Research Statement

Julian Shun

1 Introduction

In today's data-driven world with rapidly increasing data sizes, performance has become more important than ever before. Reducing the running time of programs lowers overall costs and has been shown to increase worker productivity as well as end-user experience. Alternatively, one can view improving performance as enabling more computation to be performed in a given amount of time, effectively increasing one's computing budget. Parallelism is the key to achieving high performance in computing. Traditionally, high-performance large-scale solutions have been developed using supercomputers or distributed clusters. However, within the past decade, commodity multicore machines have become prevalent, and today these machines support up to terabytes of memory,[1] more than enough for a majority of applications. My research shows that a single multicore machine can solve many large-scale problems in a simple, efficient, and scalable manner.

Writing efficient and scalable parallel programs is notoriously difficult, and often requires significant expertise. To address this challenge, it is crucial to provide programmers with high-level tools to enable them to develop solutions efficiently, and at the same time emphasize the theoretical and practical aspects of algorithm design to allow the solutions developed to run efficiently under all possible inputs, scale across a wide range of processor counts, and scale gracefully to larger data. *My research addresses this challenge using a three-pronged approach consisting of shared-memory programming techniques, algorithm design, and performance analysis for a broad class of irregular problems in computing.* My research results provide evidence that with appropriate programming techniques, frameworks, and algorithms, *shared-memory programs can be simple, fast, and scalable, both in theory and in practice*.

Three broad topics in my research are deterministic parallelism, large-scale graph processing, and bridging the gap between theory and practice in algorithms. These topics span the three areas of programming techniques, algorithm design, and performance analysis. The following sections describe my research contributions in each of three areas. I will then describe a benchmark suite that I developed to measure the quality of solutions based on simplicity, theoretical efficiency, as well as practical performance. Finally, I will conclude with my research vision.

2 Shared-Memory Programming

Ligra: A Lightweight Graph Processing Framework for Shared Memory. I have developed Ligra [16], a high-level framework for implementing large-scale parallel graph algorithms for shared memory. The interface contains just two simple functions, one for mapping functions over vertices (VertexMap) and one for mapping functions over edges (EdgeMap). *With just these two functions, the framework is able to express a broad class of graph algorithms*, including breadth-first search, betweenness centrality, graph eccentricity estimation, graph connectivity, PageRank, and single-source shortest paths. A key feature of Ligra is that it can efficiently process subsets of the vertices and their edges, making it particularly well-suited for graph traversal problems, which visit possibly small subsets of the vertices on each step. The framework implements an optimization for switching between a read-based execution and a write-based execution based on the size of the active set. Ligra supports the same ease of programmability as previous high-level graph processing frameworks, while being much faster. The performance of Ligra code is competitive with highly-optimized hand-written code, while requiring much lower programming effort to write. Ligra is able to process the largest publicly-available graphs on a single machine and is the first high-level framework to advocate processing large graphs in shared memory; there have been several other shared-memory graph

[1] For example, the Intel Sandy Bridge-based Dell PowerEdge R930 can be configured with up to 96 cores and 6 Terabytes of memory.

processing systems developed subsequently, e.g., Galois (SOSP 2013), X-Stream (SOSP 2013), PRISM (SPAA 2014), Polymer (PPoPP 2015), GraphMat (VLDB 2015), and Ringo (SIGMOD 2015).

I have used the Ligra framework to parallelize all existing graph eccentricity algorithms in the literature (for each vertex, compute or estimate its distance to the furthest reachable vertex in the graph), and evaluate their performance on large-scale graphs [11]. Graph eccentricities have many applications in the analysis of social networks, routing networks, and biological networks. I have also introduced a new eccentricity estimation algorithm based on executing multiple breadth-first searches in parallel, and the algorithm uses word-level parallelism to improve performance. On real-world graphs, the new algorithm *outperforms all existing algorithms in terms of running time and accuracy by up to orders of magnitude* [11].

I have also used Ligra to parallelize a variety of algorithms for local graph clustering, where the work performed is proportional to the size of the cluster(s) output instead of the entire graph [23]. These are the first parallel algorithms for graph clustering with a local work bound. I show that the algorithms achieve good parallel speedups on large real-world graphs and also prove strong theoretical guarantees about them.

Deterministic Parallelism. *One of the key challenges in parallel programming is dealing with nondeterminism arising from the parallel program and/or the parallel machine and its runtime environment.* Nondeterminism arises from race conditions in the program (concurrent accesses to the same data with at least one being a write), and makes it hard for programmers to debug and reason about the correctness/performance of their code. One way to obtain determinism in parallel programs is to not have any races. While this approach is reasonable for certain problems, in general it can be overly restrictive as it is often useful and efficient to have shared data. The goal in my research is to develop less restrictive and more efficient ways to obtain determinism.

There has been significant previous work on obtaining determinism using various approaches, including using special-purpose hardware, modifying compilers, runtime systems and/or operating systems, and designing new programming languages. In contrast to most previous work, my research focuses on designing building blocks and programming techniques for simplifying deterministic parallel programming that can be used with the *existing computing stack*, making determinism more accessible. In other words, programmers do not have to install special programming languages, compilers, runtime systems or operating systems, nor do they need access to special-purpose hardware. My research advocates a form of determinism called *internal determinism* [5]. Informally, given an abstraction level, a program is internally deterministic if key intermediate steps of the program are deterministic with respect to the abstraction level. Internal determinism has many benefits, including leading to external determinism and implying a sequential semantics, which in turn leads to many advantages such as ease of reasoning about code, verifying correctness and debugging.

Deterministic Reservations. One of the main approaches that I have developed for deterministic parallelism is the *deterministic reservations* framework for parallelizing sequential iterative algorithms [5]. The approach consists of two phases—in the *reserve* phase, the iterates concurrently mark all of the data that they affect, and in the *commit* phase, iterates whose mark is still written on all of its affected data proceed with the computation on the data. Determining successful reservations is done in a deterministic manner, so that for a given round the same iterates succeed/fail on every execution. Parallel algorithms written in this framework return the same answer as their sequential counterparts, which gives determinism, and allows the parallel and sequential algorithms to be interchanged when necessary. The algorithms developed are also very simple, as the programmer only needs to specify the `reserve` and `commit` functions called by each iterate in the two corresponding phases, as well as associated data structures. This approach leads to very simple and efficient parallel solutions for many problems that outperform their sequential counterparts with just a modest number of cores.

Theoretically, I have analyzed the dependence structure of several "sequential" iterative algorithms, showing that the dependence is low, which implies a high amount of parallelism. A tool that I use in the theoretical analyses is the *dependence graph*, which is a directed acyclic graph capturing the dependencies

among the iterates of the computation. Intuitively, a computation is parallelizable if the longest path in its dependence graph is shallow. By using an elegant argument to relate the dependence structure of the random permutation and list contraction problems to that of a random binary search tree, I am able to prove a tight logarithmic depth[2] bound for the two problems [22]. Using a variety of probabilistic analysis techniques, I have also shown logarithmic or polylogarithmic depth bounds for a variety of other "sequential" algorithms, including tree contraction [22], greedy maximal independent set and maximal matching [6], as well as incremental sorting, planar Delaunay triangulation, low-dimensional linear programming, closest pairs, and smallest enclosing disk [7]. For maximal independent set and Delaunay triangulation, proving such bounds has been open for at least a decade. I have also shown work-efficient implementations of all of the algorithms, giving deterministic parallel algorithms that are efficient both in theory and also in practice.

Phase-Concurrency. I have proposed the idea of *phase-concurrency* in parallel programs to achieve determinism. In phase-concurrency, operations are grouped such that operations in the same group are allowed to proceed concurrently because they commute, but operations in different groups are not. As an example of this idea, I have designed a deterministic phase-concurrent hash table (insert, delete, and search are in separate groups) that guarantees that the state of the table at any quiescent point is independent of the ordering of operations [17]. This hash table can naturally be used in many deterministic parallel programs, such as Delaunay refinement, removing duplicates, and edge contraction, as the programs already separate non-commutative operations into different phases to achieve determinism. I have shown that the hash table can support operations more efficiently than previous concurrent hash tables when operations are restricted to be phase-concurrent. The hash table is based on linear probing, and relies on history-independence for determinism.

3 Shared-Memory Algorithm Design

Bridging Theory and Practice. The traditional goal in parallel algorithm design is to have an algorithm with low (polylogarithmic) depth and work matching that of the best sequential algorithm for the same problem (*work-efficient*). Being work-efficient is desirable in that the parallel algorithm does not perform asymptotically more operations than the best sequential algorithm for the same problem, and so is efficient even when there is not much parallelism available. Having depth that is polylogarithmic is desirable in that it allows for ample parallelism. Work-efficient and polylogarithmic-depth algorithms have been developed for many fundamental problems in computing. Many of these algorithms, however, are not practical as they involve many sophisticated machinery and have large hidden constant factors in their complexity. On the other hand, many parallel algorithms used in practice are not work-efficient and polylogarithmic-depth. *The goal of my research has been to bridge this gap by developing parallel algorithms that are efficient both in theory and also in practice, as this allows for good performance across all possible inputs, scalability across a wide range of core counts, and graceful scalability to larger data sets.*

Graph Analytics. I have designed theoretically and practically efficient parallel solutions for several problems in graph analytics. *Graph connectivity* is a fundamental problem in computer science with many important applications. There have been many parallel algorithms for connectivity, however the simpler ones require super-linear work, and the linear-work ones are very complicated and not amenable to implementation. I have addressed this gap by developing the *first practical linear-work and polylogarithmic-depth algorithm for graph connectivity* [20]. The algorithm is based on recursively applying a recent parallel algorithm for generating low-diameter graph decompositions. Experimental evaluation shows that the new algorithm is competitive with the fastest existing parallel connectivity implementations (which are not theoretically linear-work and polylogarithmic-depth), and does not have "worst-case" inputs due to its theoretical guarantees.

Triangle counting and enumeration has emerged as a basic tool in the analysis of large-scale networks, fueling the development of solutions that scale to massive graphs. I have designed and implemented simple

[2]The depth of an algorithm is the minimum number of parallel time steps required. The gold standard in parallel algorithm design is to design algorithms that match the sequential work complexity (work-efficient), and have polylogarithmic depth.

and fast multicore parallel algorithms for exact, as well as approximate, triangle counting and other triangle computations that scale to billions of vertices and edges [24]. In addition, I have proven strong asymptotic bounds on the work, depth, and cache complexity of the solutions. I have experimentally shown that on the largest publicly available real-world graphs, the implementations obtain excellent parallel scalability on multicore machines, and are significantly faster than previous parallel solutions for the same problem.

I have developed the fastest parallel algorithm in practice for computing exact maximum flows in graphs [1]. The algorithm is a lock-free synchronous parallel algorithm based on the push-relabel paradigm, and is much simpler than previous algorithms. I have also designed external-memory (disk-based) algorithms for graph connectivity and minimum spanning forest (in the GraphChi graph processing framework) that are simple, have good theoretical guarantees, and are competitive with existing external-memory solutions that are more complicated [9].

Text Analytics. I have also designed scalable parallel solutions for processing texts (strings). I have developed simple shared-memory solutions for suffix trees, Lempel-Ziv factorization, suffix arrays, longest common prefixes, and wavelet trees. These problems have many applications, for example in data compression, information retrieval, and computational biology.

The suffix tree is one of the most important data structures in text processing, as it supports efficient pattern matching and other useful operations on strings. Existing linear-work parallel algorithms are very complicated and not amenable to implementation, while the parallel algorithms used in practice require super-linear work. I have designed the *first linear-work and polylogarithmic-depth parallel suffix tree algorithm that is also practical* [15]. The algorithm is able to build the suffix tree for the 3 GB human genome, one of the largest data sets reported in the literature for suffix tree construction, in under 3 minutes. I have also developed a simple linear-work polylogarithmic-depth parallel solution for computing the Lempel-Ziv factorization of a string, which is a widely-used subroutine in data compression packages [25]. This algorithm is again the first practical linear-work parallel algorithm for the task. In addition, I have designed a memory-efficient parallel suffix array construction algorithm, which is both faster and uses less memory than existing approaches, both in theory and also in practice [10]. The algorithm is work-efficient and has polylogarithmic depth. I have developed several practical parallel algorithms with provable performance guarantees for computing the longest common prefixes of a string [12], which are used in conjunction with suffix arrays to solve a variety of problems in string processing. Finally, I have developed the first polylogarithmic-depth algorithms for constructing wavelet trees, which are used in a variety of compressed data structures [10, 13, 14]. For all of the problems, the algorithms that I have developed are the state-of-the-art in practice for multicore machines. Due to their theoretical guarantees, the algorithms scale gracefully, both as the input size increases and also as the available parallelism increases.

Semisorting. *Semisorting* is the problem of reordering an array such that equal keys are contiguous, but different keys can appear in any order. This has many applications, for example being used in the shuffle step of the MapReduce paradigm, in the `groupBy` operation in database languages, as well as in simulating various parallel machine models with other machine models. I have designed the first practical linear-work and logarithmic depth algorithm for solving this problem [8]. In practice, the algorithm achieves good parallel speedup and outperforms parallel integer sorting and comparison sorting algorithms.

4 Shared-Memory Performance

Priority Updates. Cache coherence protocols have a significant effect on the performance of shared-memory accesses. In general, when updates are performed to a shared location concurrently by many different cores, the memory contention causes performance to worsen as the cache coherence protocol must perform significant work to ensure consistency among different caches. To reduce contention in shared-memory programs, my research develops and advocates the usage of the *priority update* operation [18], which performs an actual update only when the value written has "higher priority" than the existing value, for a large class of applications. I have conducted the first comprehensive study of its performance both experimentally and

theoretically under varying degrees of sharing, showing that it is much more efficient than many commonly-used operations, and comparable in performance to other, less powerful operations. When there is a high degree of sharing the priority update is competitive with reads and test-and-sets (less powerful operations), and *over two orders of magnitude faster* than standard writes and other atomic operations. The priority update operation has many uses in implementing efficient parallel algorithms and concurrent data structures, often in a way that is deterministic, guarantees progress, and avoids serial bottlenecks.

Graph Compression. Much of the running time of graph algorithms is spent on memory accesses, and the parallel scalability of solutions is often limited by the memory subsystem due to cores contending for resources. To alleviate this problem, I have developed Ligra+ by integrating graph compression techniques into the Ligra framework, discussed in Section 2, to reduce memory usage, thus reducing the impact of the scalability bottleneck, and as a result improving parallel performance and scalability [21]. By using efficient decoding techniques, I show that, perhaps surprisingly, Ligra+ benefits from *reduced space usage and improved parallel performance at the same time*. Ligra+ is the *first high-level graph processing system to support in-memory compression*, enabling graphs to be processed using smaller machines, while at the same time improving performance.

5 The Problem Based Benchmark Suite

To measure the programming simplicity, theoretical efficiency, and empirical performance among different solutions for given problems, I have developed a benchmark suite, called the Problem Based Benchmark Suite (PBBS) [19], containing a set of well-known fundamental problems that is representative of a broad class of non-numeric applications arising in computing (e.g. graphs, texts, geometry, machine learning, computational biology, data mining, and graphics). Unlike most existing benchmarks, which are based on specific code, the PBBS benchmarks are defined in terms of the problem specifications—a concrete description of valid inputs and corresponding valid outputs, along with some specific inputs. Any algorithms, programming methodologies, specific programming languages, or machines can be used to solve the problems. The benchmark suite is designed to compare the benefits and shortcomings of different algorithmic and programming approaches, and to serve as a dynamically improving set of educational examples of how to parallelize applications. The PBBS has made it easy to perform comparisons in terms of simplicity, and theoretical/practical performance among various algorithms and programming techniques for the problems studied in my research. The benchmark suite is publicly available at `http://www.cs.cmu.edu/~pbbs`, and has been used by other researchers for benchmarking. The long-term goal of the benchmark suite is to foster algorithmic, programming, and performance research that lead to simple and efficient large-scale parallel solutions for important problems.

6 Research Vision

The continued growth in the volume of data and the emergence of many new computing technologies makes it a very exciting time to do research. The emergence of more powerful multicore machines and nonvolatile memory technologies lead to important research challenges that I plan to help address. As I have done in the past, *I intend to adopt a unified research approach that considers programming techniques, algorithm design, architectural aspects, and performance analysis, as well as the interplay among them*. Certain classes of algorithms are more well-suited to certain types of architectures, and achieving the best performance requires significant attention to both. Furthermore, to make the solutions accessible to others, they should be simple to understand, analyze and use, which will involve both algorithmic and programming considerations. Finally, enabling the solutions to be efficient on a wide variety of shared-memory architectures will involve significant attention to architectural details as well as designing high-level abstractions that hide the architectural details from the programmer. This unified approach will enable me to design solutions that simultaneously satisfy three goals—efficiency, usability, and portability.

Multicore machines with hundreds of cores and tens of terabytes of memory will soon be readily

available.[3] The availability of more powerful multicore machines in the future is promising, as this enables programmers to process larger data sets more efficiently in shared memory. However, taking advantage of these larger multicore machines will involve new challenges, such as designing more efficient cache coherence protocols and raising the importance of locality in algorithms. I am interested in collaborating with other researchers on designing cache coherence protocols that scale to larger numbers of cores. Existing multicore solutions that take non-uniform memory access (NUMA) behavior into account often require significant programming effort. In the future, I plan to design high-level programming frameworks and runtime systems that enable users to write codes that exhibit good locality, without requiring the user to be aware of NUMA behavior.

Emerging nonvolatile memory technologies, such as phase-change memory, offer significantly lower energy and higher density than DRAM, and these technologies are projected to become the dominant memory in the future. For these memories, *writes are significantly more costly than reads*, both in terms of latency and energy. Therefore, my goal is to design parallel algorithms that take this read-write asymmetry into account. I have made some progress in this area by designing algorithms for various fundamental problems that perform asymptotically better than traditional algorithms under a model where reads and writes have asymmetric costs [2, 3, 4]. I am currently designing algorithms for other important problems under this setting, and plan to study the performance of these new algorithms on the memory technologies when they become available in the future.

My research approach is to consider both the theoretical and practical aspects of solutions. I will use theory to inform what makes a good algorithm in practice and also investigate factors that affect practical performance to inform how to develop more accurate theoretical models. Parallel algorithms have been studied from a theoretical perspective since over two decades ago, but due to their complexity many have not been implemented in practice. On the other hand, many parallel solutions used in practice are ad hoc and do not have any provable guarantees. My research focuses on bridging this gap between theory and practice in parallel computing by designing solutions that are both theoretically and practically efficient.

My goal is to be at the forefront of the next generation of algorithms, architectures, and programming tools for large-scale parallel computing. My research involves many areas of computer science, and I am excited to collaborate with other researchers, especially in the areas of systems, theory, and architecture. I am also eager to work closely with researchers from various domains to identify applications that can benefit from large-scale parallel solutions. I am confident that my research results will have high impact, and I am excited to pursue my research agenda and take part in the effort to address the important challenges in parallel computing.

References

[1] (by alphabetical order) Niklas Baumstark, Guy Blelloch and **Julian Shun**. Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm. *European Symposium on Algorithms (ESA)*, pp. 106–117, 2015.

[2] (by alphabetical order) Naama Ben-David, Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Yan Gu, Charles McGuffey and **Julian Shun**. Parallel Algorithms for Asymmetric Read-Write Costs. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 145–156, 2016.

[3] (by alphabetical order) Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Yan Gu and **Julian Shun**. Efficient Algorithms with Asymmetric Read and Write Costs. *European Symposium on Algorithms (ESA)*, pp. 14:1–14:18, 2016.

[4] (by alphabetical order) Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Yan Gu and **Julian Shun**. Sorting with Asymmetric Read and Write Costs. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 1–12, 2015.

[5] (by alphabetical order) Guy Blelloch, Jeremy Fineman, Phillip Gibbons and **Julian Shun**. Internally Deterministic Parallel Algorithms Can Be Fast. *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 181–192, 2012.

[6] (by alphabetical order) Guy Blelloch, Jeremy Fineman and **Julian Shun**. Greedy Sequential Maximal Independent Set and Matching are Parallel on Average. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 308–317, 2012.

[3] For example, Intel has recently announced that it is working on a shared-memory server supporting up to 8 sockets, with each socket containing up to 28 cores and 6 TB of RAM, to be released in 2017.

[7] (by alphabetical order) Guy Blelloch, Yan Gu, **Julian Shun** and Yihan Sun. Parallelism in Randomized Incremental Algorithms *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 467–478, 2016.

[8] Yan Gu, **Julian Shun**, Yihan Sun and Guy Blelloch. A Top-Down Parallel Semisort. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 24–34, 2015.

[9] Aapo Kyrola, **Julian Shun** and Guy Blelloch. Beyond Synchronous: New Techniques for External Memory Graph Algorithms. *Symposium on Experimental Algorithms (SEA)*, pp. 123–137, 2014.

[10] Julian Labeit, **Julian Shun** and Guy Blelloch. Parallel Lightweight Wavelet Tree, Suffix Array and FM-Index Construction. *IEEE Data Compression Conference (DCC)*, pp. 33–42, 2016.

[11] **Julian Shun**. An Evaluation of Parallel Eccentricity Estimation Algorithms on Undirected Real-World Graphs. *ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 1095–1104, 2015.

[12] **Julian Shun**. Fast Parallel Computation of Longest Common Prefixes. *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 387–398, 2014.

[13] **Julian Shun**. Improved Parallel Construction of Wavelet Trees and Rank/Select Structures *To appear in IEEE Data Compression Conference (DCC)*, 2017.

[14] **Julian Shun**. Parallel Wavelet Tree Construction. *IEEE Data Compression Conference (DCC)*, pp. 61–72, 2015.

[15] **Julian Shun** and Guy Blelloch. A Simple Parallel Cartesian Tree Algorithm and its Application to Parallel Suffix Tree Construction. *ACM Transactions on Parallel Computing (TOPC)*, Vol. 1 Issue 1, Article No. 8, 2014.

[16] **Julian Shun** and Guy Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 135–146, 2013.

[17] **Julian Shun** and Guy Blelloch. Phase-concurrent Hash Tables for Determinism. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 96–107, 2014.

[18] **Julian Shun**, Guy Blelloch, Jeremy Fineman and Phillip Gibbons. Reducing Contention Through Priority Updates. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 152–163, 2013.

[19] **Julian Shun**, Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri and Kanat Tangwongsan. Brief Announcement: The Problem Based Benchmark Suite. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 68–70, 2012.

[20] **Julian Shun**, Laxman Dhulipala and Guy Blelloch. A Simple and Practical Linear-Work Parallel Algorithm for Connectivity. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 143–153, 2014.

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

[22] **Julian Shun**, Yan Gu, Guy Blelloch, Jeremy Fineman and Phillip Gibbons. Sequential Random Permutation, List Contraction and Tree Contraction are Highly Parallel. *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 431–448, 2015.

[23] **Julian Shun**, Farbod Roosta-Khorasani, Kimon Fountoulakis and Michael Mahoney. Parallel Local Graph Clustering. *Proceedings of the VLDB Endowment*, 9(12), pp. 1041–1052, 2016.

[24] **Julian Shun** and Kanat Tangwongsan. Multicore Triangle Computations Without Tuning. *IEEE International Conference on Data Engineering (ICDE)*, pp. 149–160, 2015.

[25] (by alphabetical order) **Julian Shun** and Fuyao Zhao. Practical Parallel Lempel-Ziv Factorization. *IEEE Data Compression Conference (DCC)*, pp. 123–132, 2013.