Research Statement

Laxman Dhulipala

1 Introduction

Graphs arise naturally in many practical settings. Examples include social networks and Web graphs, protein-interaction graphs in molecular biology, transaction graphs in finance and economics, and brain networks in neuroscience. In recent years, ***graph processing systems***—systems that enable users to represent and quickly solve problems on graphs—have been developed to address the widespread need to understand graphs. For instance, graph processing systems at Google process similarity graphs to solve connected components, minimum spanning forest, and a variety of other graph problems, often using the results in downstream machine learning tasks. In the sciences and humanities, graph processing has been used to understand gene co-expression, study the functional organization of the brain, and even to understand the historical social network surrounding Sir Francis Bacon in early modern Britain. In today's increasingly connected world, graph analysis using these systems provides users with a powerful lens through which to explore and understand complex real-world data. The goal of my research is to build graph processing systems which are *fast, easy and affordable to use, scale to very large graphs, and support a wide-variety of problems and use-cases, such as graphs which change over time.*

Large graphs pose a significant challenge for graph processing systems today. For instance, the largest publicly-available graph today is the WebDataCommons hyperlink graph, which contains over 3.5 billion vertices and 128 billion directed edges. The key to achieving high performance when processing a graph of this size is ***parallelism***. Existing high-performance systems exploit parallelism by designing algorithms for supercomputers or distributed clusters which are notoriously difficult to program and reason about, can only solve a handful of simple problems, and are also prohibitively costly. If graphs continue to grow rapidly in size as the trends in Figure 1 suggest, future graph processing systems will need to support analyzing much larger graphs, and thus understanding how to design systems that can solve a broad set of problems on very large graphs quickly and cost-effectively is of great interest.

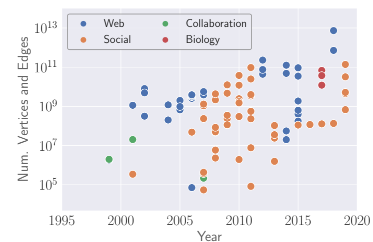

Figure 1: Plot of the total number of vertices and edges in graphs versus the year that the graph was collected. Graph data is sourced from the SNAP and LAW datasets.

Many graphs also *change over time*, but most graph processing systems assume a static setting and do not support streaming and dynamic graphs. A streaming graph processing system represents a graph as it is updated, and makes consistent snapshots of the graph available to graph analytics queries that arrive concurrently with the updates. In dynamic graph processing, the objective is to dynamically maintain properties of the graph, such as the connected components, or the number of cliques in the graph, as the graph is modified by the updates. Using a streaming or dynamic system to analyze a changing graph can provide *orders of magnitude performance improvements* over simply using a static graph processing system. Streaming and dynamic systems also enable *real-time understanding* of graphs, which is critical to important applications such as fraud-detection in financial networks and real-time recommendation. Unfortunately, existing streaming and dynamic graph solutions have significant limitations, such as poor memory utilization, contention, and high latency, which makes them prohibitively slow and unable to scale to large graphs.

My Approach. I believe the way to design graph processing systems is by utilizing parallel algorithms and data structures that are ***theoretically-efficient*** (possess strong provable bounds on their theoretical costs) and also ***practically-efficient***. Specifically, my approach is to design algorithms and data structures in the shared-memory setting, and analyze them in terms of their *work and depth*, two fundamental cost measures for parallel algorithms. I then build systems based on provably-efficient implementations of these algorithms and data structures designed for *multicore machines* (machines containing multiple CPUs sharing a single address space). Such machines equipped with hundreds of cores and terabytes of memory can be rented for tens of dollars on the hour from cloud vendors such as Amazon Web Services. Using this approach, I have designed *theoretically-efficient static, dynamic, and streaming graph processing systems* which are *an order of magnitude faster* than existing systems and support a *much broader set of problems*, while using *up to three orders of magnitude fewer cores and two orders of magnitude less memory*.

In the following sections, I will describe my research organized by contributions to different types of graph processing and algorithm design, and will draw attention to three themes in my work: designing high-level programming frameworks for graph programming, designing compressed and memory-efficient data structures, and bridging the gap between theory and practice in algorithms. Finally, I will conclude by presenting my research vision and discussing future directions stemming from my research.

2 Parallel Graph Processing

Existing approaches for processing very large graphs rely on distributed or external memory systems, but graph algorithms, which often have poor locality and perform little work per memory request, are challenging to implement in distributed and external-memory settings due to the high cost of data movement. In recent years, the widespread availability of multicore machines with dozens to hundreds of cores and terabytes of memory has made a ***shared-memory approach*** to parallel graph processing—designing parallel implementations for multicore machines—a viable alternative. However, designing fast shared-memory implementations is still a significant challenge which requires applying parallel algorithms that are both *practically- and theoretically-efficient*. There are several reasons that algorithms with good theoretical guarantees are desirable. For one, they are *robust*, as even adversarial inputs will not cause them to perform extremely poorly. Furthermore, they can be designed on pen-and-paper by exploiting properties of the problem instead of tailoring solutions to the particular dataset at hand. Theoretical guarantees also make it likely that the algorithm will continue to perform well even if the underlying data changes.

The two fundamental efficiency measures for parallel algorithms are the ***work***, or the total number of operations that an algorithm performs, and the ***depth***, or the longest chain of sequential dependencies in the algorithm. The gold-standard is a ***work-efficient*** parallel algorithm, which performs asymptotically the same amount of work as the fastest sequential algorithm for the task. Work-efficient algorithms are desirable since they perform well even when there is not much parallelism available (e.g., there are few cores or parallel workers available). Unfortunately, existing graph processing systems largely ignore theoretically-efficient parallel algorithms, and do not provide work-efficient implementations of even basic graph problems, such as connectivity, and simply do not solve many fundamental graph problems.

The Graph Based Benchmark Suite (GBBS). It is natural to ask whether theoretically-efficient parallel algorithms are relevant to developing fast shared-memory graph processing systems. To study this question, I have developed *The Graph Based Benchmark Suite (GBBS)* [12, 16], a problem-based benchmark suite and parallel graph processing system which provides *scalable and work-efficient implementations of algorithms for over 20 fundamental graph problems*. The answer to this question from my research is **yes**—many of the algorithmic techniques and specific algorithms suggested in the 80s and early 90s, or even 70s, are very relevant today, and lead to fast parallel implementations for multicore machines. Specifically, I have shown that theoretically-efficient parallel graph algorithms are not only competitive with the fastest existing hand-tuned codes, but are often the fastest algorithms of any that have been implemented for a wide variety of graph problems (shown in Table 1). Good theoretical bounds also ensure predictable performance as graph sizes increase. For instance, on the largest publicly available graph today, the WebDataCommons hyperlink graph[1], with over 200 billion edges, GBBS implementations can solve all of the problems shown in the table below in a matter of seconds to minutes using a commodity multicore machine with a terabyte of memory.

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

Table 1: Important graph problems considered in the Graph Based Benchmark Suite (GBBS), covering a broad range of techniques and application areas. Problems highlighted in green were previously not solvable by parallel graph processing systems on any graph of the scale of the WebDataCommons hyperlink graph.

[1] http://webdatacommons.org/hyperlinkgraph/

The key approach to designing GBBS algorithms that are theoretically-efficient and highly scalable in practice is a careful combination of *algorithm, interface, and systems design*. Specifically, the implementations are developed using the *GBBS library*, which provides a high-level C++ interface for graph processing (extending the Ligra (PPoPP'13) interface) with a rich set of *functional primitives* that are *parallel by default*, and have *clearly defined parallel cost bounds* on their work and depth. This approach enables writing high-level codes that are simultaneously simple and high-performance, by virtue of using highly-optimized primitives. Another benefit is that optimizations, such as lossless graph compression [19], are implemented transparently to high-level user code, and can thus be utilized without changing the implementation. The theoretically-efficient implementations obtained using this approach make it possible for the first time for a single-machine graph processing system to solve challenging problems such as k-cores, biconnectivity, strongly connected components, and minimum spanning forest on a graph with hundreds of billions of edges in just a few minutes.

To validate these results in industry, I have collaborated closely with the Graph Mining team led by Vahab Mirrokni at Google Research NYC, and today, both the GBBS library and GBBS algorithms such as k-core and connectivity are used at Google to improve end-to-end graph clustering latencies. My research on GBBS, which was awarded the **Best Paper Award at SPAA'18**, provides the most comprehensive study of parallel graph algorithms to date, and I believe it will be a valuable building block for the development of many other practical and theoretically-efficient parallel graph algorithms.

Performance and Cost-Effectiveness. Compared to existing solutions for very large graph processing, GBBS is more scalable and much more cost-effective. Figure 2 plots the running time and the number of hyper-threads (one hyper-thread costs about \$0.05 per hour on average on AWS) for GBBS, as well as existing external memory and distributed memory results for solving connectivity on the WebDataCommons hyperlink graph. Connectivity is shown for this example since it is one of the few problems which all existing systems capable of processing this graph solve. Note that the memory usage follows a very similar trend. Purely considering the running-times, the work-efficient shared-memory connectivity algorithm from GBBS (in green) outperformed every existing result for connectivity on this graph. Factoring in the hardware cost in terms of hyper-threads, compared to the external memory results, the implementation is an order of magnitude faster while still using a modest number of hyper-threads, and compared to the distributed-memory results, it is faster than every result, while using *orders of magnitude fewer hyper-threads*.

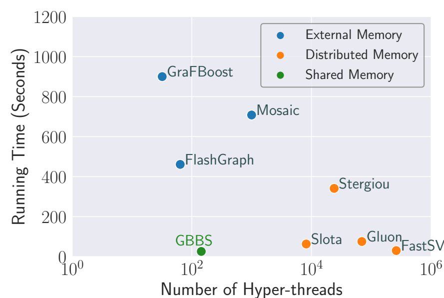

Figure 2: Hardware used (number of hyper-threads, in log-scale) versus running time in seconds for solving connectivity on the WebDataCommons hyperlink graph.

Julienne: An Interface for Ordered Graph Algorithms. The approach underlying GBBS is nicely illustrated in one of the core parts of its programming library, *Julienne* [11]. Julienne is the first high-level framework that can accelerate a broad class of *ordered graph algorithms* that must maintain a dynamic mapping between vertices and a set of *ordered buckets* over the course of the algorithm's execution. Examples include algorithms such as the Δ-stepping algorithm for single-source shortest paths, approximate set cover, and k-core, amongst several others. Julienne provides a high-level interface for parallel bucketing, which enables concise implementations (under 100 lines of code) of ordered algorithms for these problems that are *work-efficient* and significantly outperform existing custom parallel implementations. The new implementations utilize a practical algorithm for the bucketing interface which admits strong theoretical bounds and is highly scalable. I believe that this approach of (i) understanding what *abstract operations* a certain class of algorithms perform, and (ii) designing *efficient interfaces* to support the required operations enables *simpler algorithms and faster implementations*, and ultimately makes it easier to understand and teach the algorithms.

Julienne has influenced the design of other graph-processing systems such as the GraphIt DSL (OOPSLA'18), for which I have designed an ordered programming extension with optimizations that significantly improve the performance of ordered shortest-path algorithms on high-diameter graphs [21]. Julienne has also been used in the design and implementation of shared-memory parallel graph algorithms, such as for k-tip and k-wing decompositions in bipartite graphs, as well as my research on work-efficient algorithms k-clique peeling and k-clique densest subgraph [18].

Graph Processing using Non-volatile Memory. I have used GBBS to develop *Sage* [15], an efficient graph processing system for non-volatile main memory technologies (NVRAMs) motivated by the question of *how to efficiently process graphs that do not fit in the main memory of a machine*. NVRAMs provide an attractive set of features for graph analytics, including byte-addressability, low idle power, and high memory-density. NVRAM systems today can be equipped with an order of magnitude more NVRAM than traditional memory (DRAM), potentially allowing very large graph problems to be solved on a single machine. However, a significant challenge in achieving high performance is to account for the fact that NVRAM writes can be much more expensive than NVRAM reads. To address this problem, Sage introduces an approach in which the graph is stored as a read-only data structure (in NVRAM), and the amount of mutable memory (in DRAM) is kept proportional to the number of vertices. Despite the restrictiveness of this read-only approach, I have developed simple and effective techniques for obtaining over a dozen parallel graph algorithms in Sage which perform the same amount of work as their shared-memory counterparts in GBBS. Sage performs well in practice, and can solve all of these problems on the WebDataCommons hyperlink graph using a commodity machine equipped with Optane DC Persistent Memory, significantly outperforming the fastest existing NVRAM-based systems. Importantly, *Sage achieves performance close to that of GBBS run using DRAM* by effectively hiding the costs of repeatedly accessing NVRAM versus DRAM, thus providing a *cost-effective and performant path to analyze trillion-edge graphs using single-machine graph analytics*.

Other High-Performance Frameworks and Libraries. I have recently designed the *ConnectIt framework* [14], which can express over 300 new implementations of parallel connectivity, the fastest of which can solve this problem on the WebDataCommons hyperlink graph in under 10 seconds (3x faster than GBBS which was the previous state-of-the-art for any system, including state-of-the-art supercomputing systems). I have also developed a GPU-specific implementation of this framework called *GConn* [17], which achieves an average speedup over 2.47x over the fastest (highly-optimized) existing GPU connectivity implementations. GBBS, Julienne, Sage, ConnectIt, and other systems I have developed are built using *ParlayLib* [7], a shared-library of parallel building blocks which I have recently open-sourced. ParlayLib includes parallel primitives for sequences, such as map, reduce, prefix-sum (scan), filter, permutation, and sorting, as well as parallel hash-tables, histograms, and many other useful primitives, all with strong parallel cost-bounds. I believe that ParlayLib will be useful when developing other scalable and theoretically-efficient systems.

3 Streaming Graph Processing

In recent years, there has been growing interest in developing systems for processing *streaming graphs* due to the fact that many real-world graphs change and must be processed in real-time. These graph processing 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. Unfortunately, existing systems such as STINGER (HPEC'12) require either blocking queries or updates so that they are not concurrent, or giving up serializability. Some snapshot-based systems exist, however, they are highly space-inefficient or experience high latency on updates. The challenge is to design a dynamic graph representation which (i) supports lightweight snapshots, (ii) supports efficient batch updates, and (iii) only incurs modest space overheads.

The Aspen Graph-Streaming Framework. To develop a practical and theoretically-efficient solution to this problem, I have developed the Aspen multicore graph-streaming framework [13], which was awarded the **Distinguished Paper Award at PLDI'19**. Aspen supports a superset of the graph processing primitives from GBBS, extends the GBBS interface with primitives for updating graphs, and is based on a novel representation of graphs using *nested trees*. 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). Importantly, the representation enables *lightweight snapshots*, which can be used to concurrently process queries and updates while ensuring that the data structure is safe for parallelism. Although purely-functional trees could be used in Aspen, and would provide strong theoretical guarantees for both graph queries and updates, they suffer from large memory and performance overheads. To address these limitations, I have developed the *C-tree* data structure, a compressed purely-functional data structure for search trees, which enables the nested-trees representation to be space- and cache-efficient, while preserving all the other advantages of using purely-functional trees.

The key idea of a *C*-tree is to chunk the elements represented by the tree and store each chunk contiguously

in an array. Chunking improves locality when processing the elements, enables data-compression to be applied within a chunk, and also helps save space by reducing the number of tree nodes. I have designed a rich set of operations for processing and updating C-trees that provide batching, internal parallelism, and have strong theoretical bounds on their work and depth, which lead to efficient algorithms for processing and updating the graph representation based on nested trees. In practice, I have shown that Aspen can concurrently process a stream of updates and queries with update latency well under a millisecond, and throughput for batch updates ranging between *105M–442M updates per second*—two orders of magnitude higher than the best existing multicore graph-streaming system today, STINGER. Aspen's graph representation using C-trees is also up to 11x more memory efficient than STINGER, enabling larger significantly graphs to be processed using the same amount of memory. *I have shown that Aspen can concurrently apply parallel graph analytics and apply updates to the WebDataCommons hyperlink graph using only a commodity multicore machine with 1 terabyte of memory*. To this date, this graph is orders of magnitude larger than any graph that has been processed by any other graph-streaming system in the literature.

4 Batch-Dynamic Graph Algorithms

In applications where updates happen at rapid rates, e.g., when millions of users simultaneously interact with a website, traditional (sequential) dynamic algorithms require serializing the changes made and processing them *one at a time*, missing an opportunity to exploit the parallelism afforded by processing batches of changes. To obtain scalable dynamic algorithms, I have developed parallel algorithms in the ***Batch-Dynamic Model***, a generalization of the sequential dynamic model that enables scaling to high update rates. The batch-dynamic model relaxes the requirement of a sequential dynamic algorithm to recompute the property after every update, and requires it to be computed after every *batch* of updates, which can be arbitrarily sized. In contrast with the streaming setting, where the objective is to make the new graph available for subsequent queries, the objective of a batch-dynamic algorithm is to dynamically update the results of a computation in response to the changes in the graph. This model has been informally used in the literature by systems such as Naiad (SOSP'13) and Kickstarter (ASPLOS'17), although my work is the first to provide algorithms with non-trivial guarantees for fundamental graph problems in this model.

Batch-Dynamic Trees and Connectivity. I have studied efficient parallel algorithms for the classic dynamic trees problem, which is to maintain a dynamically evolving forest while supporting connectivity queries. I have shown that the Euler tour tree, a classic data structure for this problem, can be efficiently parallelized in the batch-dynamic setting, and also yield practical implementations [20]. Theoretically, for a batch of k updates over a forest of n vertices, the data structure achieves $O(k \lg(1 + n/k))$ expected work and $O(\lg n)$ depth with high probability—which achieves asymptotic improvements over simply applying a sequential dynamic algorithm for large batches. I have also developed the parallel RC-tree data structure [2], which supports path queries, subtree queries, as well as a variety of other nonlocal queries. This tree is obtained by a new algorithmic framework for dynamizing static *round-synchronous* parallel algorithms, which may have more applications in developing theoretically-efficient batch-dynamic algorithms.

I have also studied efficient parallel batch-dynamic algorithms for the dynamic connectivity problem, which build on batch-dynamic trees. Perhaps the best known sequential algorithm for dynamic connectivity is the elegant level-set algorithm of Holm, de Lichtenberg, and Thorup (HDT). I have developed a parallel batch-dynamic algorithm for connectivity [1] that is work-efficient with respect to the HDT algorithm for small batch sizes, and is asymptotically faster when the average batch size is sufficiently large. The core idea of the new algorithm is to parallelize the HDT algorithm by performing a careful exploration of the non-tree edges which allows the algorithm amortize the work while ensuring low (poly-logarithmic) depth. I believe that this result could be made practically-efficient, and am interested in designing practical implementations for batch-dynamic connectivity based on my theoretical results.

Batch-Dynamic Triangle and k-Clique-Counting. I have developed parallel batch-dynamic algorithms [10] for the triangle counting and k-clique counting problems. These fundamental subgraph counting problems have numerous applications in social network analysis and bioinformatics. For triangle counting, which is widely used in practice, my work introduces two parallel batch-dynamic algorithms, one which is efficient for sparse graphs, and another which uses fast matrix multiplication which is more efficient for dense graphs. Evaluating a multicore implementation of the batch-dynamic triangle counting algorithm on large graphs

showed that the new theoretically-efficient algorithm is up to an order of magnitude faster than the previous state-of-the-art parallel batch-dynamic triangle counting algorithm, which has weak theoretical guarantees.

5 Research Vision and Future Directions

The increasing use of technology throughout society has made it necessary to store and process unprecedented amounts of data from rich physical and digital sources. In the future, progress in many areas, ranging from advances in healthcare and genomics, to rapidly identifying disinformation and fraud in social or financial networks will depend on scalable algorithms and systems for understanding this data.

I believe the way to tackle this challenge is through a *unified approach which simultaneously takes into account the programmability, performance, and theoretical aspects of a solution*. Building solutions based upon strong theoretical foundations will be key, since theoretically-efficient parallel algorithms can significantly outperform ad-hoc solutions, especially as data sizes grow or come from unknown distributions. Furthermore, the solutions should be flexible and adapt to the user's needs without sacrificing performance, while also ensuring portability across different machines and architectures. I have found this approach to be highly applicable to designing fast graph processing solutions in my research, and am very excited to apply it to develop scalable algorithms and systems to address the data-processing challenges of the future.

Parallel Clustering. Clustering is a fundamental topic studied in statistics, machine learning, and computer science, and hundreds of papers, surveys, and books have been written on the subject over the past forty years. However, little is known about the parallel complexity of clustering, and designing high-quality clustering algorithms that scale to massive datasets remains an important open problem. I intend to study the parallel complexity of popular clustering methods, such as hierarchical agglomerative clustering (HAC), both in the graph and metric settings. To circumvent the parallel hardness results that I believe can be shown for these problems, I am interested in developing parallel approximation algorithms that are work-efficient and highly scalable. I plan to work together with my collaborators at Google Research and MIT to design a clustering benchmark that enables us to understand the tradeoff between performance, clustering quality, and cost.

Efficient Parallel Algorithms. I am interested in understanding the capabilities and limits of parallel algorithms. For instance, I have designed a near-optimal algorithm for graph connectivity [3] in the Massively Parallel Computation (MPC) model, a theoretical model for frameworks like MapReduce, Hadoop, and Spark. Specifically, for a diameter D graph, the algorithm achieves a round-complexity (analgous to depth) of $O(\lg D)$, which is conditionally optimal for graphs with $D = \Omega(\lg n)$. I have also shown non-trivial upper and lower-bounds for batch-dynamic connectivity in the MPC setting [8]. In the future, I am very interested in designing efficient parallel algorithms for fundamental problems for graphs and other settings, such as metric spaces, and computational geometry algorithms, both in the shared-memory and MPC settings.

Models of Parallel Computation for Emerging Hardware. Theoretical models for new hardware can provide valuable insights by showing the limits of what is possible using novel functionality in the new technology. Motivated by the capabilities of RDMA-based systems, I have designed a model called the *Adaptive Massively Parallel Computation* (AMPC) model [4], which augments the MPC model with the ability to perform a limited amount of *adaptive reads* from a read-only shared memory within a round of computation. I have shown that AMPC algorithms can achieve exponential speedups over existing MPC algorithms [4], and are significantly faster than MPC algorithms in practice [5]. I have also recently developed a model called the *Read-Only Semi-External Model (ROSE)* [6], motivated by my work on Sage, to study semi-external graph algorithms over a read-only graph that simultaneously optimize work, depth, and I/O complexity. In the future, I am interested in developing parallel models of computation to understand the capabilities and limitations of models that minimize data movement (e.g., processing in-memory).

High-Level Frameworks. A major goal in my research is to derive a deep understanding of complex problems and to provide appropriate abstractions. This approach has appeared in my work on graph processing, including Julienne, GBBS, Sage, and Aspen, and I believe it can be applied to many other problems. For instance, I am interested in developing a framework for expressing *contraction-based* algorithms, for problems such as parallel clustering, parallel minimum cut, and parallel graph partitioning (a problem I have previously studied [9]). Such a framework will allow us to understand commonalities amongst the algorithms and their analysis, perform performance optimizations within the framework which can accelerate all of the algorithms, and yield scalable solutions which impact both research and practice.

High-Performance Graph Databases. I am interested in developing a high-performance graph database based on Aspen that supports transactions, and features such as complex node and edge attributes. Although transaction processing has been extensively studied in the database literature, there may be many interesting opportunities to design faster algorithms tailored to graphs, and designed for emerging technologies (such as NVRAMs). A graph database would also be a valuable testbed for new batch-dynamic graph algorithms, and these algorithms can be exposed to users of the database as graph indices. A research vision along this direction is a hybrid transaction and analytics graph database system that can efficiently apply parallel graph algorithms on read-only snapshots, support dynamically-updated graph indices, and enable fast (potentially batched) transactions, with strong provable-bounds on the costs of all operations. I believe that the algorithms and systems developed in my research are a significant step toward realizing such a system.

References

[1] (by alphabetical order) Umut Acar, Daniel Anderson, Guy E. Blelloch, and **Laxman Dhulipala**. Parallel Batch-Dynamic Graph Connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 381–392, 2019.

[2] (by alphabetical order) Umut Acar, Daniel Anderson, Guy E. Blelloch, **Laxman Dhulipala**, and Sam Westrick. Parallel Batch-Dynamic Trees via Change Propagation. In *European Symposium on Algorithms (ESA)*, pp. 2:1–2:23, 2020.

[3] (by alphabetical order) Soheil Behnezhad, **Laxman Dhulipala**, Hossein Esfandiari, Jakub Lacki, and Vahab Mirrokni. Near-optimal massively parallel graph connectivity. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 1615–1636. IEEE, 2019.

[4] (by alphabetical order) Soheil Behnezhad, **Laxman Dhulipala**, Hossein Esfandiari, Jakub Lacki, Vahab Mirrokni, and Warren Schudy. Massively Parallel Computation via Remote Memory Access. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 59–68, 2019.

[5] (by alphabetical order) Soheil Behnezhad, **Laxman Dhulipala**, Hossein Esfandiari, Jakub Lacki, Vahab Mirrokni, and Warren Schudy. Parallel Graph Algorithms in Constant Adaptive Rounds: Theory meets Practice. *PVLDB*, 13(13):3588–3602, 2020.

[6] (by alphabetical order) Guy E Blelloch, **Laxman Dhulipala**, Phillip B Gibbons, Yan Gu, Charlie McGuffey, and Julian Shun. The Read-Only Semi-External Model. In *To appear in SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2021.

[7] Guy E. Blelloch, Daniel Anderson, and **Laxman Dhulipala**. ParlayLib — A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 507–509, 2020.

[8] (by alphabetical order) **Laxman Dhulipala**, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. Parallel Batch-Dynamic Graphs: Algorithms and Lower Bounds. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1300–1319, 2020.

[9] (by alphabetical order) **Laxman Dhulipala**, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. Compressing Graphs and Indexes with Recursive Graph Bisection. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 1535–1544, 2016.

[10] (by alphabetical order) **Laxman Dhulipala**, Quanquan C. Liu, Julian Shun, and Shangdi Yu. Parallel Batch-Dynamic k-Clique Counting. In *To appear in SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2021.

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

[12] **Laxman Dhulipala**, Guy E. Blelloch, and Julian Shun. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 293–304, 2018.

[13] **Laxman Dhulipala**, Guy E Blelloch, and Julian Shun. Low-Latency Graph Streaming using Compressed Purely-Functional Trees. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 918–934, 2019.

[14] **Laxman Dhulipala**, Changwan Hong, and Julian Shun. ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms. *Proc. VLDB Endow. (To appear)*, 14(4), 2021.

[15] **Laxman Dhulipala**, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E Blelloch, Phillip B Gibbons, and Julian Shun. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *PVLDB*, 13(9):1598–1613, 2020.

[16] **Laxman Dhulipala**, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. The Graph Based Benchmark Suite (GBBS). In *International Workshop on Graph Data Management Experiences and Systems (GRADES) and Network Data Analytics (NDA)*, pp. 11:1–11:8, 2020.

[17] Changwan Hong, **Laxman Dhulipala**, and Julian Shun. Exploring the Design Space of Static and Incremental Graph Connectivity Algorithms on GPUs. In *ACM Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 55–69, 2020.

[18] Jessica Shi, **Laxman Dhulipala**, and Julian Shun. Parallel Clique Counting and Peeling Algorithms. *arXiv preprint arXiv:2002.10047*, 2020.

[19] Julian Shun, **Laxman Dhulipala**, and Guy E. Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *Data Compression Conference (DCC)*, pp. 403–412, 2015.

[20] Thomas Tseng, **Laxman Dhulipala**, and Guy E. Blelloch. Batch-Parallel Euler Tour Trees. *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 92–106, 2019.

[21] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, **Laxman Dhulipala**, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing Ordered Graph Algorithms with GraphIt. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, pp. 158–170, 2020.