Provably Efficient and Scalable Shared-Memory Graph Algorithms

for Static, Dynamic, and Streaming Graphs

(Thesis Proposal)

Laxman Dhulipala

Compute Science Department
Carnegie Mellon University
ldhulipa@cs.cmu.edu

Thesis Committee

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

Abstract

Parallel graph algorithms are important to a variety of computational disciplines today due to the widespread availability of large-scale graph-based data. Existing work that processes very large graphs restricts itself to static graphs and uses distributed memory, which often requires large computational and hardware cost, while still being prohibitively slow. This thesis will argue that shared-memory algorithms and techniques can solve a wide range of fundamental problems on very large static, dynamic, and streaming graphs quickly, provably-efficiently, and scalably, using a modest amount of computational resources.

The first part of this thesis will focus on efficient parallel graph processing. We will propose the Graph Based Benchmark Suite, which contains provably-efficient implementations of over 20 fundamental graph problems. Our implementations 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. We will then introduce techniques to efficiently process graphs that are stored on non-volatile memory (NVRAMs). Compared to existing results in the literature for the WebDataCommons graph, both our shared-memory and non-volatile memory implementations use orders of magnitude fewer resources, and in many cases also run an order of magnitude faster than existing distributed memory codes.

The second part of this thesis will focus on algorithms and systems for dynamic and streaming graphs. We will work in the Parallel Batch-Dynamic model, which allows algorithms to ingest batches of updates and exploit parallelism. We will describe algorithms for forest-connectivity connectivity, and for maintaining low-outdegree orientations in this model. Next, we will turn to practical data structures and algorithms for representing graphs that change over time. We design a dynamic graph representation based on a new compressed purely-functional tree data structure, called a C-tree, which admits provably-efficient parallel batch updates. Compared to existing work, our dynamic graph representation scales to much higher update rates, while using significantly less memory. Based on these ideas, we propose a new graph-streaming system called Aspen, and show that using Aspen, we can concurrently update and analyze the WebDataCommons hyperlink graph on a single commodity multicore machine with a terabyte of main memory.

1 Introduction

In recent years, due to a wealth of applications to prediction, clustering, and information retrieval tasks, graph processing has become an important tool in many computational fields, such as biology and economics, as well as at companies such as Google, Facebook, and LinkedIn. Graphs that are available today contain *billions of vertices*, and *hundreds of billions of edges*, and often *change* over time, for example as a new transaction occurs in a financial network, or due to a road closure in a transportation network.

Due to the importance of analyzing and understanding such graphs, considerable effort over the past decade has been devoted to processing very large graphs, much of it on distributed and external memory systems. However, there is almost no existing work that processes very large graphs using shared-memory. There are two likely reasons. First, representing large graphs in standard formats can require a prohibitive amount of memory. Second, it is unclear how to achieve high performance for a number of fundamental problems, such as connectivity, minimum spanning forest, and strong connectivity, amongst many others. Although work-efficient algorithms for these problems exist, it is unknown whether they are practical, or if they scale to large graphs.

Representing and processing graphs that change over time raises additional challenges. First, it is unclear how to store such graphs in a memory-efficient format that supports fast updates. Additionally, it is important that the updates can be batched and internally parallelized, which enables scaling to very high update rates. Finally, it is unclear how to dynamically maintain properties of such graphs while exploiting batching and parallelism.

Therefore, work in the literature by and large restricts itself to static graphs, uses distributed or external memory, and focuses on a stable of simple problems. Unfortunately, existing distributed-memory solutions are extremely resource inefficient, and existing disk-based solutions are often unacceptably slow. A pressing question then, is how to design parallel algorithms for both static and dynamic graphs that achieve good performance for a diverse set of problems on a single machine, using a modest amount of RAM.

We will study this question by studying the following three topics, split across two main parts. The first part (1) will focus on provably-efficient parallel graph algorithms, and the second part (2 and 3) will focus on provably-efficient parallel dynamic and streaming algorithms/systems.

1. **Provably Efficient Parallel Graph Algorithms** We will design efficient frameworks and algorithmic techniques for developing shared-memory implementations for more than 20 fundamental graph problems. We will also study these algorithms in a theoretical model capturing read-write asymmetry, and show their efficiency on NVRAM-equipped systems, thereby showing how to scale to graphs whose edges cannot be stored in memory.

2. **Batch-Dynamic Algorithms** We study fundamental basic problems in the Parallel Batch-Dynamic model. We will design algorithms for problems such as forest connectivity, general connectivity, and low-outdegree orientation, and experimentally evaluate these algorithms.

3. **Graph Streaming** We will describe a practical and provably-efficient graph-streaming system, *Aspen*, based on a new compressed purely-functional tree data structure called a *C*-tree. We will extend Aspen to handle very large attributed graphs as well as graphs whose edges cannot be stored in memory by using NVRAM.

Thesis statement: Shared-memory parallel graph algorithms can solve a diverse set of fundamental problems on large static, dynamic, and streaming graphs quickly, provably-efficiently, and scalably, all at low cost due to using only a modest amount of computational resources.

Outline.

- Section 2 motivates our work single-machine shared-memory graph analytics, summarizes the work in this thesis that is already completed in this area, and proposes two new projects that fit under the scope of this topic, on a large-scale benchmark for connectivity algorithms, and fast, memory-efficient algorithms for the k-Truss problem.
- Section 3 introduces the Batch-Dynamic and Parallel Batch-Dynamic models, and provides motivation for proposing these new models. It then summarizes the finished work on this topic, and proposes a new project on efficiently computing low-outdegree orientations in the parallel batch-dynamic setting.
- Section 4 describes new algorithms and systems in the graph-streaming setting. It discusses motivation for this emerging area of graph-processing systems, and presents our recent work on efficient parallel data structures for efficiently updating a dynamic graph, and introduces Aspen, a new graph-streaming system with good empirical performance and strong provable bounds on its performance. We will end this section by discussing a proposed new project that extends Aspen to graphs that are larger-than-memory.

Finally, we will present a tentative schedule for completing the proposed work and writing the Thesis in Section 5, and conclude with some thoughts about the thesis statement in Section 6. We provide some shared notation, and detailed description of the parallel models considered in this work in Appendix A.

Acknowledgements. The results discussed in this thesis proposal are the result of close collaboration and interaction with many wonderful people. I am truly grateful to Guy Blelloch, who is an incomparable advisor, researcher and role model, to Julian Shun, who has inspired and helped me throughout every stage of this journey, to my talented and dedicated collaborators, and to my friends who generously discussed many ideas—both research-related and not—with me. I am especially grateful for my family, who have unfailingly supported me in life.

2 Provably-Efficient Parallel Graph Algorithms

Overview. The first topic of this thesis is on designing and implementing fast shared-memory graph algorithms that scale to very large graphs on a single machine. This topic will be split into three sub-parts, united by the theme of desinging provably-efficient parallel graph algorithms for shared-memory systems. By ***provably-efficient***, we mean that the algorithms have provable bounds on their work and depth. The gold-standard is to obtain a ***work-efficient*** parallel graph algorithm (i.e., one which performs asymptotically the same amount of work as the fastest sequential algorithm) with poly-logarithmic depth, and most of the algorithms studied in this thesis are in fact work-efficient NC algorithms.

The first part will study a family of graph problems such as k-core, weighted breadth-first search, and approximate set cover that can all be expressed using *bucketing*. These algorithms all maintain a dynamic mapping between vertices and a set of buckets, which is updated over the algorithm's execution. We will design a high-level framework called *Julienne* that admits simple codes that achieve high performance compared to existing, fairly complicated codes for these problems.

The second part will study a collection of 20 fundamental graph problems, ranging from connectivity problems such as undirected connectivity, minimum spanning forest, biconnectivity, and strong connectivity, to covering, or symmetry breaking problems like maximal independent set,

maximal matching, and graph coloring, amongst many others. We will implement efficient parallel graph algorithms for all of these problems using only simple fork-join primitives, similar to those used in Cilk. Our contributions from this part of the thesis will be made publicly available as the Graph Based Benchmark Suite (GBBS), a problem-based benchmark suite for graph problems.[1]

Shortest Path Problems	Breadth-First Search, Integral-Weight SSSP, General-Weight SSSP, Positive-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, k-Truss, Approximate Densest Subgraph, Triangle Counting
Eigenvector Problems	PageRank

Table 1: 22 important graph problems considered in the Graph Based Benchmark Suite (GBBS), covering a broad range of techniques and application areas.

A full list of the problems studied in this part of the thesis can be found in Table 1. An important aspect of this work is the fact that the algorithms implemented *span a broad range of techniques and application areas*. For example, we consider not only shortest-path problems, but also a wide family of connectivity problems. We also consider certain P-complete problems which may seem "hopeless" to parallelize, but in practice offer ample parallelism on real-world instances (and obtain good speedups, relative to fast sequential implementations). The diversity of the problems studied in this thesis should be viewed in contrast to the current state-of-affairs in distributed and external graph algorithms, where work frequently only considers a handful of problems which are usually expressible as iterated matrix-vector products. We note that in many of these cases, our implementations are the first publicly-available parallel implementations of algorithms for these problems.

The third part will study how to apply our ideas and techniques to graphs that do not fit within the DRAM of a single machine. Instead of assuming the that graph is stored on disk, we consider storing graphs on an emerging class of NVRAM technologies, which are byte-addressible, and are likely to be cheaper, as well as offer higher memory-density and capacity than DRAMs. A challenge of these new technologies is to overcome an *asymmetry* between reads and writes—write operations to NVRAMs are more expensive than reads in terms of energy and throughput. We observe that most graphs found in practice are sparse, but still tend to have many more edges than vertices, often from one to two orders of magnitude more. Due to this fact, we assume that the system contains enough DRAM to store the vertices, but not all of the edges, which are stored on NVRAM. To study what problems can be studied efficiently under this restriction, we define a 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. We study many of the problems considered in the second part of this thesis in the PSAM model and show that they are work-efficient in the new model. Finally, we perform an experimental evaluation of our algorithms on Intel's Optane DC Persistent Memory and show that our algorithms achieve performance comparable to state-of-the-art DRAM implementations, and are significantly faster than existing graph-processing systems that support NVRAMs.

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

2.1 Related Work

Efficient 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., [159, 97, 114, 7, 175, 127, 144, 93, 53, 141, 126, 71, 25, 122]). 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 [96]. 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 [96], 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 [152, 34]. Other problems, like k-core are P-complete [10], which rules out polylogarithmic-depth algorithms for them unless $\mathsf{P} = \mathsf{NC}$ [78]. However, even k-core admits an algorithm with strong theoretical guarantees that is efficient in practice [61].

Graph Processing Systems. Motivated by the need to process very large graphs, there have been many graph processing frameworks developed in the literature (e.g., [58, 143, 136, 185, 117, 76, 113, 131, 162] among many others). We refer the reader to [119, 188] for surveys of existing frameworks.

Processing Very Large Graphs. Several recent graph processing systems evaluate the scalability of their implementations by solving problems on massive graphs [167, 56, 115, 61, 95, 170]. All of these systems report running times either on the Hyperlink2012 graph or Hyperlink2014 graphs, two web crawls released by the WebDataCommons that are the largest and second largest publicly-available graphs respectively.

2.2 Contributions and Techniques

(1) We will describe a graph-processing framework called Julienne, which extends the Ligra graph processing framework with primitives for bucketing. Using Julienne, we obtain simple and practical work-efficient parallel algorithms for weighted breadth-first search, k-core, approximate set cover, and a simple, high-level implementation of Δ-stepping.

(2) We will discuss implementations of provably-efficient parallel algorithms for 22 important graph problems, included as part of the Graph Based Benchmark Suite. We will provide results showing that our implementations outperform existing state-of-the-art implementations on the largest real-world graphs.

Graph Dataset	Num. Vertices	Num. Edges	Uncompressed Size	Compressed Size	Savings
ClueWeb [39]	978,408,098	74,744,358,623	285GB	100GB	2.8x
Hyperlink2014 [121]	1,724,573,718	124,141,874,032	474GB	186GB	2.5x
Hyperlink2012 [121]	3,563,602,789	225,840,663,232	867GB	354GB	2.4x

Table 2: Graph inputs, including number of vertices, edges, memory required to store the graph in an uncompressed CSR format, memory required to store the graph in a parallel byte-compressed CSR format, and the savings obtained over the uncompressed format by the compressed format.

(3) We will describe a new parallel computational model, called the *Parallel Semi-Asymmetric Model*, capturing an inherent asymmetry between reads and writes on emerging hardware technologies like non-volatile memory (NVRAM). We will show how many of the algorithms in GBBS achieve excellent bounds in this model, and provide a corresponding empirical study on the recently announced Intel Optane DC NVRAM modules, showing that implementations of PSAM graph algorithms achieve state-of-the-art results on NVRAM-equipped machines.

We now describe some of the shared themes and techniques that are developed in this work.

Framework-Based Design. We rely heavily on *framework-based design* in the work that will be presented in this thesis. This methodology can be summarized as follows in the context of graph algorithms: design a set of simple, high-level primitives on graphs, vertices, and related objects, and use these to describe graph algorithms in a way that is agnostic of the underlying (memory) representation of the object. We provide implementations of a variety of graph-processing primitives that are fast in practice, and importantly are equipped with costs in terms of their *work* and *depth*. The work done in this thesis builds on and extends prior work done by Julian Shun and Guy Blelloch on the *Ligra* graph processing framework [162], which introduce a simple, high-level graph processing framework for graph traversal algorithms.

The advantage of a framework with good costs on its primitives is that we can combine an analysis of the overall cost of an algorithm with a provably-efficient scheduler, like those developed in the seminal family of work-stealing schedulers [37, 12] to obtain a guarantee that an algorithm implementation is asymptotically efficient (i.e., it runs in asymptotically the same amount of work and depth as obtained in the analysis of the algorithm). We discuss details of instantiating an algorithm with a provably-efficient scheduler in more detail in Appendix A. Although this general approach is by no means new, it does not seem to be widely used in the design and implementation of parallel or distributed graph algorithms in practice. This is likely due to a combination of practitioners implementing "simple" algorithms that have no, or poor bounds on their theoretical performance, or by using parallel runtimes and schdulers that offer no theoretical bounds on algorithmic performance.

Compression. Another benefit of the framework approach of designing simple, high-level primitives is that we can provide different implementations of the primitive which offer trade-offs between different cost measures. An example illustrating this idea can be seen in Ligra+ [165], which extends Ligra to support compressed graphs, trading a modest amount of extra cycles required to decode compressed data in exchange for significant memory savings. In Ligra+, we designed primitives for efficiently manipulating and analyzing graphs stored in a compressed, difference-encoded format. By using high-level primitives in the framework, we were able to support compressed graphs without any changes to user-facing graph algorithms, with all changes confined to a new implementation of vertex primitives for compressed graphs. Part of the work that will be described in this thesis extends the Ligra+ primitives to support a wider range of functionality, such as filtering, packing, and performing intersections of neighbor lists, all with good theoretical bounds on their work and depth.

Provably-Efficient Algorithms. All of our implementations in this work have strong provable bounds on their work and depth, as shown in Table 3. There are several reasons that algorithms with such guarantees are desirable. For one, they are robust as even adversarially-chosen 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 reduction in work often translates to faster running times on the same number of cores [61]. We note that most running times that have been reported in the literature on the Hyperlink Web graph use parallel algorithms that are not provably-efficient.

Avoiding Contention. 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 fetch_and_adds 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 5). For these graphs, we observed that many early rounds, which process vertices with low coreness perform a large number of fetch_and_adds on memory locations corresponding to high-degree vertices, resulting in high contention [163]. To reduce contention, we designed a work-efficient histogram implementation that can perform this step while only incurring $O(\log n)$ contention w.h.p. 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.

Memory-Efficiency. Another important theme in this work is memory-efficiency, which we have already partly discussed above, in the context of compression. Compression optimizes the memory usage of statically representing a graph. Memory-efficiency is also important during an algorithm's execution. One of the core primitives used by our algorithms is EDGEMAP (described in Appendix A). 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.

2.3 Parallel Graph Algorithms in Shared-Memory

In this part of the thesis, we will 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, triangle counting and several other problems. We will 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, the largest and second largest publicly-available graphs, as well as several smaller real-world graphs at various scales.

Ordered Algorithms from Julienne. Several of the algorithms in our study require maintaining a dynamic mapping between vertices, and a set of ordered buckets over the course of their execution. These problems include weighted breadth-first search (computing single-source shortest paths on an integer-weighted graph), Δ-stepping, parallel approximate set-cover, and k-core, amongst several others. To provide simple and theoretically-efficient implementations of these algorithms, we designed and implement a work-efficient interface for bucketing in the Ligra shared-memory graph processing framework [162]. 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).

Problem	(1)	(72h)	(SU)	**Alg.**	**Model**	**Work**	**Depth**
Breadth-First Search (BFS)	576	8.44	68	–	TS	$O(m)$	$O(\mathsf{diam}(G)\log n)$
Integral-Weight SSSP (weighted BFS)	3770	58.1	64	[61]	PW	$O(m)$ expected	$O(\mathsf{diam}(G)\log n)$ w.h.p.[†]
General-Weight SSSP (Bellman-Ford)	4010	59.4	67	[54]	PW	$O(\mathsf{diam}(G)m)$	$O(\mathsf{diam}(G)\log n)$
SS Widest Path (Bellman-Ford)	3210	48.4	66	[54]	PW	$O(\mathsf{diam}(G)m)$	$O(\mathsf{diam}(G)\log n)$
SS Betweenness Centrality (BC)	2260	37.1	60	[40]	FA	$O(m)$	$O(\mathsf{diam}(G)\log n)$
$O(k)$-Spanner	2390	36.5	65	[125]	TS	$O(m)$	$O(k\log n)$ w.h.p.
Low-Diameter Decomposition (LDD)	980	16.6	59	[126]	TS	$O(m)$	$O(\log^2 n)$ w.h.p.
Connectivity	1640	25.0	65	[164]	TS	$O(m)$ expected	$O(\log^3 n)$ w.h.p.
Spanning Forest	2420	35.8	67	[164]	TS	$O(m)$ expected	$O(\log^3 n)$ w.h.p.
Biconnectivity	9860	165	59	[175]	FA	$O(m)$ expected	$O(\max(\mathsf{diam}(G)\log n, \log^3 n))$ w.h.p.
Strong Connectivity (SCC)*	8130	185	43	[34]	PW	$O(m\log n)$ expected	$O(\mathsf{diam}(G)\log n)$ w.h.p.
Minimum Spanning Forest (MSF)	9520	187	50	[195]	PW	$O(m\log n)$	$O(\log^2 n)$
Maximal Independent Set (MIS)	2190	32.2	68	[32]	FA	$O(m)$ expected	$O(\log^2 n)$ w.h.p.
Maximal Matching (MM)	7150	108	66	[32]	PW	$O(m)$ expected	$O(\log^3 m/\log\log m)$ w.h.p.
Graph Coloring	8920	158	56	[83]	FA	$O(m)$	$O(\log n + L\log\Delta)$
Approximate Set Cover	5320	90.4	58	[36]	PW	$O(m)$ expected	$O(\log^3 n)$ w.h.p.
k-core	8515	184	46	[61]	FA	$O(m)$ expected	$O(\rho\log n)$ w.h.p.
Approximate Densest Subgraph	3780	51.4	73	[15]	FA	$O(m)$	$O(\log^2 n)$
Triangle Counting (TC)	—	1168	—	[166]	–	$O(m^{3/2})$	$O(\log n)$
PageRank Iteration	973	13.1	74	[41]	FA	$O(n+m)$	$O(\log n)$

Table 3: 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 TRAM used (**MM**) 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 (w.h.p.)*** if it has $O(k\cdot f(n))$ cost with probability at least $1-1/n^k$.

Performance. Table 3 reports running times for the problems considered in our work on the WebDataCommons Hyperlink2012 graph on a single 72-core machine with 2-way hyper-threading enabled, and 1TB of RAM. The table also describes the work and depth bounds of the algorithms. Details about the machine model we use can be found in Appendix A, but informally one can think

of the model as being roughly equivalent to a PRAM, augmented with different unit-cost atomic instructions, such as a test-and-set, or fetch-and-add.

Our first observation is that in our algorithms achieve good speedups, ranging between 43–74x across all problems. We note that we could not obtain a single-thread running time for triangle counting due to the sheer amount of time (based on speedups of around 70x for triangle counting on smaller graphs, the running time should be about a whole day of computation). Other than triangle counting, all of our running times take between seconds, to a few minutes, with the longest parallel time for computing strong connectivity, minimum spanning forests, and for computing all k-cores of the graph. Even in these cases, the running times are barely over three minutes. We believe that running times in this range are within a realistic range that enable data analysts, or other experimental researchers to iteratively perform analyses on these graphs without long running times (for example, one could start a job and get a coffee or use the bathroom, with the job finishing by the time one is back—having running times in this range enable the analyst to "be in the loop" without suffering from excruciating boredom). We will discuss how our performance compares with other existing numbers reported in the literature for this graph in detail in Section 2.5.

2.4 Parallel Graph Algorithms on NVRAMs

Why Non-Volatile Memory? As we have shown at this point in the thesis, single-machine, shared-memory graph analytics by-and-large outperform their distributed memory counterparts, running up to orders of magnitude faster using much fewer resources [162, 120, 165, 62]. From a cost-efficiency perspective, it is hard to imagine a use-case where using distributed memory leads to significant improvements in running times without an order of magnitude increase in hardware, and therefore cost. The rise of single-machine analytics is mostly due to a steady increase in main-memory sizes, and main-memory bandwidth over the past two decades—today one can readily purchase a single multi-socket machine equipped with anywhere from 8–16TB of DRAM, and build a machine configuration that can achieve 100–150GB/s of bandwidth. However, building such a machine can still be expensive, costing tens of thousands of dollars, with DRAM being responsible for the overwhelming majority of this cost.

Over the past decade, Non-Volatile Random Access Memories (NVRAMs) have been in development, and have started to become available in the market more recently as DIMMs (memory modules).[2] These NVRAM DIMMs, 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). In addition, due to their non-volatility, these memories preserve their state under power failures, which is an attractive feature for fault-tolerant computing.

Challenges. 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, there are several important challenges that are specific to using NVRAM. The most significant one 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 [30, 19, 47, 181]. 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.

As an example of the technology and its tradeoffs, 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

[2]For example, one can rent machines on Google Cloud supporting Intel's Optane DC Memory today, and Intel has made these devices available to many research groups, including ours.

Problem	(1)	(48h)	(SU)	Work	Depth
BFS	561	12.2	45.9	$O(m)$	$O(\mathsf{d}(G)\log n)$
Weighted BFS	4420	98	45.1	$O(m)^*$	$O(\mathsf{d}(G)\log n)^\ddagger$
Bellman-Ford	3760	82.3	45.5	$O(\mathsf{d}(G)m)$	$O(\mathsf{d}(G)\log n)$
1-Src Widest Path	3479	77.5	44.8	$O(\mathsf{d}(G)m)$	$O(\mathsf{d}(G)\log n)$
1-Src Betweenness	3267	68.5	47.6	$O(m)$	$O(\mathsf{d}(G)\log n)$
$O(k)$**-Spanner**	2219	55.1	40.2	$O(m)^*$	$O(k\log n)^\ddagger$
LDD	985	24.0	41.0	$O(m)^*$	$O(\log^2 n)^\ddagger$
Connectivity	1564	36.2	43.2	$O(m)^*$	$O(\log^3 n)^\ddagger$
Spanning Forest	2439	61.3	38.3	$O(m)^*$	$O(\log^3 n)^\ddagger$
Biconnectivity†	10930	234	46.7	$O(m)^*$	$O(\max(\mathsf{d}(G)\log n, \log^3 n))^\ddagger$
MIS	2308	52.3	44.1	$O(m)^*$	$O(\log^2 n)^\ddagger$
Maximal Matching†	7280	166	43.1	$O(m)^*$	$O(\frac{\log^4 m}{\log\log m})^\ddagger$
Graph Coloring	10880	239	45.5	$O(m)^*$	$O(\log n + L\log\Delta)^*$
Apx Set Cover†	7968	193	41.2	$O(m)^*$	$O(\log^3 n)^\ddagger$
k**-core**	8348	215	38.8	$O(m)^*$	$O(\rho\log n)^\ddagger$
Apx Dens. Subgraph	1930	42.2	45.7	$O(m)$	$O(\log^2 n)$
Triangle Counting†	—	3529	—	$O(m^{3/2})$	$O(\log n)$
PageRank Iteration	1033	23.6	43.5	$O(m)$	$O(\log n)$
PageRank	—	827	—	$O(P_{it}\cdot m)$	$O(P_{it}\log n)$

Table 4: Running times (in seconds) and speedup of our algorithms on the Hyperlink2012 graph using NVRAM. (1) corresponds to the single-threaded time, (48h) corresponds to the running time on 48 cores with hyper-threading, and (SU) is the parallel speedup. Note that the single-threaded times for triangle counting and PageRank are omitted since they did not finish in a reasonable amount of time. Also shown are the work and depth on the PSAM model. We use † to denote that our algorithm uses $O(n + m/\log n)$ words of memory. We use * to denote that a bound holds in expectation and ‡ to denote that a bound holds with high probability or w.h.p. ($O(kf(n))$ cost with probability at least $1 - 1/n^k$). $d(G)$ is the diameter of the graph, Δ is the maximum degree, $L = \min(\sqrt{m}, \Delta) + \log^2 \Delta \log n/\log\log n$, and P_{it} is the number of iterations of PageRank until convergence. In all cases we assume $m = \Omega(n)$.

as much NVRAM as DRAM [75]), 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 [179, 91] (a factor of 12 total).

Our Approach: Parallel Semi-Asymmetric Algorithms. We propose a model for analyzing algorithms in the semi-asymmetric setting. 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 $O(n + m/\log n)$ 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. Most of our algorithms are work-efficient (performing asymptotically the same work as the best sequential algorithm for the problem) and have polylogarithmic depth (parallel time). 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.

Why is assuming $O(n)$ words of internal memory, i.e., internal memory proportional to the number of vertices realistic? As we have mentioned, the NVRAM-equipped machine we use in our work has 8x as much NVRAM as DRAM, and recent work in this area has utilized machines with 16x as much NVRAM as DRAM [75]. Table 5 reports the number of vertices, edges, and the average degree for a collection of large real-world graphs used in our experiments. Observe that the

Graph Dataset	Num. Vertices	Num. Edges	Avg. Degree
LiveJournal	4,847,571	68,993,773	14.3
LiveJournal-Sym	4,847,571	85,702,474	17.8
com-Orkut	3,072,627	234,370,166	76.2
Twitter	41,652,231	1,468,365,182	35.2
Twitter-Sym	41,652,231	2,405,026,092	57.7
ClueWeb	978,408,098	42,574,107,469	43.5
ClueWeb-Sym	978,408,098	74,744,358,622	76.4
Hyperlink2014	1,724,573,718	64,422,807,961	37.3
Hyperlink2014-Sym	1,724,573,718	124,141,874,032	72.0
Hyperlink2012	3,563,602,789	128,736,914,167	36.1
Hyperlink2012-Sym	3,563,602,789	225,840,663,232	63.3

Table 5: Real-world graph inputs used in this thesis, including vertices, edges, and average degree.

average degree for these graphs is usually between 35–70, with the exception of the LiveJournal graph which is quite small, and even this graph has a reasonable average degree of 14–17, well within the NVRAM to DRAM ratio we operate in. Although this assumption may be violated by certain graph classes, like meshes, or road-networks (near-planar graphs), to the best of our knowledge the sizes of such graphs that appear in practice today easily fit within DRAM.

We experiment with implementations of our algorithms on a variety of large-scale real-world graphs. Our implementations are able to scale 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 symmetrized graph). Table 4 shows the running times on the Hyperlink2012 graph using a 48-core machine with 3TB of NVRAM and 375GB of DRAM. Note that we cannot fit the entire Hyperlink2012 graph and run algorithms on this graph in the DRAM of this machine. Compared to the state-of-the-art DRAM-only algorithms from the Graph Based Benchmark Suite (GBBS) [62], our times are 1.03x faster on average and 1.87x faster on average than Galois algorithms [75] (state-of-the-art algorithms designed for NVRAM). Moreover, our algorithms running on NVRAM nearly match the running times of GBBS algorithms running *entirely in DRAM*, with all but three algorithms within 17%, by effectively hiding the costs of repeatedly accessing NVRAM versus DRAM.

Related Work. A significant amount of research has focused on reducing expensive writes to NVRAMs. Early work has designed algorithms for database operators [49, 180, 181]. Blelloch et al. [31, 19, 30] formally 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 [92, 79, 35, 20]. Other models, algorithms, and systems to reduce writes or memory footprint on NVRAMs have also been described [47, 132, 45, 112, 191, 48, 157].

Persistence is a key property of the NVRAMs due to their non-volatility. From the algorithmic perspective, many new persistent data structures have been designed for NVRAMs [22, 21, 14, 161, 137, 52, 74]. There has also been systems research on automatic recovery schemes and transactional memory for NVRAMs [110, 184, 8, 107, 192, 55, 109, 194]. Blelloch et al. [33] introduce a programming model for fault-tolerant programming on NVRAMs. Finally, there has been several recent papers on benchmarking performance on NVRAMs [108, 179, 91].

Parallel graph processing frameworks have received significant attention due to the need to quickly analyze large graphs. The only previous graph processing work targeting NVRAMs is by Gill et al. [75], which we compare with in this thesis.

Paper	Problem	Graph	Memory	Hyper-threads	Nodes	Time
Mosaic [115]	BFS*	2014	0.768	1,000	1	6.55
	Connectivity*	2014	0.768	1,000	1	708
	SSSP*	2014	0.768	1,000	1	8.6
FlashGraph [56]	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 [95]	BFS*	2012	0.064	32	1	900
	BC*	2012	0.064	32	1	800
Slota et al. [168]	Largest-CC*	2012	16.3	8,192	256	63
	Largest-SCC*	2012	16.3	8,192	256	108
	Approx k-core*	2012	16.3	8,192	256	363
Stergiou et al. [170]	Connectivity	2012	128	24,000	1000	341
Gluon [59]	BFS	2012	24	69,632	256	380
	Connectivity	2012	24	69,632	256	75.3
	PageRank	2012	24	69,632	256	158.2
	SSSP	2012	24	69,632	256	574.9
FastSV [193]	Connectivity	2012	393.2	4,456,448	4096	30.0
Us–DRAM	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 6: 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 on DRAM. *These problems are run on directed versions of the graph.

2.5 Processing Massive Web Graphs

We now quantitatively compare the performance of our approach in shared-memory and NVRAM with existing state-of-the-art results in the literature. We will compare our 72-core running times on three very large real-world graphs to running times reported for these graphs by existing work. Table 6 summarizes essential information about our comparison with state-of-the-art existing results in the literature, including the graph problem, the graph being considered, the amount of memory used by the system, the number of hyper-threads used, the number of nodes (machines) used, and the running time in seconds. We note that most of these results process the *directed* versions of these graphs, which have about half as many edges as the symmetrized version.[3] Unless otherwise mentioned, all results we report from the literature use the directed versions of these graphs. To make the comparison fair 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 6.

[3]We note that our algorithms run on the directed version of the graph run between 1.5–2x faster, due to the fact that the set of vertices and edges traversed by a directed graph traversal is between 1.5–2x smaller than by traversing the largest connected component of the undirected graph.

FlashGraph [56] 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 [115] 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-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 [95] 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. [168] 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. [59] 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. [170] describe a connectivity algorithm that runs in $O(\log n)$ 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.

Finally, very recently, Zhang et al. [193] report running times for solving connectivity on the Hyperlink2012 graph on the NSREC Cori supercomputer. They adapt the Shiloach-Vishkin (SV) algorithm, a classic PRAM connectivity algorithm, to distributed memory and propose several modifications which reduce the number of distributed iterations of SV empirically. Their fastest running times for the Hyperlink2012 graph are reported using 4096 nodes of the Cori. Each node uses a Knights-Landing (KNL) processor with 68 cores, which are 4-way hyper-threaded, and each node is equipped with 96GB of DRAM. The total hardware cost is a little over 4.45 million hyper-threads, and 393TB of DRAM (this represents 36% of Cori's capacity for KNL nodes). Compared to their results, our shared-memory running time is 1.2x faster, while we use 3090x fewer hyper-threads, and 393x fewer memory. We note that some of our work in progress improves our connectivity running time to about 8.5s, which represents a 3.5x speedup over their fastest time, with the same magnitude of hardware savings.

We plan to perform a similar comparison as done for DRAM for our codes running in NVRAM, and will discuss these results in the thesis.

2.6 Finished and Proposed Work

The finished work includes the following papers:

- *Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing*, in SPAA'17, with Guy E. Blelloch and Julian Shun [61].
- *Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable*, in SPAA'18, with Guy E. Blelloch and Julian Shun [63]. This work is summarized in Section 2.3.

The proposed work will contain three papers, which are currently in progress.

Semi-Asymmetric Parallel Graph Algorithms for NVRAMs., This work is summarized in Section 2.4, and is joint work with Guy E. Blelloch, Phil Gibbons, Yan Gu, Hongbo Kang, Charlie McGuffey and Julian Shun. A sizable chunk of this work is complete, but there are still some interesting directions to consider, either as part of this work, or as a subsequent effort.

The main question is to obtain work-efficient algorithms for several important problems, like connectivity, maximal matching, and triangle counting, in the new model. For connectivity, our current approach requires $O(n^\epsilon)$ depth when processing sparse graphs, due to relying on LDDs. The reason is that we have to contract to a graph that requires $O(n)$ edges, and if $m = O(n^{1+\epsilon})$ for some $\epsilon > 0$, β must be set to $O(1/n^\epsilon)$ to make the number of cut edges sufficiently small in expectation.

It seems that a recent sampling lemma on k-out sampling essentially solves this problem [86], at least for connectivity, leading to an $O(m+n)$ work algorithm that runs in $O(\log^3 n)$ depth, and only uses $O(n)$ words of internal memory. For maximal matching, our approach currently requires a single bit per-edge, for a filtering structure. Can we show that a similar sampling lemma, like the one for connectivity, is also applicable to maximal matching? A positive answer to this question would likely have consequences in other models of parallel computation with $O(n)$ internal space, such as the Massively Parallel Computation model when machines have $O(n)$ space each.

Finally, for triangle counting, we are aware of a simple $O(m\alpha \log n)$ work algorithm running in $O(\log n)$ depth and $O(n)$ words of internal memory, based on parallel set-intersection (the algorithm does not make any interesting use of the internal memory). α in this context is the arboricity of the graph, or the minimum number of disjoint forests the graph can be decomposed into. However, this

is still a $\log n$ factor away from the fastest sequential algorithm due to Chiba-Nishizeki [51]. Can we improve this result to $O(m\alpha)$ work and $O(\mathsf{polylog}(n))$ depth, using only $O(n)$ words of internal memory?

ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity. In this project, we will perform a thorough experimental investigation into existing connectivity algorithms and techniques for achieving high performance in the shared-memory setting. The fastest existing parallel multicore CPU algorithms are all based on a combination of edge sampling and/or applying techniques such as linking and compressing trees (e.g., such as Union-Find, or the hook-compress paradigm). However, understanding how to combine these different design choices of sampling and connectivity techniques has been left unexplored.

In this project, we will consider three different sampling schemes: k-out Sampling (based on [173], and modified to use random edge-sampling as in [86]), BFS-based sampling, and a new LDD-based sampling approach. We will combine our sampling approach with different connectivity techniques from the literature on Union-Find [139, 94], Shiloach-Vishkin style algorithms, and a set of connectivity algorithms recently proposed by Liu and Tarjan [111]. We believe that this work (in progress) will be the most comprehensive evaluation of parallel connectivity algorithms on shared-memory systems to date. Additionally, our framework will support synthesizing similar implementations for computing a *spanning forest*, and also for performing *incremental connectivity*.

This project is joint work with Changwan Hong and Julian Shun.

Scalable and Memory-Efficient Algorithms for k-Truss. In this project, we will study a parallel algorithm for k-truss decomposition that scales to the largest publicly-available graph using a single multicore with a terabyte of memory. Given an undirected graph G, define the degree of an edge to be the number of triangles it participates in. A k-truss of a graph is a maximal subgraph $H \subseteq G$ s.t. every edge in H has induced degree at least k.

The k-truss problem has been the subject of extensive research over the past few years, being one of the main challenge problems proposed by the MIT GraphChallenge [150]. The reason for this interest is two-fold. First, computing trussness is a challenging graph kernel that requires non-trivial programming techniques and graph processing features, and thus serves as a good benchmark problem to measure the capabilities of a graph processing system. Second, k-truss has many real-world applications in community detection since the structure captured by trusses better captures "well-knit" communities compared to k-core decomposition, and other density measures based purely on edge-density. Despite being called k-Truss, the problem is really to compute *all* of the trusses. The trusses can be represented implicitly by computing a map from each edge to its *trussness*, or the largest k such that the edge participates in a non-empty k-truss. Therefore, we are interested in the problem of computing the trussness value for all edges in the graph.

We start with a baseline implementation based on the bucketing introduced in Julienne [61]. This method already significantly outperforms existing state-of-the-art codes for k-Truss, while being slightly more memory-efficient. To scale to larger graphs, we plan to use several ideas from the memory-efficient hashing and compression literature which will enable us to process the Hyperlink2012 webgraph on a single machine with 1 terabyte of RAM. Storing trussness naively using a hash-based approach is difficult to scale to the Hyperlink2012 graph due to the fact that it contains 225B bidirectional edges, or roughly 112B undirected edges. These edges require roughly $112 \cdot 12$ bytes of memory to store 8 byte edge ids, with a 4 byte trussness value for each undirected edge, which is more than the memory capacity of this machine.

In addition to designing a significantly more memory-efficient algorithm, we obtain good theoretical bounds on the work and depth of our algorithm, showing that it is work-efficient, running in $O(m\alpha)$ work, where α is the arboricity of the graph. We also show that the k-truss problem

as well as the more general nuclei-decomposition problem [151] is P-complete. Our algorithms are work-efficient and run in depth proportional to the peeling complexity of the underlying graph.

This project is joint work with Guy E. Blelloch and Julian Shun.

3 Batch-Dynamic Graph Algorithms

Overview. A major research challenge today is to build algorithms that can efficiently process dynamically evolving graphs. Many graphs found in the real world, like hyperlink graphs, the Twitter graph, and transaction graphs for currencies and marketplaces change over time, and can therefore be modeled as dynamic graphs. In order to understand the nature of public opinion, detect state-sponsored actors in social networks, SEO-manipulation, and malicious transactions in a *timely manner*, or simply to provide accurate, up-to-date information about a network, organizations including governments, internet service providers, and technology companies must solve fundamental problems over dynamically evolving graphs.

Informally, the objective in dynamic graph algorithms is to maintain a property of an underlying graph, such as its connectivity structure, the k-cores of the graph, or the number of triangles incident to each vertex, as the graph evolves. If the dynamic algorithm for the property is significantly faster than static recomputation, then algorithms and services querying the dynamic structure can access the fresh information (e.g., quickly query whether two vertices are connected, or the coreness of a vertex) possibly *orders of magnitude faster* than recomputing the result [90, 190, 2, 104, 176].

However, due to their inherently sequential nature, existing dynamic algorithms cannot scale to high update rates. The poor scalability is because the algorithms are designed to recompute the maintained property after *every update*. The fact that updates are processed sequentially means that a large body of seminal theoretical work on dynamic graph algorithms might not be applicable to situations with high update rates where processing updates in parallel is important. To build practical algorithms that can scale to high update rates, we relax the requirement that the property must be recomputed after every update, and require it to be computed after *every batch of updates*, which can be arbitrarily sized. In this relaxed model, which we refer to as the Batch-Dynamic model, the algorithm can adapt to high update rates by batching together many updates and processing them simultaneously. We note that studying this basic model even without considering parallelism is interesting in its own right.

We start by formally defining a Batch-Dynamic model of computation in the context of dynamic graph problems. Initially, the underlying maintained graph is empty. Computation in the model proceeds over a series of steps, where the i'th step is either a batch of updates, $B_i = \{e_1, \ldots, e_k\}$, or a batch of queries $Q_i = \{q_1, \ldots, q_k\}$. If the batch contains updates, we apply an update algorithm which processes the updates and updates some internal representation in light of the new updates. If the batch contains queries, we run a query algorithm and return the results of the queries to the caller. Since we would like to exploit parallelism over the batch of updates or queries, we consider designing algorithms for both the updates and queries in the TRAM model, and refer to this as the Parallel Batch-Dynamic model. We note that the Batch-Dynamic model is not limited to the TRAM setting—it may be of both theoretical and practical interest to study it in other emerging models of computation, such as MPC [87, 67].

3.1 Related Work

There are only a few results on parallel dynamic algorithms. Earlier results [70, 57] are not work-efficient with respect to the fastest sequential dynamic algorithms, do not support batch updates,

and perform polynomial work per update. Some more recent results such as parallel dynamic depth-first search [98] and minimum spanning forest [103] process updates one at a time, and are therefore not Batch-Dynamic algorithms. Work efficient parallel Batch-Dynamic algorithms include those for the well-spaced point sets problem [6] and those for the dynamic trees problem [146, 3, 178].

We note the line of elegant work on parallel tree algorithms by Sun, Ferzovic and Blelloch [29, 172]. Sun and Blelloch recently utilized these results to develop work-efficient parallel Batch-Dynamic algorithms for dynamic computational geometry problems, including range and segment queries [171]. Finally, our work in part three of this thesis presents a parallel Batch-Dynamic data structure for representing a dynamic graph that supports good work and depth bounds [64].

3.2 Contributions

(1) We will show that Euler tour trees, a data structure introduced by Henzinger and King [84] and Miltersen et al. [128], achieve asymptotically optimal work and optimal depth in the parallel Batch-Dynamic setting.

(2) We will describe parallel Batch-Dynamic algorithms for the dynamic connectivity problem that work-efficient with respect to the classic dynamic connectivity algorithm of Holm et al., and achieve low worst-case depth. Additionally, when the average batch size of edge deletions is large enough, our algorithm can asymptotically improve on the work bound of the Holm et al. algorithm.

We discuss some ideas for other interesting problems that will be studied in this thesis in Section 3.5.

3.3 Forest Connectivity

Parallel Batch-Dynamic Forest Connectivity.. 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, the only Batch-Dynamic data structure for this problem is a recent result by Acar et al. [3]. 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 (ETTs) [84, 128], can be parallelized under batches of edge insertions and deletions.

Our recent work has shown that ETTs can be efficiently parallelized by designing sequence data structures that can be efficiently batch split and batch joined [178]. The data structures designed achieve optimal work-bounds—adding and removing a batch of k edges in $O(k \log(1 + n/k))$ work and $O(\log n)$ depth. We note that the data structure is the *fastest existing solution to the Batch-Dynamic trees problem.* An experimental evaluation of our new data structures have shown 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. The new implementations also outperform the fastest existing sequential dynamic trees data structures, such as link-cut trees and other high-performance sequential implementations, empirically.

3.4 Dynamic Connectivity

Computing the connected components of a graph is a fundamental problem that has been studied in many different models of computation [160, 147, 11, 85]. The ***connectivity problem*** takes as input an undirected graph G and requires an assignment of labels to vertices such that two vertices have the same label if and only if they are in the same connected component. The dynamic version of the 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. In this project we will describe efficient parallel Batch-Dynamic algorithms for maintaining graph connectivity. Our Batch-Dynamic algorithm runs in $O(\log^3 n)$ depth w.h.p. 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 contribution of this work is summarized by the following theorem:

Theorem 1. *There is a parallel Batch-Dynamic data structure which, given batches of edge insertions, deletions, and connectivity queries processes all updates in $O\big(\log n \log\big(1+\frac{n}{\Delta}\big)\big)$ expected amortized work per edge insertion or deletion where Δ is the average batch size of a deletion operation. The cost of connectivity queries is $O(k \log(1+n/k))$ work and $O(\log n)$ depth for a batch of k queries. The depth to process a batch of edge insertions and deletions is $O(\log n)$ and $O(\log^3 n)$ respectively.*

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 ($O(\log^3 n)$ vs. $O(\log^4 n)$, both w.h.p.), 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.

3.5 Finished and Proposed Work

The finished work includes the following papers:

- *Batch-Parallel Euler Tour Trees* with Thomas Tseng, and Guy E. Blelloch (appeared in ALENEX'19) [178].
- *Parallel Batch-Dynamic Graph Connectivity* with Umut A. Acar, Daniel Anderson, and Guy E. Blelloch (appeared in SPAA'19) [4].

Future Work. There are many open problems that are worthy of study in the Batch-Dynamic model. For any problem that admits a sequential dynamic algorithm that runs in $O(f(n))$ time, one can ask whether the problem admits an efficient Batch-Dynamic algorithm, i.e. one which runs in $O(k \cdot f(n))$ work and low depth for a batch of k updates. In fact, due to the fact that an entire batch is provided simultaneously, we can sometimes beat the work bound in the Batch-Dynamic model. For example, in the case of forest connectivity many sequential algorithms run in $O(\log n)$ time per update (there is a matching lower-bound due to Patrascu and Demaine [138]), whereas in the Batch-Dynamic setting, one can perform k updates in $O(k \log(1 + n/k))$ work and $O(\log n)$ depth. In the limit, when $k = O(n)$, the Batch-Dynamic solution is asymptotically as efficient as the fastest parallel connectivity algorithms. Therefore, a natural question is whether we can achieve efficient, or even faster 'trade-off' bounds for problems such as graph connectivity, biconnectivity, minimum spanning tree, maximal independent set, and maximal matching, to name several problems.

Although we stated the Batch-Dynamic model in terms of graph problems, it is of significant theoretical and practical interest to study it in the context of other problems, such as string data structures (of relevance to computational biology), and computational geometry (of relevance to computer graphics and computer vision). Recent work has developed several Batch-Dynamic algorithms for computational geometry problems such as range, segment and rectangle queries [171]. It would be interesting to study Batch-Dynamic algorithms for other problems that admit fast sequential dynamic algorithms, such as Delaunay triangulation [60] and convex hulls [135].

Low Out-Degree Orientation. Another concrete direction for graph problems is to study parallel Batch-Dynamic algorithms for the low-outdegree orientation problem, which have a wide range of applications in designing dynamic graph algorithms for uniformly sparse graphs [42, 102]. Given an undirected graph G with arboricity α, one can always orient (direct) the edges such that the directed out-degree of every vertex is at most α. A dynamic low-outdegree orientation algorithm maintains such an orientation dynamically, as edges are inserted and deleted from the graph. The problem was first studied in a seminal paper by Brodal and Fagerberg [42], who showed that there is an algorithm that maintains an $O(\alpha)$-orientation (an orientation where the max out-degree is always bounded by $O(\alpha)$) with amortized work $O(\log n)$. The particular "work" measure of interest is the number of *flips*, or edge re-orientations that must be made, and their result shows that achieving $O(\alpha)$-outdegree with $O(\log n)$ amortized flips is possible. In this discussion we will talk about flips, but the bounds also hold for update time. Later works showed that a range of tradeoffs is possible for these measures, e.g., obtaining $O(\alpha \log n)$-outdegree with $O(1)$ amortized flips, and $O(\alpha\sqrt{\log n})$-outdegree with $O(\sqrt{\log n})$ amortized flips. A more recent line of work has made progress towards de-amortizing these algorithms, and obtaining algorithms that have low *worst-case* number of flips. E.g., Kopelowitz et al. obtain $O(\alpha \log n)$-outdegree with $O(\log n)$ worst case flips [102]. There is also a recent result by Berglin and Brodal who give a very simple algorithm achieving $O(\log n)$ worst case flips and $O(\alpha)$ out-degree [23].

Why target low-outdegree orientation? For one, it has a wealth of applications to other dynamic problems on uniformly sparse graphs. For example, one can obtain a fully dynamic maximal

matching algorithm with $O(\alpha \log n)$ worst case update time (it is a nice exercise to think about how). Another application is to dynamic MIS [134], and to dynamic graph coloring [169]. Although there are asymptotically faster algorithms for this problem, including a recent result by Solomon that actually achieves constant amortized time per update, the algorithm based on low out-degree orientations is extremely simple and possibly practical to implement. This idea can be extended with some work to a fully dynamic algorithm for maintaining an approximate maximum cardinality matching in $O(\alpha\epsilon^{-2})$ worst case update time [140], which extends an earlier breakthrough of Peng and Gupta [80] by taking into account the density of the input graph. Another reason why studying low-outdegree orientation is attractive is the fact that most algorithms for this problem have a distinctly greedy, or inherently sequential nature, and so parallelizing such algorithms is a challenging and interesting question.

4 Graph Streaming

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., [68, 69, 77, 50, 116, 189]). These graph-streaming systems receive a stream of queries and a stream of 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 [68, 69, 77]. Unfortunately, these frameworks require either blocking queries or updates so that they are not concurrent, or giving up serializability [189]. Another approach is to use snapshots [50, 116]. Existing snapshot-based systems, however, are either highly space-inefficient, or suffer from poor latency for 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 this part of the thesis, we will answer this question by designing a practical and provably-efficient data structure and parallel batch-dynamic updating algorithms for this structure. Our data structure, which is based on representing graphs using purely-functional trees, naturally admits lightweight snapshots, supports concurrently processing both updates and queries, and permits safe parallelism. We will show that the empirical performance of static graph algorithms using this structure is significantly faster than existing graph-streaming systems, and requires only modest overheads over state-of-the-art *static* graph processing systems. Most importantly, the data structure is equipped with a parameter which enables us to naturally improve memory-efficiency, at the cost of a slight increase in the depth of algorithms. This feature enables us to support significantly larger graphs using a limited amount of memory than existing graph streaming systems. We will end by discussing future work to extend our ideas to settings where the graph represented is larger-than-memory, and for implementing practical parallel dynamic graph algorithms, such as those proposed in the parallel Batch-Dynamic model.

Our Approach. In principle, representing graphs using *purely-functional balanced search trees* [1, 133] 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 $O(\log n)$ 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 [149]. 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 $O(\log n + deg(v))$ 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 [165], allowing massive graphs to be stored using just one or two bytes per edge [63]. This approach cannot be used directly on trees. This would all seem to put a search tree representation at a severe disadvantage.

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, 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. We compare to some other chunking schemes, including B-trees [16] and ropes [5, 72, 38, 24] in Section 4.3 after presenting our approach.

4.1 Related Work

Graph Streaming Systems. 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 [68, 69, 77, 186, 9, 155, 154, 153, 129, 46, 183, 174, 158, 44]. Hornet [44], 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 [50, 116, 89, 88].

Although we use purely functional trees to support snapshots, many other systems for supporting persistence and snapshots use version lists [18, 145, 65]. 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 [116] 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.

GraphOne [106] is a system developed concurrently with our work that can run queries on the most recent version of the graph while processing updates concurrently by using a combination of an adjacency list and an edge list. They report an update rate of 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.

Time-Evolving Graph Processing. There are also many systems that have been built for analyzing graphs over time [100, 81, 101, 123, 124, 82, 73, 148, 177, 182]. These systems are similar to processing 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 our work at present, functional data structures are particularly well-suited for this task since it is easy to keep any number of persistent versions by keeping their roots.

Graph Databases. There has been significant research on graph databases (e.g., [43, 99, 156, 142, 105, 66, 130]). The main difference between graph-streaming 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 the need to support transactions, graph databases have significant overhead even for graph analytic queries such as PageRank and shortest paths. McColl et al. [118] show that Stinger is orders of magnitude faster than state-of-the-art graph databases.

4.2 Contributions

(1) We will describe a practical compressed purely-functional data structure for search trees, called the C-tree. C-tree operations support batching and internal parallelism, and have strong theoretical bounds on work and depth.
(2) We will 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. Aspen incorporates optimizations such as flat-snapshotting, which reduce the cost of accessing vertices for global graph algorithms. Aspen supports a superset of the graph processing primitives from Ligra, extending the Ligra interface with primitives for updating graphs.
(3) We will perform an experimental evaluation of Aspen in a variety of regimes over graph datasets at different scales, including the WebDataCommons Hyperlink2012 graph (with over 200 billion edges), showing significant improvements over state-of-the-art graph-streaming frameworks, and modest overhead over static graph processing frameworks.

4.3 C-trees

We now provide some high-level intuition for C-trees, and illustrate some of their properties.

Randomized Chunking. 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

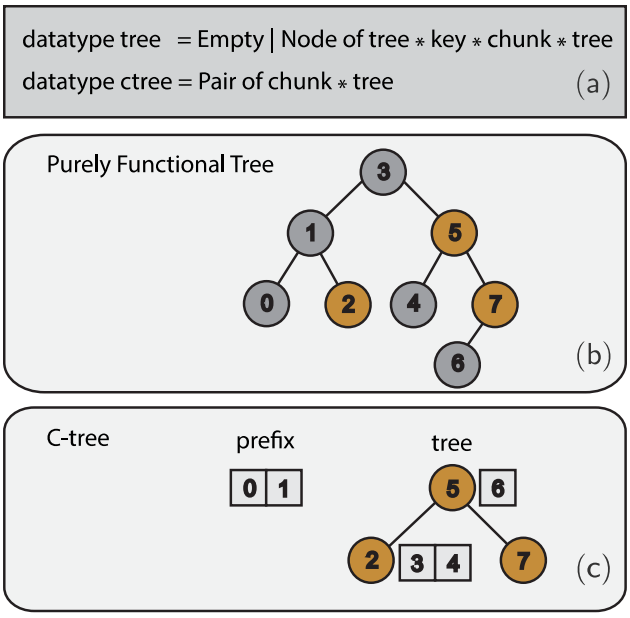

Figure 1: Subfigure (a) defines the C-tree data structure in an ML-like language. Subfigure (b) shows a purely-functional tree where each element in S is stored in a separate tree node. We color the elements in S that are sampled as heads yellow, and color the non-head elements gray. Subfigure (c) illustrates 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.

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 will describe theoretically efficient implementations of tree primitives in the thesis.

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*** $H(E) = \{e \in E \mid h(e) \mod b = 0\}$. 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 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 $O(b \log n)$ with high probability, 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.

Space Savings. Consider the layout of a C-tree compared to a purely-functional tree. As mentioned earlier, we have that 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, and the amount of space needed for tree nodes.

Key-Compression. 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 [165, 187]. 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 [27, 187].

Note that in the common case when b is a constant, the size of each chunk is small ($O(\log n)$ w.h.p.). 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 ($O(\log n)$) 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+ [165].

Performing Updates. We provide a high-level description of how our batch update operation works. The main idea is to first design an implementation of SPLIT on C-trees. The split takes a C-tree, T and an element k, and returns two C-trees, L, R, and a boolean b, where L contains all elements less than k, R consists of all elements greater than k, and b indicates whether k was present in T. Using SPLIT, generating a C-tree $C = A \cup B$ can be done by exposing one of the two C-trees (wlog A), which returns the element at the root of A's tree, and splitting B with respect to this element. This decomposes B into all elements less than the root, and all elements greater than the root. The algorithm then recurses, with one recursive call unioning the root's left subtree in A and the left subtree from splitting B, and similarly with the right subtrees. Finally, the trees from the two recursive calls are merged using a JOIN primitive. Since we store the elements which are sampled as heads in ordinary purely-functional trees, this approach can be implemented using the elegant join-based approach for purely functional trees introduced by Blelloch, Ferzovic, and Sun [29, 172]. We have left some important details out of the high-level description above, which make both the implementation and analysis of the approach more challenging. The main complication is to handle splitting chunks, and handle sending these chunks to the appropriate sub-problem in the recursive step. We will provide these details in the full thesis. Our update bounds can be summarized by the following theorem:

Theorem 2. *Inserting or deleting a batch of q elements into a C-tree of size s can be performed in $O(b^2(k \log((n/k) + 1)))$ expected work and $O(b \log k \log n)$ depth w.h.p. where $k = \min(q, s)$ and $n = \max(q, s)$.*

Relationship to Other Trees. Our data structure is loosely based on a previous sequential approach to chunking [26]. 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.

Another option is to use B-trees [16]. However, the objective of a B-tree is to reduce the height of a search tree to accelerate searches in external memory, whereas our goal is to build a data structure

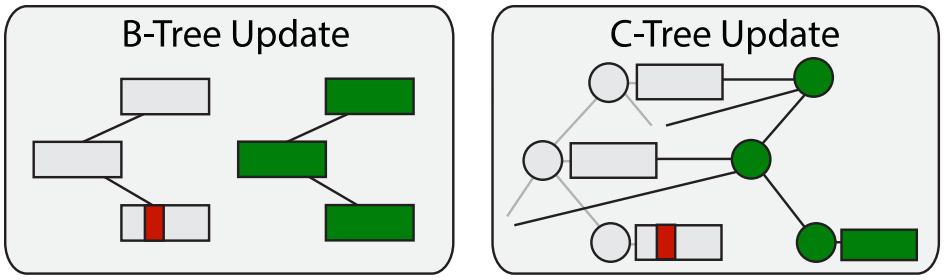

Figure 2: This figure shows the difference between performing a single update in a B-Tree versus an update in a C-tree. The data marked in green is newly allocated in the update. Observe that updating single element in a C-tree in the worst-case requires copying a path of nodes, and copying a single 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.

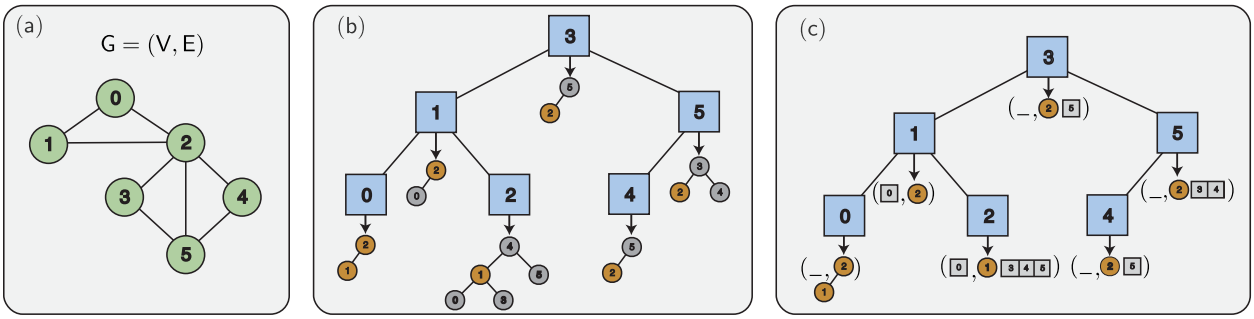

Figure 3: We illustrate how the graph (shown in subfigure (a)) is represented as a simple tree of trees (subfigure (b)) and as a tree of C-trees (subfigure (c)). As in Figure 1, we color elements (in this case vertex IDs) that are sampled as heads yellow. The prefix and tree in each C-tree are drawn as a tuple, following the datatype definition in Figure 1.

that stores contiguous elements 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 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, 72, 38, 24]. 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 hashing to select the heads has an important advantage in simplifying 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.

4.4 Representing Graphs as Trees

Representation. An undirected graph can be implemented 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 $O(\log n)$ overall depth using any balanced tree implementation (w.h.p. using a treap). Figure 3 illustrates the vertex-tree

and the edge-trees for an example graph. 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. We plan to integrate C-tree-based compression for the vertex-tree in ongoing work.

Batch Updates. Inserting and deleting edges are defined similarly, and so we only provide details for edge insertions. 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 our C-tree constructor. Next, we insert a sequence of element updates into the vertex-tree with each $(source, tree)$ pair from the previous sequence. This operation combines existing values (edge-trees) with the new edge-trees by calling UNION, the batch-update operation, on the old edge-tree and new edge-tree. The work and depth of this step can be easily upper-bounded based on our bounds for C-trees, as summarized by the following theorem:

Theorem 3. *Inserting or deleting k edges from a graph represented as a tree of C-tree can be done in $O(k \log n)$ expected work and $O(\log^3 n)$ depth w.h.p.*

An interesting question is whether we can obtain good batch-bounds, of the sort that we obtained for C-tree batch-insertions, for edge insertions and deletions into a graph. This is something that we are studying in ongoing work, and such a result seems possible.

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.

Flat Snapshots for Global Graph Algorithms Algorithms in our framework that use EDGEMAP incur an extra $O(K \log n)$ 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 $O(D \log n)$ depth for a graph with diameter D using a static-graph processing framework [63], a naive implementation using our framework will require performing $O(m + n \log n)$ 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 $O(K \log n)$ 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 $O(\log n)$ 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.

Local Algorithms For 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 $\log n$ (see Table 7), 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.

Graph	n	m	Avg. Deg.	Flat Snap.	Aspen$^{\text{PAM}}$	Aspen (No DE)	Aspen (DE)	Savings
LiveJournal	4.84M	85M	17.8	0.0722	2.77	0.748	0.582	4.75x
com-Orkut	3.07M	234M	76.2	0.0457	7.12	1.47	0.893	7.98x
Twitter	46.5M	2.40B	57.7	0.620	73.5	15.6	9.42	7.80x
ClueWeb	984M	74.7B	76.4	14.5	2271	468	200	11.3x
Hyperlink2014	1.72B	124.1B	72.0	25.6	3776	782	363	10.4x
Hyperlink2012	3.56B	225.8B	63.3	53.1	6889	1449	702	9.81x

Table 7: Statistics about our input graphs, including number of vertices (n), number of edges (m), average degree, and 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$^{\text{PAM}}$**, **Aspen (No DE)**, and **Aspen (DE)** show the amount of memory in GBs required to represent the graph using uncompressed trees (essentially the same as those in the PAM library), 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.

4.5 Aspen

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.

Aspen is implemented in C++ and uses PAM [172] 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 needed for implementing batch update operations. 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, and plan to explore this idea in the full thesis. 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 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.

4.6 Experimental Results

We now briefly present some of the main experimental results showing the advantages of our work. We use weight-balanced trees as the underlying balanced tree implementation for all purely-functional trees in this work [29, 172]. 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 Sizes. Table 7 lists the graphs we use, and 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

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 8: 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. The times for batch deletions are similar to the time for insertions. All times are on 72 cores with hyper-threading.

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 9: 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.

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. 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. We will present more detailed experimental analysis of the choice of chunk size, and its effect on graph algorithms in the thesis.

Update Performance. Table 8 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. The running time can be calculated by dividing the batch size by the throughput. 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.

Comparison with STINGER. The results in Table 9 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.

Other results. We will show the following additional results in the thesis:

- Algorithms implemented using Aspen, which include global graph algorithms such as breadth-first search, betweenness centrality, low-diameter decomposition, and maximal matching, as well as local graph algorithms, such as 2-Hop queries and local graph clustering queries are all scalable, achieving between 32–78x speedup across inputs.
- Updates and queries can be run concurrently in Aspen with only a slight increase in latency (about 3% in the worst case). The performance is measured while running a sequential update stream on a single hyper-thread, while the remaining hyper-threads are running a stream of parallel breadth-first search queries (one after the other).
- Aspen outperforms Stinger by 1.8–10.2x in terms of graph algorithm performance (running a breadth-first search from a given vertex), while using 8.5–11.4x less memory.
- Aspen outperforms LLAMA by 2.8–7.8x in terms of graph algorithm performance (running a breadth-first search from a given vertex) while using 1.9–3.5x less memory.
- Aspen is competitive with state-of-the-art *static* graph processing systems, including Ligra, GAP [17], and Galois [131], ranging from being 1.4x slower to 30x faster.

4.7 Proposed Work: Aspen Improvements, and extension to NVRAM

Finally, we end this section by discussing our ongoing effort to improve Aspen, and extend our work to settings where the dynamic graph we are streaming cannot fit within DRAM. We start by outlining some of the rough-edges around Aspen, and proposed fixes for these issues that will make our work more robust.

Deterministic vs. Random Updates. Blandford and Blelloch presented an elegant deterministic data structure for *compact ordered sets* [26]. An ordered set data structure represents a set S of size n from an ordered universe U of size m. There is an information theoretic lower-bound, which says that any data structure representing such a set must use $\Omega(n \log \frac{m+n}{n})$ bits (since there are $\binom{m}{n}$ possible size-n sets, one must take $\log \binom{m}{n}$ bits to represent such a set for the sparse and likely case when $n \leq m/2$). Suppose the information theoretic lower-bound for a data structure is I. A succinct data structure requires $I + o(I)$ space. A compact data structure requires $O(I)$ space, i.e., it is a constant-factor away from the information theoretic lower bound.

Blandford and Blelloch present a compact data structure for representing ordered sets, which requires space $O(n \log \frac{m+n}{n})$ bits. Their approach is based on performing a deterministic chunking of the input sequence, and using difference coding to represent the ordered sequence within every chunk. The approach is quite similar to the one we use in our C-trees (our work is inspired by this earlier work). A natural question then is whether we can obtain similar bounds as C-trees, while using a deterministic chunking scheme like the one in [26]. Adapting the scheme from [26] would give us a *parallel compact ordered set* data structure that supports batch-updates. Aside from being an interesting theoretical question, this could also have consequences for Aspen in practice, such as lowering Aspen's memory usage, due to having the size of all blocks in the deterministic structure being within B and $4B$ due to deterministic chunking. This would enable pool-allocation of every array size between B and $4B$, avoiding wasting memory when allocating blocks for trees.

Batch Bounds for Graph Updates. Our bounds for graph updates currently require $O(k \log n)$ work (in expectation) to process a batch of k edge updates (insertions or deletions). An interesting question is whether we can achieve a better batch bound, of the form $O(k \log(m/k + 1))$ which would be helpful when the batch size becomes large. The difficulty comes from the fact that we are using a two-level tree structure. Each edge-tree guarantees a batch-bound on its internal (at most n) edges, and the vertex-tree also guarantees a batch-bound on updates to the vertices, so intuitively one would expect such a bound to hold when performing k arbitrary edge updates. The difficulty is

to handle the case where the k updates are concentrated in a small number of edge-trees, and to argue that this does not hurt the overall work. Although understanding this question is probably more relevant in theory, it would be good to show that the graph data structures we design have stronger asymptotic properties when processing large batches of updates.

Aspen on NVRAM. Future systems are likely to contain a significant amount of NVRAM in addition to a smaller amount of DRAM. In such systems, assuming that the entire dynamic graph fits within the DRAM may be an unrealistic assumption. However, such systems are likely to still have a modest amount of DRAM—between 8–16x less DRAM than NVRAM for example. How do we design fast, provably-efficient, and NVRAM-friendly graph streaming systems in this new setting?

One idea is to use our C-tree approach to design a hybrid data structure that utilizes both DRAM and NVRAM. For simplicity, let us ignore the graph setting and suppose that we are just storing a dynamic dataset on a collection of elements that is too large to fit entirely within DRAM. Let n, the size of the tree, be larger than DRAM, and at most the size of the NVRAM. Suppose the NVRAM is 16x larger than DRAM, and that the NVRAM has an effective cache-line size of 256. Then, we can apply C-trees with a chunking parameter of 1024. Instead of storing tails in DRAM, we can store them in NVRAM. If each element requires four bytes to store, the expected size of a tail is 1024, which is larger than the effective cache-line size of the NVRAM device. The tree on the sampled elements will be stored in DRAM, and the value for each element stores a handle to a block allocated on NVRAM. Note that the tree portion of the C-tree contains $n/1024$ sampled elements in expectation, with the remaining elements stored in tails. Since vertex-tree nodes require 56 bytes (due to padding and pointers), the fraction of DRAM used for the vertex-tree is $(16/1024) \cdot 56 = .875$, which is within the bounds for our internal memory. We can likely make the vertex-tree node size 48, which would allow for a smaller block size when the data uses all of the NVRAM.

There are several desirable properties of this approach. First, finding a single element requires at most a *single* block read from NVRAM in the worst case (just one read in the case where the block size is the effective cache-line size). This is because a lookup first performs a tree-search on the tree stored in DRAM. If the key is found in the tree, the search can stop without fetching from NVRAM. Otherwise, there is exactly one block that needs to be read from NVRAM to detect whether the search-key is present or not. Second, note that a single update to a C-tree affects at most one block in terms of the elements contained in that block. Updating this block requires writing a new NVRAM block containing the elements of the old block, and the newly added element. The remaining blocks along the path to the affected block can all be copied, which require no NVRAM writes, and can be re-used. Note that avoiding NVRAM writes for path-copying requires storing the reference-count for a block in DRAM. Understanding whether such an optimization, which eliminates NVRAM writes during a path-copy in exchange for a small amount of metadata stored in DRAM is an interesting question. Finally, it would be very interesting to do a head-to-head comparison between our C-tree approach and highly optimized implementations of B-trees, as well as other tree data-structures, to better understand the design and performance of hybrid data structures that utilize both DRAM and NVRAM.

5 Proposed Schedule

- November–January: start work on polishing Aspen, and extending it to NVRAM (Section 4.7). Also survey low-outdegree orientation papers and try to obtain some results here (Section 3.5).
- December 1st: Submit the paper on connectivity benchmarking to VLDB (Section 2.6).

- February: If the connectivity benchmarking paper is rejected, submit it to the third deadline for SIGMETRICS. Also submit two to three papers not directly related to this thesis to SPAA and ICML, with co-authors. Consider submitting NVRAM graph algorithm paper to SPAA (Section 2.6).
- February: If we have results for low-outdegree orientation, consider writing them up for SPAA (Section 3.5).
- May: Submit Aspen on NVRAM paper to VLDB. If they reject, we can send it to PPoPP, followed by PLDI (Section 4.7).
- July: Submit k-truss work to KDD or VLDB. The results will be on massive graphs, using memory-efficient techniques that seem more broadly applicable than just for k-Truss (Section 2.6).
- July–October: work on writing thesis.
- October: Defend thesis.

6 Conclusion

How will this thesis answer the thesis statement put forth in Section 1? We want to understand whether shared-memory parallel graph algorithms can solve a diverse set of fundamental problems on large-scale static/dynamic/streaming graphs quickly, provably-efficiently, and scalably, at low cost, using a modest amount of computational resources.

It is hard to say what is fundamental, but one can measure the empirical importance of a problem by counting the number of papers written on the problem, or the number of papers that use the problem as an important sub-routine. By that measurement, many of the problems that we study in this thesis are surely important to the algorithms and parallel algorithms communities, if not fundamental. By virtue of running on the largest publicly available graphs, we will show that our results solve these problems on large-scale datasets. By comparing to the fastest state-of-the-art running times for various problems and benchmarks, we will have shown that our results are fast and scalable (using self-speedup, or speedup over the fastest sequential codes as the measure of scalability). Since our results all have good provable bounds on their work and depth we will show that they are all provably-efficient. Finally, since our results only use a single commodity multicore machine, of the sort one can easily rent using a cloud service for a few dollars an hour, our results will only use a modest amount of computational resources. I hope that this thesis will adequately defend this thesis statement and show that shared-memory technology is a promising path toward solving large-scale problems on massive graph datasets.

A Notation, Models, and Cost

Notation. 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 $n-1$. For undirected graphs we use $N(v)$ to denote the neighbors of vertex v and $deg(v)$ to denote its degree. For directed graphs, we use $in\text{-}deg(v)$ and $out\text{-}deg(v)$ to denote the in and out-neighbors of a vertex v. We use $\mathsf{diam}(G)$ 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. Δ is used to denote the maximum degree of the graph. 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 column and compressed-sparse row formats as ***CSC*** and ***CSR*** respectively.

Parallel Model. The parallel model used in this thesis is the TRAM, which can be understood to be roughly equivalent to the CRCW PRAM (it admits a cross-simulation with the CRCW PRAM, losing only an $O(\log^* n)$-factor in the depth). A program in the TRAM consists of a set of threads sharing an unbounded memory. Each process is essentially just a Random Access Machine—it runs a program stored in memory that uses standard RAM instructions and uses a constant number of registers. A computation terminates once a process calls an `end` instruction. A computation can be forked using the `fork`, which takes a positive integer k and forks k new child threads. Each child thread receives a unique identifier in $[1, \ldots, k]$ in a special register. When a process runs `fork`, it suspends until all children terminate. Note that whenever possible, we use $k = 2$ in our algorithms, and explicitly specify whenever this is not the case. We refer to [28] for more details about the model.

Parallel Cost Measures. A computation in the TRAM can be viewed as a series-parallel DAG in which each instruction is a vertex, sequential instructions are composed in series, and the forked subthreads are composed in parallel. The ***work*** of a computation is the number of vertices and the ***depth*** (***span***) is the length of the longest path in the DAG.

Ligra, Ligra+, and Julienne. We make use of the Ligra and Ligra+ frameworks for shared-memory graph processing in this thesis and review components from these frameworks here [162, 165]. Ligra provides data structures for representing a graph $G = (V, E)$, ***vertexSubset***s (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 $O(\log n)$ 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 $O(\log n)$ 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 $O(\log n)$ depth version, but at the cost of $O(\mathit{in\text{-}deg}(v))$ depth.

Experimental Setup. The 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 Cilk Plus to express parallelism and 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 [37]. We note that our programs use a work-stealing scheduler that we implemented, based on the algorithm of Arora et al. [13]. 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.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.

[2] U. A. Acar. *Self-adjusting computation*. PhD thesis, Citeseer, 2005.

[3] U. A. Acar, V. Aksenov, and S. Westrick. Brief announcement: Parallel dynamic tree contraction via self-adjusting computation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[4] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala. Parallel batch-dynamic graph connectivity. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019.*, pages 381–392, 2019.

[5] U. A. Acar, A. Charguéraud, and M. Rainey. Theory and practice of chunked sequences. In *European Symposium on Algorithms (ESA)*, pages 25–36, 2014.

[6] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoglu. Parallelism in dynamic well-spaced point sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

[7] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 1986.

[8] M. Alshboul, H. Elnawawy, R. Elkhouly, K. Kimura, J. Tuck, and Y. Solihin. Efficient checkpointing with recompute scheme for non-volatile main memory. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(2):18, 2019.

[9] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.*, 11(6):691–704, Feb. 2018.

[10] R. Anderson and E. W. Mayr. A P-complete problem and approximations to it. Technical report, 1984.

[11] A. Andoni, C. Stein, Z. Song, Z. Wang, and P. Zhong. Parallel graph connectivity in log diameter rounds. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2018.

[12] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems*, 34(2):115–144, 2001.

[13] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)*, 34(2), Apr 2001.

[14] H. Attiya, O. Ben-Baruch, P. Fatourou, D. Hendler, and E. Kosmas. Tracking in order to recover: Recoverable lock-free data structures. *arXiv preprint arXiv:1905.13600*, 2019.

[15] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *VLDB*, 5(5):454–465, 2012.

[16] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, Sep 1972.

[17] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[18] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[19] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[20] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IPDPS*, 2018.

[21] N. Ben-David, G. E. Blelloch, M. Friedman, and Y. Wei. Delay-free concurrency on faulty persistent memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2019.

[22] N. Ben-David, G. E. Blelloch, Y. Sun, and Y. Wei. Multiversion concurrency with bounded delay and precise garbage collection. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 241–252. ACM, 2019.

[23] E. Berglin and G. S. Brodal. A Simple Greedy Algorithm for Dynamic Graph Orientation. In Y. Okamoto and T. Tokuyama, editors, *28th International Symposium on Algorithms and Computation (ISAAC 2017)*, volume 92 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[24] J.-P. Bernardy. The haskell Yi package, 2008.

[25] M. Birn, V. Osipov, P. Sanders, C. Schulz, and N. Sitchinava. Efficient parallel and external matching. In *Euro-Par*, 2013.

[26] D. K. Blandford and G. E. Blelloch. Compact representations of ordered sets. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–19, 2004.

[27] D. K. Blandford, G. E. Blelloch, and I. A. Kash. An experimental analysis of a compact graph representation. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 49–61, 2004.

[28] G. E. Blelloch and L. Dhulipala. Introduction to parallel algorithms. http://www.cs.cmu.edu/~realworld/slidesS18/parallelChap.pdf, 2018. Carnegie Mellon University.

[29] G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2016.

[30] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[31] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Efficient algorithms with asymmetric read and write costs. In *European Symposium on Algorithms (ESA)*, 2016.

[32] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[33] G. E. Blelloch, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. The parallel persistent memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[34] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallelism in randomized incremental algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[35] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[36] G. E. Blelloch, R. Peng, and K. Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

[37] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5), Sept. 1999.

[38] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: An alternative to strings. *Softw. Pract. Exper.*, 25(12), 1995.

[39] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *International World Wide Web Conference (WWW)*, 2004.

[40] U. Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2), 2001.

[41] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *International World Wide Web Conference (WWW)*, pages 107–117, 1998.

[42] G. S. Brodal and R. Fagerberg. Dynamic representations of sparse graphs. In *Workshop on Algorithms and Data Structures*, pages 342–351. Springer, 1999.

[43] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[44] 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)*, pages 1–7, Sep. 2018.

[45] T. Cai, F. Chen, Q. He, D. Niu, and J. Wang. The matrix kv storage system based on nvm devices. *Micromachines*, 10(5):346, 2019.

[46] Z. Cai, D. Logothetis, and G. Siganos. Facilitating real-time graph mining. In *International Workshop on Cloud Data Management (CloudDB)*, pages 1–8, 2012.

[47] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simhadri. Write-avoiding algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[48] Q. Chen, H. Lee, Y. Kim, H. Y. Yeom, and Y. Son. Design and implementation of skiplist-based key-value store on non-volatile memory. *Cluster Computing*, 22(2):361–371, 2019.

[49] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.

[50] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *European Conference on Computer Systems (EuroSys)*, pages 85–98, 2012.

[51] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[52] N. Cohen, R. Guerraoui, and M. I. Zablotchi. The inherent cost of remembering consistently. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[53] R. Cole, P. N. Klein, and R. 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.

[54] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[55] A. Correia, P. Felber, and P. Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–282, 2018.

[56] D. M. Da Zheng, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity SSDs. In *FAST*, 2015.

[57] S. K. Das and P. Ferragina. An $o(n)$ work EREW parallel algorithm for updating MST. In *European Symposium on Algorithms (ESA)*, 1994.

[58] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *PLDI*, pages 752–768, 2018.

[59] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *PLDI*, pages 752–768. ACM, 2018.

[60] O. Devillers, S. Meiser, and M. Teillaud. Fully dynamic delaunay triangulation in logarithmic expected time per operation. *Computational Geometry*, 2(2):55–80, 1992.

[61] L. Dhulipala, G. E. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[62] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2018.

[63] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 393–404, 2018.

[64] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. *arXiv preprint arXiv:1904.08380*, 2019.

[65] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[66] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer. Weaver: a high-performance, transactional graph database based on refinable timestamps. *Proceedings of the VLDB Endowment*, 9(11):852–863, 2016.

[67] D. Durfee, L. Dhulipala, J. Kulkarni, R. Peng, S. Sawlani, and X. Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. *arXiv preprint arXiv:1908.01956*, 2019.

[68] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing (HPEC)*, pages 1–5, 2012.

[69] G. Feng, X. Meng, and K. Ammar. Distinger: A distributed graph data structure for massive dynamic graph processing. In *IEEE International Conference on Big Data (BigData)*, pages 1814–1822, 2015.

[70] P. Ferragina and F. Luccio. Batch dynamic algorithms for two graph problems. *International Conference on Parallel Architectures and Languages Europe (PARLE)*, 1994.

[71] J. T. Fineman. Nearly work-efficient parallel algorithm for digraph reachability. In *STOC*, 2018.

[72] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly threaded parallelism in manticore. *J. Funct. Program.*, 20(5-6):537–576, Nov. 2010.

[73] F. Fouquet, T. Hartmann, S. Mosser, and M. Cordy. Enabling lock-free concurrent workers over temporal graphs composed of multiple time-series. In *ACM Symposium on Applied Computing (SAC)*, volume 8, pages 1054–1061, 2018.

[74] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, volume 53, pages 28–40, 2018.

[75] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali. Single machine graph analytics on massive datasets using intel optane DC persistent memory. *CoRR*, abs/1904.07162, 2019.

[76] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[77] O. Green and D. A. Bader. custinger: Supporting dynamic graph algorithms for gpus. In *IEEE Conference on High Performance Extreme Computing (HPEC)*, pages 1–6, 2016.

[78] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-completeness Theory*. Oxford University Press, Inc., 1995.

[79] Y. Gu, Y. Sun, and G. E. Blelloch. Algorithmic building blocks for asymmetric memories. In *European Symposium on Algorithms (ESA)*, 2018.

[80] M. Gupta and R. Peng. Fully dynamic (1+ e)-approximate matchings. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 548–557. IEEE, 2013.

[81] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *European Conference on Computer Systems (EuroSys)*, pages 1:1–1:14, 2014.

[82] T. Hartmann, F. Fouquet, M. Jimenez, R. Rouvoy, and Y. Le Traon. Analyzing complex data in motion at scale with temporal graphs. In *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 596–601, 2017.

[83] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *SPAA*, 2014.

[84] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *ACM Symposium on Theory of Computing (STOC)*. ACM, 1995.

[85] J. Holm, K. De Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[86] J. Holm, V. King, M. Thorup, O. Zamir, and U. Zwick. Random k-out subgraph leaves only $o(n/k)$ inter-component edges. *CoRR*, abs/1909.11147, 2017.

[87] G. F. Italiano, S. Lattanzi, V. S. Mirrokni, and N. Parotsidis. Dynamic algorithms for the massively parallel computation model. In *spaa*, pages 49–58. ACM, 2019.

[88] A. Iyer, L. E. Li, and I. Stoica. Celliq : Real-time cellular network analytics at scale. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 309–322, 2015.

[89] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *International Workshop on Graph Data Management Experiences and Systems (GRADES)*, pages 5:1–5:6, 2016.

[90] R. Iyer, D. Karger, H. Rahul, and M. Thorup. An experimental study of polylogarithmic, fully dynamic, connectivity algorithms. *Journal of Experimental Algorithmics (JEA)*, 6:4, 2001.

[91] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, et al. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[92] R. Jacob and N. Sitchinava. Lower bounds in the asymmetric external memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[93] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[94] S. V. Jayanti, R. E. Tarjan, and E. Boix-Adserà. Randomized concurrent set union and generalized wake-up. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 187–196, 2019.

[95] S.-W. Jun, A. Wright, S. Zhang, S. Xu, et al. Grafboost: Using accelerated flash storage for external graph analytics. In *ISCA*, pages 411–424, 2018.

[96] R. M. Karp and V. Ramachandran. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. A)*, chapter Parallel Algorithms for Shared-memory Machines. MIT Press, Cambridge, MA, USA, 1990.

[97] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. In *STOC*, 1984.

[98] S. Khan. Near optimal parallel algorithms for dynamic DFS in undirected graphs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[99] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica. ZipG: A memory-efficient graph store for interactive queries. In *ACM SIGMOD International Conference on Management of Data*, pages 1149–1164, 2017.

[100] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *International Conference on Data Engineering (ICDE)*, pages 997–1008, 2013.

[101] U. Khurana and A. Deshpande. Storing and analyzing historical graph data at scale. In *International Conference on Extending Database Technology (EDBT)*, pages 65–76, 2016.

[102] T. Kopelowitz, R. Krauthgamer, E. Porat, and S. Solomon. Orienting fully dynamic graphs with worst-case time bounds. In *International Colloquium on Automata, Languages, and Programming*, pages 532–543. Springer, 2014.

[103] T. Kopelowitz, E. Porat, and Y. Rosenmutter. Improved worst-case deterministic parallel dynamic minimum spanning forest. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[104] I. Krommidas and C. Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *Journal of Experimental Algorithmics (JEA)*, 12:16, 2008.

[105] 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)*, pages 830–841, 2016.

[106] P. Kumar and H. H. Huang. GraphOne: A data store for real-time analytics on evolving graphs. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 249–263, 2019.

[107] L. Lersch, W. Lehner, and I. Oukid. Persistent buffer management with optimistic consistency. In *International Workshop on Data Management on New Hardware*, pages 14:1–14:3, 2019.

[108] J. Liu and S. Chen. Initial experience with 3D XPoint main memory. In *IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 300–305, 2019.

[109] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, 2018.

[110] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. Compiler-directed failure atomicity for nonvolatile memory. Technical report, Virginia Polytechnic Institute, 2019.

[111] S. Liu and R. E. Tarjan. Simple concurrent labeling algorithms for connected components. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 3:1–3:20, 2019.

[112] X. Liu, Y. Hua, X. Li, and Q. Liu. Write-optimized and consistent rdma-based nvm systems. *arXiv preprint arXiv:1906.08173*, 2019.

[113] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *UAI*, 2010.

[114] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 1986.

[115] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *EuroSys*, 2017.

[116] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *IEEE International Conference on Data Engineering (ICDE)*, pages 363–374, 2015.

[117] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[118] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A performance evaluation of open source graph databases. In *Workshop on Parallel Programming for Analytics Applications*, pages 11–18, 2014.

[119] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2), Oct. 2015.

[120] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *booktitle=Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[121] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science*, 1(1), 2015.

[122] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1), 2003.

[123] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen. ImmortalGraph: A system for storage and analysis of temporal graphs. *ACM TOS*, pages 14:1–14:34, 2015.

[124] O. Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.

[125] G. L. Miller, R. Peng, A. Vladu, and S. C. Xu. Improved parallel algorithms for spanners and hopsets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 192–201, 2015.

[126] G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decompositions using random shifts. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[127] G. L. Miller and V. Ramachandran. A new graph triconnectivity algorithm and its parallelization. *Combinatorica*, 12(1), Mar 1992.

[128] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoretical Computer Science (TCS)*, 130(1), 1994.

[129] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, Sept. 2016.

[130] Neo4j.

[131] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471, 2013.

[132] R. Nissim and O. Schwartz. Revisiting the i/o-complexity of fast matrix multiplication with recomputations. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 714–716, 2019.

[133] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[134] K. Onak, B. Schieber, S. Solomon, and N. Wein. Fully dynamic MIS in uniformly sparse graphs. In *icalp*, pages 92:1–92:14, 2018.

[135] M. H. Overmars and J. Van Leeuwen. Maintenance of configurations in the plane. *Journal of computer and System Sciences*, 23(2):166–204, 1981.

[136] S. Pai and K. Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *OOPSLA*, pages 1–19, 2016.

[137] W. Pan, T. Xie, and X. Song. Hart: A concurrent hash-assisted radix tree for dram-pm hybrid memory systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[138] M. Patrascu and E. D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.

[139] M. Patwary, P. Refsnes, and F. Manne. Multi-core spanning forest algorithms using the disjoint-set data structure. In *IPDPS*, 2012.

[140] D. Peleg and S. Solomon. Dynamic $(1+\varepsilon)$-approximate matchings: a density-sensitive approach. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 712–729. Society for Industrial and Applied Mathematics, 2016.

[141] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6), 2002.

[142] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX Conference on Annual Technical Conference (ATC)*, pages 41–52, 2012.

[143] D. Prountzos, R. Manevich, and K. Pingali. Synthesizing parallel graph programs via automated planning. In *PLDI*, pages 533–544, 2015.

[144] V. Ramachandran. A framework for parallel graph algorithm design. In *Optimal Algorithms*, 1989.

[145] D. P. Reed. Naming and synchronization in a decentralized computer system, 1978.

[146] J. H. Reif and S. R. Tate. Dynamic parallel tree contraction (extended abstract). In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1994.

[147] O. Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, Sept. 2008.

[148] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.

[149] Y. Saad. *Iterative methods for sparse linear systems*, volume 82. SIAM, 2003.

[150] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, et al. Graphchallenge. org: Raising the bar on graph analytic performance. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[151] A. E. Sariyuce, C. Seshadhri, and A. Pinar. Parallel local algorithms for core, truss, and nucleus decompositions. *CoRR*, abs/1704.00386, 2017.

[152] W. Schudy. Finding strongly connected components in parallel using $O(\log^2 N)$ reachability queries. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008.

[153] D. Sengupta and S. L. Song. EvoGraph: On-the-fly efficient mining of evolving graphs on GPU. In *International Supercomputing Conference (ISC)*, pages 97–119, 2017.

[154] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan. GraphIn: An online high performance incremental graph processing framework. In *Euro-Par*, pages 319–333, 2016.

[155] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating dynamic graph analytics on GPUs. *Proc. VLDB Endow.*, 11(1):107–120, Sept. 2017.

[156] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *ACM SIGMOD International Conference on Management of Data*, pages 505–516, 2013.

[157] Y. Shen and Z. Zou. Efficient subgraph matching on non-volatile memory. In *International Conference on Web Information Systems Engineering*, pages 457–471. Springer, 2017.

[158] X. Shi, B. Cui, Y. Shao, and Y. Tong. Tornado: A system for real-time iterative analysis over evolving data. In *ACM SIGMOD International Conference on Management of Data*, pages 417–430, 2016.

[159] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 1982.

[160] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[161] T. Shull, J. Huang, and J. Torrellas. Autopersist: an easy-to-use java nvm framework based on reachability. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 316–332, 2019.

[162] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2013.

[163] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[164] J. Shun, L. Dhulipala, and G. E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014.

[165] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *Data Compression Conference (DCC)*, 2015.

[166] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *IEEE International Conference on Data Engineering (ICDE)*, 2015.

[167] G. M. Slota, S. Rajamanickam, and K. Madduri. *Supercomputing for Web Graph Analytics.* Apr 2015.

[168] G. M. Slota, S. Rajamanickam, and K. Madduri. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *IPDPS*, 2016.

[169] S. Solomon and N. Wein. Improved dynamic graph coloring. *CoRR*, abs/1904.12427, 2019.

[170] S. Stergiou, D. Rughwani, and K. Tsioutsiouliklis. Shortcutting label propagation for distributed connected components. In *WSDM*, 2018.

[171] Y. Sun and G. E. Blelloch. Parallel range and segment queries with augmented maps. *arXiv preprint:1803.08621*, 2018.

[172] Y. Sun, D. Ferizovic, and G. E. Belloch. Pam: Parallel augmented maps. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 290–304, 2018.

[173] M. Sutton, T. Ben-Nun, and A. Barak. Optimizing parallel graph connectivity computation via subgraph sampling. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 12–21, 2018.

[174] T. Suzumura, S. Nishii, and M. Ganse. Towards large-scale graph stream processing platform. In *International World Wide Web Conference (WWW)*, pages 1321–1326, 2014.

[175] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. on Computing*, 14(4), 1985.

[176] R. E. Tarjan and R. F. Werneck. Dynamic trees in practice. *Journal of Experimental Algorithmics (JEA)*, 14:5, 2009.

[177] M. Then, T. Kersten, S. Günnemann, A. Kemper, and T. Neumann. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. *Proc. VLDB Endow.*, 10(8):877–888, Apr. 2017.

[178] T. Tseng, L. Dhulipala, and G. Blelloch. Batch-parallel Euler tour trees. *Algorithm Engineering and Experiments (ALENEX)*, 2019.

[179] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Persistent memory i/o primitives. In *International Workshop on Data Management on New Hardware*, pages 12:1–12:7, 2019.

[180] S. D. Viglas. Adapting the B^+-tree for asymmetric I/O. In *Advances in Databases and Information Systems (ADBIS)*, 2012.

[181] S. D. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.

[182] K. Vora, R. Gupta, and G. Xu. Synergistic analysis of evolving graphs. *ACM Trans. Archit. Code Optim.*, 13(4):32:1–32:27, Oct. 2016.

[183] K. Vora, R. Gupta, and G. 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)*, pages 237–251, 2017.

[184] C. Wang, S. Chattopadhyay, and G. 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*, pages 33–44, 2019.

[185] K. Wang, G. H. Xu, Z. Su, and Y. D. 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)*, pages 387–401, 2015.

[186] M. Winter, R. Zayer, and M. Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *IEEE Conference on High Performance Extreme Computing (HPEC)*, pages 1–7, 2017.

[187] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes (2nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., 1999.

[188] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7, 2017.

[189] C. Yin, J. Riedy, and D. A. Bader. A new algorithmic model for graph analysis of streaming data. In *International Workshop on Mining and Learning with Graphs*, 2018.

[190] C. D. Zaroliagis. Implementations and experimental studies of dynamic graph algorithms. In *Experimental algorithmics*, pages 229–278. Springer, 2002.

[191] M. Zarubin, P. Damme, T. Kissinger, D. Habich, W. Lehner, and T. Willhalm. Integer compression in nvram-centric data stores: Comparative experimental analysis to dram. In *ACM International Workshop on Data Management on New Hardware*, page 11, 2019.

[192] L. Zhang and S. Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[193] Y. Zhang, A. Azad, and Z. Hu. FastSV: a distributed-memory connected component algorithm with fast convergence. *CoRR*, abs/1910.05971, 2019.

[194] T. Zhou, P. Zardoshti, and M. Spear. Brief announcement: Optimizing persistent transactions. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 169–170, 2019.

[195] W. Zhou. A practical scalable shared-memory parallel algorithm for computing minimum spanning trees. Master's thesis, KIT, 2017.