD)*. 2010, pp. 135–146.

[230] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[231] Yael Maon, Baruch Schieber, and Uzi Vishkin. "Parallel Ear Decomposition Search (EDS) and st-numbering in Graphs". In: *Theoretical Computer Science* 47 (1986), pp. 277–298.

[232] David W. Matula and Leland L. Beck. "Smallest-last Ordering and Clustering and Graph Coloring Algorithms". In: *J. ACM* 30.3 (1983), pp. 417–427.

[233] Robert McColl, Oded Green, and David A Bader. "A New Parallel Algorithm for Connected Components in Dynamic Graphs". In: *IEEE International Conference on High-Performance Computing (HiPC)*. 2013, pp. 246–255.

[234] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. "A Performance Evaluation of Open Source Graph Databases". In: *Workshop on Parallel Programming for Analytics Applications*. 2014, pp. 11–18.

[235] Robert Ryan McCune, Tim Weninger, and Greg Madey. "Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing". In: *ACM Comput. Surv.* 48.2 (2015), 25:1–25:39.

[236] Andrew McGregor. "Graph Stream Algorithms: A Survey". In: *SIGMOD Rec.* 43.1 (2014), pp. 9–20.

[237] William McLendon III, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. "Finding Strongly Connected Components in Distributed Graphs". In: *J. Parallel Distrib. Comput.* 65.8 (2005), pp. 901–910.

[238] Frank McSherry, Michael Isard, and Derek G. Murray. "Scalability! But at what COST?" In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2015.

[239] Kurt Mehlhorn and Ulrich Meyer. "External-Memory Breadth-First Search with Sublinear I/O". In: *European Symposium on Algorithms (ESA)*. 2002, pp. 723–735.

[240] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. "Graph Structure in the Web—Revisited: A Trick of the Heavy Tail". In: *Proceedings of the 23rd international conference on World Wide Web*. 2014, pp. 427–432.

[241] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. "The Graph Structure in the Web–Analyzed on Different Aggregation Levels". In: *The Journal of Web Science* 1.1 (2015), pp. 33–47.

[242] Ulrich Meyer and Peter Sanders. "Δ-stepping: a Parallelizable Shortest Path Algorithm". In: *J. Algorithms* 49.1 (2003), pp. 114–152.

[243] Ulrich Meyer and Peter Sanders. "Parallel Shortest Path for Arbitrary Graphs". In: *European Conference on Parallel Processing (Euro-Par)*. 2000, pp. 461–470.

[244] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. "ImmortalGraph: A system for storage and analysis of temporal graphs". In: *ACM TOS* (2015), 14:1–14:34.

[245] Othon Michail. "An introduction to temporal graphs: An algorithmic perspective". In: *Internet Mathematics* 12.4 (2016), pp. 239–280.

[246] Gary L Miller, Richard Peng, and Shen Chen Xu. "Parallel Graph Decompositions using Random Shifts". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2013, pp. 196–203.

[247] Gary L. Miller and Vijaya Ramachandran. "A New Graph Triconnectivity Algorithm and its Parallelization". In: *Combinatorica* 12.1 (1992), pp. 53–76.

[248] Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. "Improved Parallel Algorithms for Spanners and Hopsets". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2015, pp. 192–201.

[249] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. "Complexity Models for Incremental Computation". In: *Theoretical Computer Science (TCS)* 130.1 (1994).

[250] C Mohan, Bruce Lindsay, and Ron Obermarck. "Transaction Management in the R* Distributed Database Management System". In: *ACM Transactions on Database Systems (TODS)* 11.4 (1986), pp. 378–396.

[251] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. "ARIES: a Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging". In: *ACM Transactions on Database Systems (TODS)* 17.1 (1992), pp. 94–162.

[252] A. Montresor, F. De Pellegrini, and D. Miorandi. "Distributed k-Core Decomposition". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 24.2 (2013), pp. 288–300.

[253] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. "Incremental, Iterative Data Processing with Timely Dataflow". In: *Commun. ACM* 59.10 (2016), pp. 75–83.

[254] S. Muthukrishnan. "Data Streams: Algorithms and Applications". In: *Foundations and Trends in Theoretical Computer Science* 1.2 (2005), pp. 117–236.

[255] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. "GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions". In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2015, 69:1–69:12.

[256] Danupon Nanongkai and Thatchaphol Saranurak. "Dynamic Spanning Forest with Worst-Case Update Time: Adaptive, Las Vegas, and $O(n^{1/2-\varepsilon})$-time". In: *ACM Symposium on Theory of Computing (STOC)*. 2017, pp. 1122–1129.

[257] Neo4j. URL: http://neo4j.com.

[258] Mark E. J. Newman. "The Structure and Function of Complex Networks". In: *SIAM Review* 45.2 (2003), pp. 167–256.

[259] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. "A Lightweight Infrastructure for Graph Analytics". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2013, pp. 456–471.

[260] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. "A Lightweight Infrastructure for Graph Analytics". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2013.

[261] Roy Nissim and Oded Schwartz. "Revisiting the I/O-Complexity of Fast Matrix Multiplication with Recomputations". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 714–716.

[262] Sadegh Nobari, Thanh-Tung Cao, Panagiotis Karras, and Stéphane Bressan. "Scalable Parallel Minimum Spanning Forest Computation". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2012, pp. 205–214.

[263] Krzysztof Nowicki and Krzysztof Onak. "Dynamic Graph Algorithms with Batch Updates in the Massively Parallel Computation Model". In: *arXiv preprint arXiv:2002.07800* (2020).

[264] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[265] Mark Ortmann and Ulrik Brandes. "Triangle Listing Algorithms: Back from the Diversion". In: *Algorithm Engineering and Experiments (ALENEX)*. 2014, pp. 1–8.

[266] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. "Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems". In: *Proc. VLDB Endow.* 10.11 (2017), pp. 1166–1177.

[267] Anna Pagh and Rasmus Pagh. "Uniform Hashing in Constant Time and Optimal Space". In: *SIAM Journal on Computing* 38.1 (2008), pp. 85–96.

[268] Rasmus Pagh and Francesco Silvestri. "The Input/Output Complexity of Triangle Enumeration". In: *ACM Symposium on Principles of Database Systems (PODS)*. 2014, pp. 224–233.

[269] Sreepathi Pai and Keshav Pingali. "A Compiler for Throughput Optimization of Graph Algorithms on GPUs". In: *OOPSLA*. 2016, pp. 1–19.

[270] Richard C. Paige and Clyde P. Kruskal. "Parallel Algorithms for Shortest Path Problems". In: *International Conference on Parallel Processing (ICPP)*. 1985, pp. 14–20.

[271] Wen Pan, Tao Xie, and Xiaojia Song. "HART: A Concurrent Hash-Assisted Radix Tree for DRAM-PM Hybrid Memory Systems". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 921–931.

[272] Xinghao Pan, Dimitris Papailiopoulos, Samet Oymak, Benjamin Recht, Kannan Ramchandran, and Michael I Jordan. "Parallel Correlation Clustering on Big Graphs". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2015, pp. 82–90.

[273] M.M.A. Patwary, P. Refsnes, and F. Manne. "Multi-core Spanning Forest Algorithms using the Disjoint-set Data Structure". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2012, pp. 827–835.

[274] Roger Pearce, Maya Gokhale, and Nancy M Amato. "Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory". In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2010, pp. 1–11.

[275] K. Pechlivanidou, D. Katsaros, and L. Tassiulas. "MapReduce-Based Distributed k-Shell Decomposition for Online Social Networks". In: *SERVICES*. 2014.

[276] David Peleg and Alejandro A Schäffer. "Graph Spanners". In: *Journal of Graph Theory* 13.1 (1989), pp. 99–116.

[277] Seth Pettie and Vijaya Ramachandran. "A Randomized Time-Work Optimal Parallel Algorithm for Finding a Minimum Spanning Forest". In: *SIAM J. on Computing* 31.6 (2002), pp. 1879–1895.

[278] C. A. Phillips. "Parallel Graph Contraction". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1989, pp. 148–157.

[279] Chung Keung Poon and Vijaya Ramachandran. "A Randomized Linear Work EREW PRAM Algorithm to Find a Minimum Spanning Forest". In: *International Symposium on Algorithms and Computation (ISAAC)*. 1997, pp. 212–222.

[280] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. "Managing Large Graphs on Multi-cores with Graph Awareness". In: *USENIX Annual Technical Conference (ATC)*. 2012, pp. 41–52.

[281] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. "Synthesizing Parallel Graph Programs via Automated Planning". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2015, pp. 533–544.

[282] William Pugh. "Skip Lists: a Probabilistic Alternative to Balanced Trees". In: *Commun. ACM* 33.6 (1990), pp. 668–677.

[283] Sridhar Rajagopalan and Vijay V. Vazirani. "Primal-Dual RNC Approximation Algorithms for Set Cover and Covering Integer Programs". In: *SIAM J. on Computing* 28.2 (1999), pp. 525–540.

[284] Vijaya Ramachandran. "A framework for parallel graph algorithm design". In: *International Symposium on Optimal Algorithms*. 1989, pp. 33–40.

[285] Vijaya Ramachandran. "Parallel Open Ear Decomposition with Applications to Graph Biconnectivity and Triconnectivity". In: *Synthesis of Parallel Algorithms*. Ed. by John H Reif. Morgan Kaufmann Publishers Inc., 1993.

[286] David P. Reed. "Naming and Synchronization in a Decentralized Computer System". MIT EECS (PhD Thesis), 1978.

[287] J. Reif. *Optimal Parallel Algorithms for Integer Sorting and Graph Connectivity*. Tech. rep. TR-08-85. Harvard University, 1985.

[288] John H. Reif and Sandeep Sen. "Parallel Computational Geometry: An Approach using Randomization". In: *Handbook of Computational Geometry*. Ed. by J.R. Sack and J. Urrutia. Elsevier Science, 1999. Chap. 18, pp. 765–828. ISBN: 9780080529684.

[289] John H. Reif and Stephen R. Tate. "Dynamic Parallel Tree Contraction (Extended Abstract)". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1994, pp. 114–121.

[290] Omer Reingold. "Undirected Connectivity in Log-space". In: *J. ACM* 55.4 (2008), 17:1–17:24.

[291] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. "On Querying Historical Evolving Graph Sequences". In: *Proc. VLDB Endow.* 4.11 (2011), pp. 726–737.

[292] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. "Managing Non-Volatile Memory in Database Systems". In: *ACM International Conference on Management of Data (SIGMOD)*. 2018, pp. 1541–1555.

[293] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Persistent Memory I/O Primitives". In: *International Workshop on Data Management on New Hardware*. 2019, 12:1–12:7.

[294] Sascha Rothe and Hinrich Schütze. "CoSimRank: A Flexible & Efficient Graph-Theoretic Similarity Measure". In: *Annual Meeting of the Association for Computational Linguistics*. 2014, pp. 1392–1402.

[295] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Vol. 82. SIAM, 2003.

[296] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing". In: *Proc. VLDB Endow.* 11.4 (2017), pp. 420–431.

[297] Ahmet Erdem Sariyüce and Ali Pinar. "Fast Hierarchy Construction for Dense Subgraphs". In: *Proc. VLDB Endow.* 10.3 (2016), pp. 97–108.

[298] Ahmet Erdem Sariyüce, C Seshadhri, and Ali Pinar. "Local Algorithms for Hierarchical Dense Subgraph Discovery". In: *Proc. VLDB Endow.* 12.1 (2018), pp. 43–56.

[299] Ahmet Erdem Sariyüce, C Seshadhri, and Ali Pinar. "Parallel Local Algorithms for Core, Truss, and Nucleus Decompositions". In: *Proc. VLDB Endow.* 12.1 (2018), pp. 43–56.

[300] Ahmet Erdem Sariyüce, C Seshadhri, Ali Pinar, and Umit V Catalyurek. "Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions". In: *International World Wide Web Conference (WWW)*. 2015, pp. 927–937.

[301] T. Schank. "Algorithmic Aspects of Triangle-Based Network Analysis". PhD thesis. Universitat Karlsruhe, 2007.

[302] Thomas Schank and Dorothea Wagner. "Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study". In: *Workshop on Experimental and Efficient Algorithms (WEA)*. Santorini Island, Greece, 2005, pp. 606–609.

[303] Warren Schudy. "Finding Strongly Connected Components in Parallel Using $O(\lg^2 N)$ Reachability Queries". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2008, pp. 146–151.

[304] Raimund Seidel and Cecilia R Aragon. "Randomized Search Trees". In: *Algorithmica* 16.4-5 (1996), pp. 464–497.

[305] Stephen B. Seidman. "Network Structure and Minimum Degree". In: *Soc. Networks* 5.3 (1983), pp. 269–287.

[306] Dipanjan Sengupta and Shuaiwen Leon Song. "EvoGraph: On-the-Fly Efficient Mining of Evolving Graphs on GPU". In: *International Supercomputing Conference (ISC)*. 2017, pp. 97–119.

[307] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. "GraphIn: An Online High Performance Incremental Graph Processing Framework". In: *European Conference on Parallel Processing (Euro-Par)*. 2016, pp. 319–333.

[308] Martin Sevenich, Sungpack Hong, Adam Welc, and Hassan Chafi. "Fast In-Memory Triangle Listing for Large Real-World Graphs". In: *Workshop on Social Network Mining and Analysis*. 2014, 2:1–2:9.

[309] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. "Accelerating Dynamic Graph Analytics on GPUs". In: *Proc. VLDB Endow.* 11.1 (2017), pp. 107–120.

[310] Bin Shao, Haixun Wang, and Yatao Li. "Trinity: A Distributed Graph Engine on a Memory Cloud". In: *ACM International Conference on Management of Data (SIGMOD)*. 2013, pp. 505–516.

[311] Yishu Shen and Zhaonian Zou. "Efficient Subgraph Matching on Non-Volatile Memory". In: *International Conference on Web Information Systems Engineering*. 2017, pp. 457–471.

[312] Hanmao Shi and Thomas H Spencer. "Time-Work Tradeoffs of the Single-Source Shortest Paths Problem". In: *J. Algorithms* 30.1 (1999).

[313] Jessica Shi, Laxman Dhulipala, and Julian Shun. "Parallel Clique Counting and Peeling Algorithms". In: *arXiv preprint arXiv:2002.10047* (2020).

[314] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. "Tornado: A System For Real-Time Iterative Analysis Over Evolving Data". In: *ACM International Conference on Management of Data (SIGMOD)*. 2016, pp. 417–430.

[315] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. "Graph Processing on GPUs: A Survey". In: *ACM Comput. Surv.* 50.6 (2018), 81:1–81:35.

[316] Yossi Shiloach and Uzi Vishkin. "An $O(\lg n)$ Parallel Connectivity Algorithm". In: *J. Algorithms* 3.1 (1982), pp. 57–67.

[317] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. "CoreScope: Graph Mining Using k-Core Analysis–Patterns, Anomalies and Algorithms". In: *ICDM*. 2016.

[318] Thomas Shull, Jian Huang, and Josep Torrellas. "AutoPersist: an Easy-to-use Java NVM Framework based on Reachability". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2019, pp. 316–332.

[319] Julian Shun and Guy E. Blelloch. "Ligra: A Lightweight Graph Processing Framework for Shared Memory". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2013, pp. 135–146.

[320] Julian Shun and Guy E Blelloch. "Phase-Concurrent Hash Tables for Determinism". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2014, pp. 96–107.

[321] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. "A Simple and Practical Linear-work Parallel Algorithm for Connectivity". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2014, pp. 143–153.

[322] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. "Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+". In: *Data Compression Conference (DCC)*. 2015, pp. 403–412.

[323] Julian Shun and Kanat Tangwongsan. "Multicore Triangle Computations without Tuning". In: *IEEE International Conference on Data Engineering (ICDE)*. 2015, pp. 149–160.

[324] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. "Brief Announcement: The Problem Based Benchmark Suite". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2012.

[325] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W. Mahoney. "Parallel Local Graph Clustering". In: *Proc. VLDB Endow.* 9.12 (2016), pp. 1041–1052.

[326] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. "Reducing Contention Through Priority Updates". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2013, pp. 299–300.

[327] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. "Work-Efficient Parallel Union-Find with Applications to Incremental Graph Connectivity". In: *European Conference on Parallel Processing (Euro-Par)*. 2016.

[328] Daniel Dominic Sleator and Robert Endre Tarjan. "A Data Structure for Dynamic Trees". In: *Journal of Computer and System Sciences* 26.3 (1983), pp. 362–391.

[329] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-adjusting binary search trees". In: *Journal of the ACM* 32.3 (1985), pp. 652–686.

[330] G. M. Slota, S. Rajamanickam, and K. Madduri. "A Case Study of Complex Graph Analysis in Distributed Memory: Implementation and Optimization". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 293–302.

[331] George M Slota and Kamesh Madduri. "Simple Parallel Biconnectivity Algorithms for Multicore Platforms". In: *IEEE International Conference on High-Performance Computing (HiPC)*. 2014, pp. 1–10.

[332] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. "BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2014, pp. 550–559.

[333] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. *Supercomputing for Web Graph Analytics*. Tech. rep. SAND2015-3087C. Sandia National Lab, 2015.

[334] Thomas H. Spencer. "Time-work Tradeoffs for Parallel Algorithms". In: *J. ACM* 44.5 (1997).

[335] Daniel A. Spielman and Shang-Hua Teng. "Nearly-Linear Time Algorithms for Graph Partitioning, Graph Sparsification, and Solving Linear Systems". In: *ACM Symposium on Theory of Computing (STOC)*. 2004, pp. 81–90.

[336] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. "Shortcutting Label Propagation for Distributed Connected Components". In: *International Conference on Web Search and Data Mining (WSDM)*. 2018, pp. 540–546.

[337] Stergios Stergiou and Kostas Tsioutsiouliklis. “Set Cover at Web Scale”. In: *SIGKDD*. 2015.

[338] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. “GraphGrind: Addressing Load Imbalance of Graph Partitioning”. In: *International Conference on Supercomputing (ICS)*. 2017, 16:1–16:10.

[339] Peng Sun, Yonggang Wen, Ta Nguyen Binh Duong, and Xiaokui Xiao. “GraphMP: An Efficient Semi-External-Memory Big Graph Processing System on a Single Machine”. In: *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 2017, pp. 276–283.

[340] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. “PAM: Parallel Augmented Maps”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2018, pp. 290–304.

[341] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. “PAM: Parallel Augmented Maps”. In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2018.

[342] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. “On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-Versioned Indexes”. In: *Proc. VLDB Endow.* 13.2 (2019), pp. 211–225.

[343] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. “Efficient Subgraph Matching on Billion Node Graphs”. In: *Proc. VLDB Endow.* 5.9 (2012), pp. 788–799.

[344] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. “Towards Large-scale Graph Stream Processing Platform”. In: *International World Wide Web Conference (WWW)*. 2014, pp. 1321–1326.

[345] Robert E Tarjan. “Dynamic Trees as Search Trees via Euler Tours, Applied to the Network Simplex Algorithm”. In: *Mathematical Programming* 78.2 (1997), pp. 169–177.

[346] Robert E Tarjan and Uzi Vishkin. “An Efficient Parallel Biconnectivity Algorithm”. In: *SIAM J. on Computing* 14.4 (1985), pp. 862–874.

[347] Robert Endre Tarjan. “Efficiency of a Good but not Linear Set Union Algorithm”. In: *J. ACM* 22.2 (1975), pp. 215–225.

[348] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. “Automatic Algorithm Transformation for Efficient Multi-Snapshot Analytics on Temporal Graphs”. In: *Proc. VLDB Endow.* 10.8 (2017), pp. 877–888.

[349] Mikkel Thorup. “Decremental Dynamic Connectivity”. In: *J. Algorithms* 33.2 (1999), pp. 229–243.

[350] Mikkel Thorup. "Near-Optimal Fully-Dynamic Graph Connectivity". In: *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*. 2000, pp. 343–350.

[351] Mikkel Thorup and Uri Zwick. "Approximate Distance Oracles". In: *J. ACM* 52.1 (2005), pp. 1–24.

[352] Vincent A Traag, Ludo Waltman, and Nees Jan van Eck. "From Louvain to Leiden: Guaranteeing Well-Connected Communities". In: *Scientific reports* 9.1 (2019), pp. 1–12.

[353] Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. "Batch-Parallel Euler Tour Trees". In: *Algorithm Engineering and Experiments (ALENEX)* (2019), pp. 92–106.

[354] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. "Batch-Parallel Euler Tour Trees". In: *CoRR* abs/1810.10738 (2018).

[355] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. "Batch-Parallel Euler Tour Trees". In: *ALENEX*. 2019.

[356] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. "The Anatomy of the Facebook Social Graph". In: *CoRR* abs/1111.4503 (2011).

[357] Stratis D. Viglas. "Adapting the B^+-Tree for Asymmetric I/O". In: *Advances in Databases and Information Systems (ADBIS)*. 2012.

[358] Stratis D. Viglas. "Write-Limited Sorts and Joins for Persistent Memory". In: *Proc. VLDB Endow.* 7.5 (2014), pp. 413–424.

[359] Keval Vora, Rajiv Gupta, and Guoqing Xu. "KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2017, pp. 237–251.

[360] Keval Vora, Rajiv Gupta, and Guoqing Xu. "Synergistic Analysis of Evolving Graphs". In: *ACM Trans. Archit. Code Optim.* 13.4 (2016), 32:1–32:27.

[361] Chundong Wang, Sudipta Chattopadhyay, and Gunavaran Brihadiswarn. "Crash Recoverable ARMv8-Oriented B+-Tree for Byte-Addressable Persistent Memory". In: *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 2019, pp. 33–44.

[362] Jia Wang and James Cheng. "Truss Decomposition in Massive Networks". In: *Proc. VLDB Endow.* 5.9 (2012), pp. 812–823.

[363] Kai Wang, Guoqing (Harry) Xu, Zhendong Su, and Yu David Liu. "GraphQ: Graph Query Processing with Abstraction Refinement - Scalable and Programmable Analytics over Very Large Graphs on a Single PC". In: *USENIX Annual Technical Conference (ATC)*. 2015, pp. 387–401.

[364] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. "RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 763–782.

[365] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. "Gunrock: a High-Performance Graph Processing Library on the GPU". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2016.

[366] Yiqiu Wang, Yan Gu, and Julian Shun. "Theoretically-Efficient and Practical Parallel DBSCAN". In: *ACM International Conference on Management of Data (SIGMOD)*. 2020, pp. 2555–2571.

[367] Dominic JA Welsh and Martin B Powell. "An Upper Bound for the Chromatic Number of a Graph and its Application to Timetabling Problems". In: *The Computer Journal* 10.1 (1967), pp. 85–86.

[368] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. "Efficient Structural Graph Clustering: an Index-Based Approach". In: *Proc. VLDB Endow.* 11.3 (2017), pp. 243–255.

[369] J. G. White, E. Southgate, J. N. Thomson, and S. Brenner. "The Structure of the Nervous System of the Nematode Caenorhabditis elegans". In: *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences* 314.1165 (1986), pp. 1–340.

[370] C. Wickramaarachchi, A. Kumbhare, M. Frincu, C. Chelmis, and V. K. Prasanna. "Real-Time Analytics for Fast Evolving Social Graphs". In: *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2015.

[371] Martin Winter, Rhaleb Zayer, and Markus Steinberger. "Autonomous, Independent Management of Dynamic Graphs on GPUs". In: *IEEE Conference on High Performance Extreme Computing (HPEC)*. 2017, pp. 1–7.

[372] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes (2nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., 1999.

[373] Stefan Wuchty and Eivind Almaas. "Peeling the Yeast Protein Network". In: *Proteomics* 5.2 (2005), pp. 444–449.

[374] Christian Wulff-Nilsen. "Faster Deterministic Fully-Dynamic Graph Connectivity". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2013, pp. 1757–1769.

[375] Christian Wulff-Nilsen. "Fully-Dynamic Minimum Spanning Forest with Improved Worst-Case Update Time". In: *ACM Symposium on Theory of Computing (STOC)*. 2017, pp. 1130–1143.

[376] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas AJ Schweiger. "Scan: A Structural Clustering Algorithm for Networks". In: *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 2007, pp. 824–833.

[377] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. "Big Graph Analytics Platforms". In: *Foundations and Trends in Databases* 7.1-2 (2017), pp. 1–195.

[378] Jaewon Yang and Jure Leskovec. "Defining and Evaluating Network Communities based on Ground-Truth". In: *Knowledge and Information Systems* 42.1 (2015), pp. 181–213.

[379] Chunxing Yin, Jason Riedy, and David A. Bader. "A New Algorithmic Model for Graph Analysis of Streaming Data". In: *International Workshop on Mining and Learning with Graphs*. 2018.

[380] Kaiyuan Zhang, Rong Chen, and Haibo Chen. "NUMA-Aware Graph-Structured Analytics". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2015, pp. 183–193.

[381] Lu Zhang and Steven Swanson. "Pangolin: A Fault-Tolerant Persistent Memory Programming Library". In: *USENIX Annual Technical Conference (ATC)*. 2019, pp. 897–912.

[382] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. "FastSV: A distributed-memory connected component algorithm with fast convergence". In: *SIAM Conference on Parallel Processing for Scientific Computing (PP)*. SIAM. 2020, pp. 46–57.

[383] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. "GraphIt: A High-performance Graph DSL". In: *Object-Oriented Programming Systems, Languages,and Applications (OOPSLA)* 2 (2018), 121:1–121:30.

[384] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. "Optimizing Ordered Graph Algorithms with GraphIt". In: *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. 2020, pp. 158–170.

[385] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. "FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs". In: *USENIX Conference on File and Storage Technologies (FAST)*. 2015, pp. 45–58.

[386] Da Zheng, Disa Mhembere, Vince Lyzinski, Joshua T. Vogelstein, Carey E. Priebe, and Randal Burns. "Semi-External Memory Sparse Matrix Multiplication for Billion-Node Graphs". In: *IEEE Trans. Parallel Distrib. Syst.* 28.5 (2017), pp. 1470–1483.

[387] Tingzhe Zhou, Pantea Zardoshti, and Michael Spear. "Brief Announcement: Optimizing Persistent Transactions". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2019, pp. 169–170.

[388] Wei Zhou. "A Practical Scalable Shared-Memory Parallel Algorithm for Computing Minimum Spanning Trees". MA thesis. KIT, 2017.

[389] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. "Gemini: A Computation-Centric Distributed Graph Processing System". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 301–316.