Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable

LAXMAN DHULIPALA, MIT CSAIL
GUY E. BLELLOCH, Carnegie Mellon University
JULIAN SHUN, MIT CSAIL

There has been significant recent interest in parallel graph processing due to the need to quickly analyze the large graphs available today. Many graph codes have been designed for distributed memory or external memory. However, today even the largest publicly-available real-world graph (the Hyperlink Web graph with over 3.5 billion vertices and 128 billion edges) can fit in the memory of a single commodity multicore server. Nevertheless, most experimental work in the literature report results on much smaller graphs, and the ones for the Hyperlink graph use distributed or external memory. Therefore, it is natural to ask whether we can efficiently solve a broad class of graph problems on this graph in memory.

This paper shows that theoretically-efficient parallel graph algorithms can scale to the largest publicly-available graphs using a single machine with a terabyte of RAM, processing them in minutes. We give implementations of theoretically-efficient parallel algorithms for 20 important graph problems. We also present the interfaces, optimizations, and graph processing techniques that we used in our implementations, which were crucial in enabling us to process these large graphs quickly. We show that the running times of our implementations outperform existing state-of-the-art implementations on the largest real-world graphs. For many of the problems that we consider, this is the first time they have been solved on graphs at this scale. We have made the implementations developed in this work publicly-available as the Graph Based Benchmark Suite (GBBS).

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**;

Additional Key Words and Phrases: Parallel graph algorithms, parallel graph processing

ACM Reference format:
Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM Trans. Parallel Comput.* 8, 1, Article 4 (April 2021), 70 pages.
https://doi.org/10.1145/3434393

1 INTRODUCTION

Today, the largest publicly-available graph, the Hyperlink Web graph, consists of over 3.5 billion vertices and 128 billion edges [107]. This graph presents a significant computational challenge

A conference version of this paper appeared in the 30th Symposium on Parallelism in Algorithms and Architectures (2018) [53]; in this version we give significantly more detail on the interface and algorithms.
Authors' addresses: L. Dhulipala and J. Shun, MIT CSAIL, 32 Vassar Street Cambridge, MA 02139; email: {laxman, jshun}@mit.edu; G. E. Blelloch, Computer Science Department, Carnegie Mellon University Pittsburgh, PA 15213; email: guyb@cs.cmu.edu.

2329-4949/2021/04-ART4 $15.00
https://doi.org/10.1145/3434393

for both distributed and shared memory systems. Indeed, very few algorithms have been applied to this graph, and those that have often take hours to run [85, 99, 154], with the fastest times requiring between 1–6 minutes using a supercomputer [145, 146]. In this paper, we show that a wide range of fundamental graph problems can be solved quickly on this graph, often in minutes, on a single commodity shared-memory machine with a terabyte of RAM.[1] For example, our k-core implementation takes under 3.5 minutes on 72 cores, whereas Slota et al. [146] report a running time of about 6 minutes for *approximate* k-core on a supercomputer with over 8000 cores. They also report that they can identify the largest connected component on this graph in 63 seconds, whereas we can identify *all* connected components in just 25 seconds. Another recent result by Stergiou et al. [147] solves connectivity on the Hyperlink 2012 graph in 341 seconds on a 1000 node cluster with 12000 cores and 128TB of RAM. Compared to this result, our implementation is 13.6x faster on a system with 128x less memory and 166x fewer cores. However, we note that they are able to process a significantly larger private graph that we would not be able to fit into our memory footprint. A more complete comparison between our work and existing work, including both distributed and disk-based systems [51, 78, 85, 99, 154], is given in Section 8.

Importantly, all of our implementations have strong theoretical bounds on their work and depth. There are several reasons that algorithms with good theoretical guarantees are desirable. For one, they are robust as even adversarially-chosen inputs will not cause them 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 [52]. We note that most running times that have been reported in the literature on the Hyperlink Web graph use parallel algorithms that are not theoretically-efficient.

In this paper, we present implementations of parallel algorithms with strong theoretical bounds on their work and depth for connectivity, biconnectivity, strongly connected components, low-diameter decomposition, graph spanners, maximal independent set, maximal matching, graph coloring, breadth-first search, single-source shortest paths, widest (bottleneck) path, betweenness centrality, PageRank, spanning forest, minimum spanning forest, k-core decomposition, approximate set cover, approximate densest subgraph, and triangle counting. We describe the programming interfaces, techniques, and optimizations 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. Some of the algorithms we describe are based on previous results from Ligra, Ligra+, and Julienne [52, 136, 141], and other papers on efficient parallel graph algorithms [32, 77, 142]. However, most existing implementations were changed significantly in order to be more memory efficient. Several algorithm implementations for problems like strongly connected components, minimum spanning forest, and biconnectivity are new, and required implementation techniques to scale that we believe are of independent interest. We also had to extend the compressed representation from Ligra+ [141] to ensure that our graph primitives for mapping, filtering, reducing and packing the neighbors of a vertex were theoretically-efficient. We note that using compression techniques is crucial for representing the symmetrized Hyperlink 2012 graph in 1TB of RAM, as storing this graph in an uncompressed format would require over 900GB to store the edges alone, whereas the graph requires 330GB in our

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

Table 1. Running Times (in seconds) of Our Algorithms on the Symmetrized Hyperlink2012 Graph Where (1) is the Single-Thread Time, (72h) is the 72-core Time Using Hyper-threading, and (SU) is the Parallel Speedup

Problem	(1)	(72h)	(SU)	Alg.	Model	Work	Depth
Breadth-First Search (BFS)	576	8.44	68	–	BF	$O(m)$	$O(\text{diam}(G)\log n)$
Integral-Weight SSSP (weighted BFS)	3770	58.1	64	[52]	PW-BF	$O(m)^{\dagger}$	$O(\text{diam}(G)\log n)^{\ddagger}$
General-Weight SSSP (Bellman-Ford)	4010	59.4	67	[49]	PW-BF	$O(\text{diam}(G)m)$	$O(\text{diam}(G)\log n)$
Single-Source Widest Path (Bellman-Ford)	3210	48.4	66	[49]	PW-BF	$O(\text{diam}(G)m)$	$O(\text{diam}(G)\log n)$
Single-Source Betweenness Centrality (BC)	2260	37.1	60	[41]	BF	$O(m)$	$O(\text{diam}(G)\log n)$
$O(k)$-Spanner	2390	36.5	65	[110]	BF	$O(m)$	$O(k\log n)^{\ddagger}$
Low-Diameter Decomposition (LDD)	980	16.6	59	[111]	BF	$O(m)$	$O(\log^2 n)^{\ddagger}$
Connectivity	1640	25.0	65	[140]	BF	$O(m)^{\dagger}$	$O(\log^3 n)^{\ddagger}$
Spanning Forest	2420	35.8	67	[140]	BF	$O(m)^{\dagger}$	$O(\log^3 n)^{\ddagger}$
Biconnectivity	9860	165	59	[148]	FA-BF	$O(m)^{\dagger}$	$O(\text{diam}(G)\log n + \log^3 n)^{\ddagger}$
Strongly Connected Components (SCC)*	8130	185	43	[33]	PW-BF	$O(m\log n)^{\dagger}$	$O(\text{diam}(G)\log n)^{\ddagger}$
Minimum Spanning Forest (MSF)	9520	187	50	[155]	PW-BF	$O(m\log n)$	$O(\log^2 n)^{\ddagger}$
Maximal Independent Set (MIS)	2190	32.2	68	[32]	FA-BF	$O(m)$	$O(\log^2 n)^{\ddagger}$
Maximal Matching (MM)	7150	108	66	[32]	PW-BF	$O(m)^{\dagger}$	$O(\log^2 m)^{\ddagger}$
Graph Coloring	8920	158	56	[77]	FA-BF	$O(m)$	$O(\log n + L\log\Delta)$
Approximate Set Cover	5320	90.4	58	[36]	PW-BF	$O(m)^{\dagger}$	$O(\log^3 n)^{\ddagger}$
k-core	8515	184	46	[52]	FA-BF	$O(m)^{\dagger}$	$O(\rho\log n)^{\ddagger}$
Approximate Densest Subgraph	3780	51.4	73	[18]	FA-BF	$O(m)$	$O(\log^2 n)$
Triangle Counting (TC)	—	1168	—	[142]	BF	$O(m^{3/2})$	$O(\log n)$
PageRank Iteration	973	13.1	74	[42]	FA-BF	$O(n+m)$	$O(\log n)$

Theoretical bounds for the algorithms and the variant of the binary-forking model used are shown in the last three columns. Section 3.3 provides more details about the binary-forking model. We mark times that did not finish in 5 hours with —. *SCC was run on the directed version of the graph. † denotes that a bound holds in expectation, and ‡ denotes that a bound holds with high probability. We say that an algorithm has $O(f(n))$ cost ***with high probability (whp)*** if it has $O(k \cdot f(n))$ cost with probability at least $1 - 1/n^k$. We assume $m = \Omega(n)$.

compressed format (less than 1.5 bytes per edge). We show the running times of our algorithms on the Hyperlink 2012 graph as well as their work and depth bounds in Table 1. To make it easy to build upon or compare to our work in the future, we describe the Graph Based Benchmark Suite (GBBS), a benchmark suite containing our problems with clear I/O specifications, which we have made publicly-available.[2]

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

2 RELATED WORK

Parallel Graph Algorithms. Parallel graph algorithms have received significant attention since the start of parallel computing, and many elegant algorithms with good theoretical bounds have been developed over the decades (e.g., [5, 25, 45, 63, 84, 89, 98, 109, 111, 112, 121, 125, 135, 148]). 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,

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

many problems that are of practical importance do admit work-efficient algorithms [88]. 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* [88], 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 [33, 131]. Other problems, like k-core are P-complete [7], which rules out polylogarithmic-depth algorithms for them unless P = NC [73]. However, even k-core admits an algorithm with strong theoretical guarantees on its work that is efficient in practice [52].

Parallel Graph Processing Frameworks. Motivated by the need to process very large graphs, there have been many graph processing frameworks developed in the literature (e.g., [71, 97, 101, 115, 136] among many others). We refer the reader to [105, 152] for surveys of existing frameworks. Several recent graph processing systems evaluate the scalability of their implementations by solving problems on massive graphs [52, 85, 99, 145, 147, 154]. All of these systems report running times either on the Hyperlink 2012 graph or Hyperlink 2014 graphs, two web crawls released by the WebDataCommons that are the largest and second largest publicly-available graphs respectively. We describe these recent systems and give a detailed comparison of how our implementations compare to these existing solutions in Section 8.

Benchmarking Parallel Graph Algorithms. There are a surprising number of existing benchmarks of parallel graph algorithms. SSCA [15] specifies four graph kernels, which include generating graphs in adjacency list format, subgraph extraction, and graph clustering. The Problem Based Benchmark Suite (PBBS) [139] is a general benchmark of parallel algorithms that includes 6 problems on graphs: BFS, spanning forest, minimum spanning forest, maximal independent set, maximal matching, and graph separators. The PBBS benchmarks are *problem-based* in that they are defined only in terms of the input and output without any specification of the algorithm used to solve the problem. We follow the style of PBBS in this paper of defining the input and output requirements for each problem. The Graph Algorithm Platform (GAP) Benchmark Suite [22] specifies 6 kernels: BFS, SSSP, PageRank, connectivity, betweenness centrality, and triangle counting.

Several recent benchmarks characterize the architectural properties of parallel graph algorithms. GraphBIG [113] describes 12 applications, including several problems that we consider, like k-core and graph coloring (using the Jones-Plassmann algorithm), but also problems like depth-first search, which are difficult to parallelize, as well as dynamic graph operations. CRONO [4] implements 10 graph algorithms, including all-pairs shortest paths, exact betweenness centrality, traveling salesman, and depth-first search. LDBC [81] is an industry-driven benchmark that selects 6 algorithms that are considered representative of graph processing including BFS, and several algorithms based on label propagation.

Unfortunately, all of the existing graph algorithm benchmarks we are aware of restrict their evaluation to small graphs, often on the order of tens or hundreds of millions of edges, with the largest graphs in the benchmarks having about two billion edges. As real-world graphs are frequently sev-

eral orders of magnitude larger than this, evaluation on such small graphs makes it hard to judge whether the algorithms or results from a benchmark scale to terabyte-scale graphs. This paper provides a problem-based benchmark in the style of PBBS for fundamental graph problems, and evaluates theoretically-efficient parallel algorithms for these problems on the largest real-world graphs, which contain hundreds of billions of edges.

3 PRELIMINARIES

3.1 Graph 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 $n - 1$. We call these unique integer identifiers for vertices *vertex IDs*. For undirected graphs we use $N(v)$ to denote the neighbors of vertex v and $d(v)$ to denote its degree. For directed graphs, we use $N^-(v)$ and $N^+(v)$ to denote the in and out-neighbors of a vertex v, and $d^-(v)$ and $d^+(v)$ to denote its in and out-degree, respectively. We use $\text{dist}_G(u, v)$ to refer to the shortest path distance between u and v in G. We use $\text{diam}(G)$ 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. Given an undirected graph $G = (V, E)$ the density of a set $S \subseteq V$, or $\mathcal{D}(S)$, is equal to $\frac{|E(S)|}{|S|}$ where $E(S)$ are the edges in the induced subgraph on S. Δ is used to denote the maximum degree of the graph. 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.

3.2 Atomic Primitives

We use three common atomic primitives in our algorithms and implementations: **testAndSet(TS)**, **fetchAndAdd(FA)**, and **priorityWrite(PW)**. A testAndSet($\&x$) checks if x is false, and if so atomically sets it to true and returns true; otherwise it returns false. A fetchAndAdd ($\&x$) atomically returns the current value of x and then increments x. A priorityWrite($\&x, v, p$) atomically compares v with the current value of x using the priority function p, and if v has higher priority than the value of x according to p it sets x to v and returns true; otherwise it returns false.

3.3 Parallel Model and Cost

In the analysis of algorithms in this paper we use a work-depth model which is closely related to the PRAM but better models current machines and programming paradigms that are asynchronous and allow dynamic forking. We can simulate this model on the CRCW PRAM equipped with the same operations with an additional $O(\log^* n)$ factor in the depth *whp* due to load-balancing [31]. Furthermore, a PRAM algorithm using P processors and T time can be simulated in our model with PT work and T depth.

The ***Binary-Forking Model*** [28, 31] consists of a set of threads that share an unbounded memory. Each thread is basically equivalent to a Random Access Machine—it works on a program stored in memory, has a constant number of registers, and has standard RAM instructions (including an `end` to finish the computation). The binary-forking model extends the RAM with a `fork` instruction that forks 2 new child threads. Each child thread receives a unique integer in the range $[1, 2]$ in its first register and otherwise has the identical state as the parent, which has a 0 in that register. They all start by running the next instruction. When a thread performs a `fork`, it is suspended un-

til both of its children terminate (execute an end instruction). A computation starts with a single root thread and finishes when that root thread ends. Processes can perform reads and writes to the shared memory, as well as the testAndSet instruction. This model supports what is often referred to as nested parallelism. If the root thread never does a fork, it is a standard sequential program.

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

Model Variants. We augment the binary-forking model described above with two atomic instructions that are used by our algorithms: fetchAndAdd and priorityWrite and discuss the model with these instruction as the FA-BF, and PW-BF variants of the binary-forking model, respectively. We abbreviate the basic binary-forking model with only the testAndSet instruction as the BF model. Note that the basic binary-forking model includes a testAndSet, as this instruction is necessary to implement joining tasks in a parallel schedulers (see for example [9, 38]), and since all modern multicore architectures include the testAndSet instruction.

3.4 Parallel Primitives

The following parallel procedures are used throughout the paper. A ***monoid*** over a type E is an object consisting of an associative function $\oplus : \mathsf{E} \times \mathsf{E} \to \mathsf{E}$, and an identity element $\bot : \mathsf{E}$. A monoid is specified as a pair, $(\bot, \oplus)$. ***Scan*** takes as input an array A of length n, and a monoid $(\bot, \oplus)$ and returns the array $(\bot, \bot \oplus A[0], \bot \oplus A[0] \oplus A[1], \ldots, \bot \oplus_{i=0}^{n-2} A[i])$ as well as the overall sum, $\bot \oplus_{i=0}^{n-1} A[i]$. Scan can be done in $O(n)$ work and $O(\log n)$ depth (assuming $\oplus$ takes $O(1)$ work) [84]. ***Reduce*** takes an array A and a monoid $(\bot, \oplus)$ and returns the sum of the elements in A with respect to the monoid, $\bot \oplus_{i=0}^{n-1} A[i]$. ***Filter*** takes an array A and a predicate f and returns a new array containing $a \in A$ for which $f(a)$ is true, in the same order as in A. Reduce and filter can both be done in $O(n)$ work and $O(\log n)$ depth (assuming $\oplus$ and f take $O(1)$ work). Finally, the ***PointerJump*** primitive takes an array P of parent pointers which represent a directed rooted forest (i.e., $P[v]$ is the parent of vertex v) and returns an array R where $R[v]$ is the root of the directed tree containing v. This primitive can be implemented in $O(n)$ work, and $O(\log n)$ depth *whp* in the BF model [31].

4 INTERFACE

In this section we describe the high-level graph processing interface used by our algorithm implementations in GBBS, and explain how the interface is integrated into our overall system architecture. The interface is written in C++ and extends the Ligra, Ligra+, and Julienne frameworks [52, 136, 141] with additional functional primitives over graphs and vertices that are parallel by default. In what follows, we provide descriptions of these functional primitives, as well as the parallel cost bounds obtained by our implementations.

System Overview. The GBBS library, which underlies our benchmark algorithm implementations is built as a number of layers, which we illustrate in Figure 1.[3] We use a shared-memory approach to parallel graph processing in which the entire graph is stored in the main memory of a single multicore machine. Our codes exploit nested parallelism using scheduler-agnostic parallel primitives, such as *fork-join* and parallel-for loops. Thus, they can easily be compiled to use different

[3] A brief version of this interface was presented by the authors and their collaborators in [59].

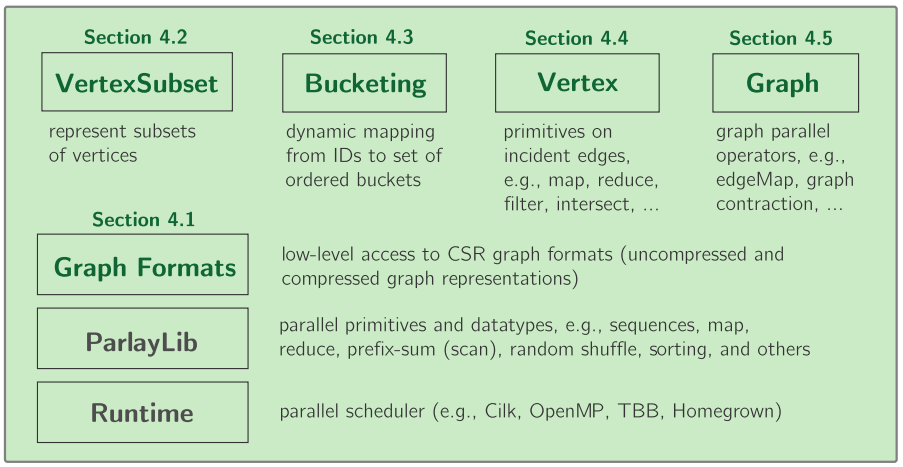

Fig. 1. System architecture of GBBS. The core interfaces are the vertexSubset (Section 4.2), *bucketing* (Section 4.3), *vertex* (Section 4.4), and *graph* interfaces (Section 4.5). These interfaces utilize parallel primitives and routines from ParlayLib [27]. Parallelism is implemented using a parallel runtime system—Cilk, OpenMP, TBB, or a homegrown scheduler based on the Arora-Blumofe-Plaxton deque [10] that we implemented ourselves—and can be swapped using a command line argument. The vertex and graph interfaces use a compression library that mediates access to the underlying graph, which can either be compressed or uncompressed (see Section 4.1).

Table 2. Type Names Used in the Interface, and Their Definitions

Type Name	Definition	Found In
unit	An empty tuple indicating a trivial value (similar to void in languages like C)	—
E option	Either a value of type E (Some(e : E)) or no value (None)	—
E monoid	A pair of an identity element, $\perp$: E, and an associative function, $\oplus$: E × E → E	Section 3.4
E sequence	A parallel sequence containing values of type E	Section 3.4
A → B	A function with arguments of type A with results of type B	—
vtxid	A vertex ID (unique integer identifiers for vertices)	Section 3.1
vertexSubset	Data type representing a subset of vertex IDs	Section 4.2
E vertexSubset	An augmented vertexSubset (each vertex ID has an associated value of type E)	Section 4.2
vset	Abbreviation for a vertexSubset	Section 4.2
identifier	A unique integer representing a bucketed object	Section 4.3
bktid	A unique integer for each bucket	Section 4.3
bktorder	The order to traverse buckets in (increasing or decreasing)	Section 4.3
bktdest	Type representing which bucket an identifier is moving to	Section 4.3
edge	A tuple representing an edge in the graph	—
nghlist	Data type representing the neighbors of a vertex	Section 4.4
graph	Data type representing a collection of vertices and edges	Section 4.5

The third column provides a reference to where the type is defined in the text (if applicable).

parallel runtimes such as Cilk, OpenMP, TBB, and also a custom work-stealing scheduler implemented by the authors [27]. Theoretically, our algorithms are implemented and analyzed in the binary-forking model [31] (see Section 3.3). Our interface makes use of several new types which are defined in Table 2. We also define these types when they are first used in the text.

4.1 Graph Representations

We first cover the different types of graph representations used in the library. The basic graph representation stores graphs in the compressed sparse row format (described below). To efficiently store very large graphs, we also utilize a compressed graph format which encodes sorted neighbor lists using difference encoding, which we describe below. Finally, our library also supports arbitrary edge weights, and provides functionality for compressing integer edge weights. As described in Section 3, in this paper, we deal with graphs where vertices are identified by unique integers between 0 to $n-1$. We use the vtxid type to refer to these integer vertex IDs.

Compressed Graphs. Graphs in GBBS are stored in the ***compressed sparse row (CSR)*** format. CSR stores two arrays, I and A, where the vertices are in the range $[0, n-1]$ and incident edges of a vertex v are stored in $\{A[I[v]], \ldots, A[I[v+1]-1]\}$ (with a special case for vertex $n-1$). The *uncompressed* format in GBBS is equivalent to the CSR format. The format assumes that the edges incident to a vertex are sorted by the neighboring vertex ID. GBBS also supports the *compressed* graph formats from the Ligra+ framework [141]. Specifically, we provide support for graphs where neighbor lists are encoded using byte codes and a parallel generalization of byte codes, which we describe next.

In the ***byte format***, we store a vertex's neighbor list by difference encoding consecutive vertex IDs, with the first difference encoded with respect to the source vertex ID. Decoding is done by sequentially uncompressing each difference, and summing the differences into a running sum which gives the vertex ID of the next neighbor. As this process is sequential, graph algorithms using the byte format that map over the neighbors of a vertex will have poor depth bounds.

We enable parallelism using the ***parallel-byte format*** from Ligra+ [141]. This format breaks the neighbors of a high-degree vertex into blocks, where each block contains a constant number of neighbors. Each block is difference encoded with respect to the source, and the format stores the blocks in a neighbor list in sorted order. As each block can have a different size, it also stores offsets that point to the start of each block. Using the parallel-byte format, the neighbor vertex IDs of a high-degree vertex can then be decoded in parallel over the blocks. We refer the reader to Ligra+ [141] for a detailed discussion of the idea. We provide many parallel primitives for processing neighbor lists compressed in the parallel-byte format in Section 7.4.

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

Both the byte and parallel-byte schemes above provide support for weighted graphs. If the graph weight type is E, the encoder simply interleaves the weighted elements of type E with the differences generated by the byte or parallel byte code. Additionally, GBBS supports compressing integer weights using variable-length coding, similar to Ligra+ [141].

4.2 VertexSubset Interface

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

Primitives. We use four primitives defined on vertexSubset, which we illustrate in Figure 2. VERTEXMAP takes a vertexSubset and applies a user-defined function f over each vertex. This

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

Fig. 2. The core primitives in the vertexSubset interface used by GBBS, including the type definition of each primitive and the cost bounds. We use vset as an abbreviation for vertexSubset in the figure. A vertexSubset is a representation of a set of vertex IDs, which are unique integer identifiers for vertices. If the input vertexSubset is augmented, the user-defined functions supplied to vertexMap and vertexFilter take a pair of the vertex ID and augmented value as input, and the addToSubset primitive takes a sequence of $vertexID$ and augmented value pairs.

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

4.3 Bucketing Interface

We use the bucketing interface and data structure from Julienne [52], which represents a dynamic mapping from identifiers to buckets. Each bucket is represented as a vertexSubset, and the interface allows vertices to dynamically be moved through different buckets as priorities change. The interface enables priority-based graph algorithms, including integral-weight shortest paths, k-core decomposition, and others [52]. Algorithms using the interface iteratively extract the highest priority bucket, potentially update incident vertex priorities, and repeat until all buckets are empty.

The interface is shown in Figure 3. The interface uses several types that we now define. An **identifier** is a unique integer representing a bucketed object. An identifier is mapped to a **bktid**, a unique integer for each bucket. The order that buckets are traversed in is given by the **bktorder** type. **bktdest** is an opaque type representing where an identifier is moving inside of the structure. Once the structure is created, an object of type buckets is returned to the user.

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

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

Fig. 3. The bucketing interface used by GBBS, including the type definition of each primitive and the cost bounds. The bucketing structure represents a dynamic mapping between a set of identifiers to a set of buckets. The total number of identifiers is denoted by n. † denotes that a bound holds in expectation, and ‡ denotes that a bound holds *whp*. We define the semantics of each operation in the text below.

itive updates the bktids for multiple identifiers by supplying the bucket structure and a sequence of identifier and bktdest pairs.

The costs for using the bucket structure can be summarized by the following theorem from [52]:

Theorem 4.1. *When there are n identifiers, T total buckets, K calls to* updateBuckets*, each of which updates a set S_i of identifiers, and L calls to* nextBucket*, parallel bucketing takes $O(n + T + \sum_{i=0}^{K} |S_i|)$ expected work and $O((K + L) \log n)$ depth whp.*

We refer to the Julienne paper [52] for more details about the bucketing interface and its implementation. We note that the implementation is optimized for the case where only a small number of buckets are processed, which is typically the case in practice.

4.4 Vertex Interface

GBBS provides vertex data types for both symmetric and asymmetric vertices, used for undirected and directed graphs, respectively. The vertex data type interface (see Figure 4) provides functional primitives over vertex neighborhoods, such as map, reduce, scan, count (a special case of reduce over the $(0, +)$ monoid where the map function is a boolean function), as well as primitives to extract a subset of the neighborhood satisfying a predicate (filter) and an internal primitive to mutate the vertex neighborhood and delete edges that do not satisfy a given predicate (pack). Since pack mutates the underlying vertex neighborhood in the graph, which requires updating the number of edges remaining in the graph, we do not expose it to the user, and instead provide APIs to pack a graph in-place using the packGraph and (ngh/src)Pack primitives described later. The interface also provides a sequential iterator that takes as input a function f from edges to booleans, and applies f to each successive neighbor, terminating once f returns false. Note that for directed graphs, each of the neighborhood operators has two versions, one for the in-neighbors and one for the out-neighbors of the vertex.

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

The cost bounds for the interface are derived by applying known bounds for efficient sequence primitives (see Section 3). We provide additional details about the implementations of our compressed implementations in Section 7.4.

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

Fig. 4. The core vertex interface used by GBBS, including the type definition of each primitive and the cost bounds for our implementations on uncompressed graphs. Note that for directed graphs, each of the neighborhood operators has two versions, one for the in-neighbors and one for the out-neighbors of the vertex. The cost bounds for the primitives on compressed graphs are identical assuming the compression block size is $O(\log n)$ (note that for compressed graphs, i-th has work and depth proportional to the compression block size of the graph in the worst case). The cost bounds shown here assume that the user-defined functions supplied to MAP, REDUCE, SCAN, COUNT, FILTER, PACK, and ITERATE all cost $O(1)$ work to evaluate. d_{it} is the number of times the function supplied to ITERATE returns true. nghlist is an abstract type for the neighbors of a vertex, and is used by the vertex-vertex operators. The edge type is a triple (u, v, w_{uv}) where the first two entries are the ids of the endpoints, and the last entry is the weight of the edge. l and h are the degrees of the smaller and larger degree vertices supplied to a vertex-vertex operator, respectively.

4.5 Graph Interface

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

4.5.1 Graph Operators. The graph operators, their types, and the cost bounds provided by our implementation are shown in the top half of Figure 5. The interface provides primitives for querying the number of vertices and edges in the graph (NUMVERTICES and NUMEDGES), and for fetching the vertex object for the i-th vertex (GETVERTEX).

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

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

Fig. 5. The core graph interface used by GBBS, including the type definition of each primitive and the cost bounds for our implementations on uncompressed graphs. vset is an abbreviation for vertexSubset when providing a type definition. Note that for directed graphs, the interface provides two versions of each vertexSubset operator, one for the in-neighbors and one for the out-neighbors of the vertex. The edge type is a triple (u, v, w_{uv}) where the first two entries are the ids of the endpoints, and the last entry is the weight of the edge. The vertexSubset operators can take both unaugmented and augmented vertexSubsets as input, but ignore the augmented values in the input. U is the vertexSubset supplied as input to a vertexSubset operator. For the src-based primitives, $U' \subseteq U$ is the set of vertices that are matched by the condition function (see the text below). The cost bounds for the primitives on compressed graphs are identical assuming the compression block size is $O(\log n)$. The cost bounds shown here assume that the user-defined functions supplied to the vertexSubset operators all cost $O(1)$ work to evaluate. † denotes that a bound holds in expectation, and ‡ denotes that a bound holds *whp*.

packGraph. The interface also provides a primitive over edges called packGraph which operates similarly to filterGraph, but works in-place and mutates the underlying graph. packGraph takes as input a graph $G(V, E)$, and a boolean function P over the edges specifying edges to preserve. packGraph mutates the input graph to remove all edges that do not satisfy the predicate. This primitive is used by the biconnectivity (Algorithm 9), strongly connected components (Algorithm 11), maximal matching (Algorithm 13), and minimum spanning forest (Algorithm 10) algorithms studied in this paper.

extractEdges. The extractEdges primitive takes as input a graph $G(V, E)$, and a boolean function P over edges which specifies edges to extract, and returns an array containing all edges where $P(u, v, w_{uv}) = \text{true}$. This primitive is useful in algorithms studied in this paper such as

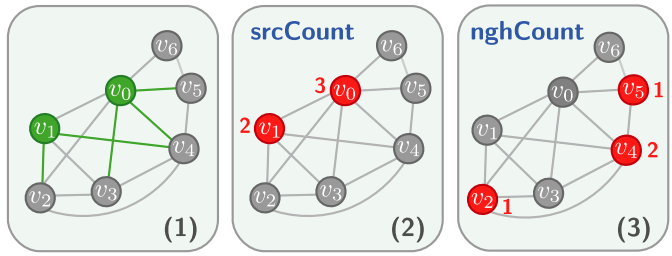

Fig. 6. Illustration of srcCount and nghCount primitives. The input is illustrated in Panel (1), and consists of a graph and a vertexSubset, with vertices in the vertexSubset illustrated in green. The green edges are edges for which the user-defined predicate, P, returns true. Panel (2) and Panel (3) show the results of applying srcCount and nghCount, respectively. In Panel (2), the cond function C returns true for both vertices in the input vertexSubset. In Panel (3), the condition function C only returns true for v_2, v_4, and v_5, and false for v_0, v_1, v_3, and v_6. The output is an augmented int vertexSubset, illustrated in red, where each source (neighbor) vertex v s.t. $C(v)$ = true has an augmented value containing the number of incident edges where P returns true.

maximal matching (Algorithm 13) and minimum spanning forest (Algorithm 10) where it is used to extract subsets of edges from a CSR representation of the graph, which are then processed using an algorithm operating over edgelists (edges tuples stored in an array).

contractGraph. Lastly, the contractGraph primitive takes a graph and an integer cluster labeling L, i.e., a mapping from vertices to cluster ids, and returns the graph $G' = (V', E')$ where $E' = \{(L(u), L(v)) \mid (u, v) \in E\}$, with any duplicate edges or self-loops removed. V' is V with all vertices with no incident edges in E' removed. This primitive is used by the connectivity (Algorithm 7) and spanning forest (Algorithm 8) algorithms studied in this paper. The primitive can naturally be generalized to weighted graphs by specifying how to reweight parallel edges (e.g., by averaging, or taking a minimum or maximum), although this generalization is not necessary for the algorithms studied in this paper.

Implementations and Cost Bounds. filterGraph, packGraph, and extractEdges are implemented by invoking filter and pack on each vertex in the graph in parallel. The overall work and depth comes from the fact that every edge is processed once by each endpoint, and since all vertices are filtered (packed) in parallel. contractGraph can be implemented in $O(n + m)$ expected work and $O(\log n)$ depth *whp* in the BF model using semisorting [31, 74]. In practice, contractGraph is implemented using parallel hashing [137], and we refer the reader to [140] for the implementation details.

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

edgeMap. The edgeMap primitive takes as input a graph $G(V, E)$, a vertexSubset U, and two boolean functions F and C. edgeMap applies F to $(u, v) \in E$ such that $u \in U$ and $C(v)$ = 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 $F(u, v)$ = true. Our interface defines the edgeMap primitive identically to Ligra. This primitive is used in many of the algorithms studied in this paper.

edgeMapData. The edgeMapData primitive works similarly to edgeMap, but returns an augmented vertexSubset. Like edgeMap, it takes as input a graph $G(V, E)$, a vertexSubset U, a

function F returning a value of type R option, and a boolean function C. edgeMapData applies F to $(u, v) \in E$ such that $u \in U$ and $C(v) = \text{true}$ (call this subset of edges E_a), and returns a R vertexSubsetU' where $(u, r) \in U'$ (r is the augmented value associated with u) if and only if $(u, v) \in E_a$ and $F(u, v) = \text{Some}(r)$. The primitive is only used in the weighted breadth-first search algorithm in this paper, where the augmented value is used to store the distance to a vertex at the start of a computation round (Algorithm 2).

srcReduce and srcCount. The ***srcReduce*** primitive takes as input a graph $G(V, E)$ and a vertexSubset U, a map function M over edges returning values of type R, a boolean function C, and a monoid A over values of type R, and returns a R vertexSubset. srcReduce applies M to each $(u, v) \in E$ s.t. $u \in U$ and $C(u) = \text{true}$ (let M_u be the set of values of type R from applying M to edges incident to u), and returns a R vertexSubset U' containing (u, r) where r is the result of reducing all values in M_u using the monoid A.

The ***srcCount*** primitive is a specialization of srcReduce, where R = int, the monoid A is $(0, +)$, and the map function is specialized to a boolean (predicate) function P over edges. This primitive is useful for building a vertexSubset where the augmented value for each vertex is the number of incident edges satisfying some condition. srcCount is used in our parallel approximate set cover algorithm (Algorithm 15).

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

nghReduce and nghCount. The nghReduce primitive is defined similarly to srcReduce above, but aggregates the results for *neighbors* of the input vertexSubset. It takes as input a graph $G(V, E)$, a vertexSubset U, a map function M over edges returning values of type R, a boolean function C, a monoid A over values of type R, and lastly an update function T from values of type R to O option. It returns a O vertexSubset. This function performs the following logic: M is applied to each edge (u, v) where $u \in U$ and $C(v) = \text{true}$ in parallel (let the resulting values of type R be M_v). Next, the mapped values for each such v are reduced in parallel using the monoid A to obtain a single value, R_v. Finally, T is called on the pair (v, R_v) and the vertex and augmented value pair (v, o) is emitted to the output vertexSubset if and only if T returns $\text{Some}(o)$. nghReduce is used in our PageRank algorithm (Algorithm 19).

The ***nghCount*** primitive is a specialization of nghReduce, where R = int, the monoid A is $(0, +)$, and the map function is specialized to a boolean (predicate) function P over edges. nghCount is used in our k-core (Algorithm 16) and approximate densest subgraph (Algorithm 17) algorithms.

Implementations and Cost Bounds. Our implementation of edgeMap in this paper is based on the edgeMapBlocked primitive introduced in Section 7.2. The same primitive is used to implement edgeMapData.

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

The NGH- primitives require are somewhat trickier to implement compared to the SRC- primitives, since these primitives require performing non-local reductions at the neighboring endpoints of edges. Both NGHREDUCE and NGHCOUNT can be implemented by first writing out all neighbors of the input vertexSubset satisfying C to an array, A (along with their augmented values). A has size at most $O(\sum_{u \in U} d(u))$. The next step applies a work-efficient semisort (e.g., [74]) to store all pairs of neighbor and value keyed by the same neighbor contiguously. The final step is to apply a prefix sum over the array, combining values keyed by the same neighbor using the reduction operation defined by the monoid, and to use a prefix sum and map to build the output vertexSubset, augmented with the final value in the array for each neighbor. The overall work is proportional to semisorting and applying prefix-sums on arrays of $|A|$, which is $O(\sum_{u \in U} d(u))$ work in expectation, and the depth is $O(\log n)$ *whp* [31, 74]. In practice, our implementations use the work-efficient histogram technique described in Section 7.1 for both NGHREDUCE and NGHCOUNT.

Optimizations. We observe that for NGH- operators there is a potential to achieve speedups by applying the direction-optimization technique proposed by Beamer for the BFS problem [21] and applied to other problems by Shun and Blelloch [136]. Recall that this technique maps over all vertices $v \in V$, and for those where $C(v) = \text{true}$, scans over the in-edges (v, u, w_{vu}) applying F to edges where u is in the input vertexSubset until $C(v)$ is no longer true. We can apply the same technique for NGHREDUCE and NGHCOUNT by performing a reduction over the in-neighbors of all vertices satisfying $C(v)$. This optimization can be applied without an increase in the theoretical cost of the algorithm whenever the number edges incident to the input vertexSubset is a constant fraction of m. The advantage is that the direction-optimized version runs in $O(n)$ space and performs inexpensive reads over the in-neighbors, whereas the more costly semisort or histogram based approach runs in $O(\sum_{u \in U} d(u))$ space and requires performing multiple writes per incident edge.

5 BENCHMARK

In this section we describe I/O specifications of our benchmark. We discuss related work and present the theoretically-efficient algorithm implemented for each problem in Section 6. We mark implementations based on prior work with a †, although in many of these cases, the implementations were still significantly modified to improve scalability on large compressed graphs.

5.1 Shortest Path Problems

Breadth-First Search (BFS)†

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

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

Integral-Weight SSSP (weighted BFS)†

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

General-Weight SSSP (Bellman-Ford)[†]

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

Single-Source Betweenness Centrality (BC)

Input: $G = (V, E)$, an unweighted graph, $src \in V$.
Output: D, a mapping from each vertex v to the dependency value of this vertex with respect to src. Section 6.1 provides the definition of dependency values. $D[v] = \infty$ if v is unreachable.

Widest Path (Bottleneck Path)

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

$O(k)$-Spanner

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

5.2 Connectivity Problems

Low-Diameter Decomposition[†]

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

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

Connectivity[†]

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

Spanning Forest†

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

Biconnectivity

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

Minimum Spanning Forest

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

Strongly Connected Components

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

5.3 Covering Problems

Maximal Independent Set†

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

Maximal Matching†

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

Graph Coloring†

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

Approximate Set Cover†

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

5.4 Substructure Problems

k-core†

Input: $G = (V, E)$, an undirected graph.
Output: D, a mapping from each vertex to its coreness value. Section 6.4 provides the definition of k-cores and coreness values.

Approximate Densest Subgraph

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

Triangle Counting†

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

5.5 Eigenvector Problems

PageRank†

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

6 ALGORITHMS

In this section, we give self-contained descriptions of all of the theoretically efficient algorithms implemented in our benchmark and discuss related work. We cite the original papers that our algorithms are based on in Table 1. We assume $m = \Omega(n)$ when stating cost bounds in this section.

Pseudocode Conventions. The pseudocode for many of the algorithms make use of our graph processing interface, as well as the atomic primitives testAndSet, fetchAndAdd, and priorityWrite. Our graph processing interface is defined in Section 4 and the atomic primitives are defined in Section 3. We use _ as a wildcard to bind values that are not used. We use anonymous functions in the pseudocode for consciseness, and adopt a syntax similar to how anonymous functions are defined in the ML language. An anonymous function is introduced using the **fn** keyword. For example,

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

is an anonymous function taking a triple representing an edge, and returning the *Rank* of the vertex v. We drop type annotations when the argument types are clear from context. The option type, E option, provides a distinction between some value of type E (**Some(e)**) and no value (**None**). Types used by our algorithms are also summarized in Table 2. We use the array initializer notation $A[0, \ldots, e) = value$ to denote an array consisting of e elements all initialized to *value* in parallel. We use standard functional sequence primitives, such as map and filter on arrays. Assuming that the user-defined map and filter functions cost $O(1)$ work to apply, these primitives cost $O(n)$ work and $O(\log n)$ depth on a sequence of length n. We use the syntax $\forall i \in [s, e)$ as shorthand for a parallel loop over the indices $[s, \ldots, e)$. For example, $\forall i \in [0, e), A[i] = i \cdot A[i]$ updates the i-th value of $A[i]$ to $i \cdot A[i]$ in parallel for $0 \leq i < e$.

6.1 Shortest Path Problems

Although work-efficient polylogarithmic-depth algorithms for single-source shortest paths (SSSP) type problems are not known due to the transitive-closure bottleneck [88], work-efficient algorithms that run in depth proportional to the diameter of the graph are known for the special cases considered in our benchmark. Several work-efficient parallel breadth-first search algorithms are known [16, 30, 94]. On weighted graphs with integral edge weights, SSSP can be solved in $O(m)$ work and $O(\text{diam}(G) \log n)$ depth [52]. Parallel algorithms also exist for weighted graphs with positive edge weights [108, 109]. SSSP on graphs with negative integer edge weights can be solved using Bellman-Ford [49], where the number of iterations depends on the diameter of the graph. Betweenness centrality from a single source can be computed using two breadth-first searches [41, 136]. We note that very recently, a breakthrough result of Andoni et al. and Li [8, 95] show that computing $(1 + \epsilon)$-approximate SSSP can be done nearly work-efficiently (up to poly-logarithmic factors) in poly-logarithmic depth. An interesting question for future work is to understand whether ideas from this line of work can result in practical parallel approximation algorithms for SSSP.

In this paper, we present implementations of five SSSP problems that are based on graph search. We also include an algorithm to construct an $O(k)$-spanner which is based on computing low-diameter decompositions. Our implementations of BFS and Bellman-Ford are based on the implementations in Ligra [136]. Our betweenness centrality implementation applies the same broad implementation strategy as the Ligra implementation, but differs significantly in the details, which we describe below. Our wBFS implementation is based on our earlier work on Julienne [52].

Breadth-First Search (BFS)

The BFS problem is to compute a mapping representing distances between the source vertex, *src* and every other vertex. The distances to unreachable vertices should be set to ∞. Algorithm 1 shows pseudocode for our BFS implementation. The BFS procedure takes as input a graph and a source vertex *src*, and calls GeneralizedBFS with an initial vertexSubset containing just the source vertex, *src*. The GeneralizedBFS procedure is used later in our Bellman-Ford algorithm (Algorithm 3).

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

ALGORITHM 1: Breadth-First Search

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

be emitted in the output vertexSubset (Line 7), and otherwise returns false (Line 8). Finally, at the end of a round the algorithm increments the value of the current distance on Line 17.

Both the GeneralizedBFS and BFS algorithms run in $O(m)$ work and $O(\text{diam}(G) \log n)$ depth on the BF model. We note that emitting a shortest-path tree from a subset of vertices instead of distances can be done using nearly identical code, with the only differences being that (i) the algorithm will store a *Parents* array instead of a *Distances* array, and (ii) the Update function will set the parent of a vertex d to s upon a successful testAndSet. The main change we made to this algorithm compared to the Ligra implementation was to improve the cache-efficiency of the edgeMap implementation using edgeMapBlocked, the block-based version of edgeMap described in Section 7.

Integral-Weight SSSP (wBFS)

The integral-weight SSSP problem is to compute the shortest path distances between a source vertex and all other vertices in a graph with positive integer edge weights. Our implementation implements the weighted breadth-first search (wBFS) algorithm, a version of Dijkstra's algorithm that is well suited for low-diameter graphs with small positive integer edge weights. Our implementation uses the bucketing interface from Julienne described in Section 4. The idea of our algorithm is to maintain a bucket for each possible distance, and to process them in order of increasing distance. Each bucket is like a frontier in BFS, but unlike BFS, when we process a neighbor u of a vertex v in the current bucket i, we place u in bucket $i + w_{uv}$.

Algorithm 2 shows pseudocode for our weighted BFS implementation from Julienne [52]. Initially, the distances to all vertices are ∞ (Line 1), and the distance to the source vertex, *src*, is 0 (Line 19). Next, the algorithm buckets the vertices based on their current distance (Line 20). We note that a distance of ∞ places a vertex in a special "unknown" bucket. While the bucketing structure contains vertices, the algorithm extracts the next bucket (Lines 21 and 22) and applies the edgeMapData primitive (see Section 4) on all edges incident to the bucket (Line 23). The map function computes the distance along an edge (s, d, w_{sd}), updating the distance to d using a priorityWrite if $D[s] + w_{sd} < D[d]$ (Lines 5–13). The function also checks if the source vertex relaxing

ALGORITHM 2: wBFS

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

this edge is the first visitor to d during this round by performing a testAndSet on the *Relaxed* array, emitting d, and the old distance to d in the output vertexSubset if so.

The next step in the round applies a vertexMapVal on the augmented vertexSubset *Moved*. The map function first resets the *Relaxed* flag for each vertex (Line 15), and then computes the new bucket each relaxed vertex should move to using the getBucket primitive (Line 17). The output is an augmented vertexSubset *NewBuckets*, containing vertices and their destination buckets (Line 24). The last step updates the buckets for all vertices in *NewBuckets* (Line 25). The algorithm runs in $O(m)$ work in expectation and $O(\text{diam}(G) \log n)$ depth *whp* on the PW-BF model, as vertices use priorityWrite to write the minimum distance to a neighboring vertex in each round. The main change we made to this algorithm was to improve the cache-efficiency of edgeMapData using the block-based edgeMapBlocked algorithm described in Section 7.

General-Weight SSSP

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

Our implementation for this problem is the classic Bellman-Ford algorithm [49]. Algorithm 3 shows pseudocode for our frontier-based version of Bellman-Ford. The algorithm runs over a

ALGORITHM 3: Bellman-Ford

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

number of rounds. The initial frontier, F, consists of just the source vertex, *src* (Line 17). In each round, the algorithm applies edgeMap over F to produce a new frontier of vertices that had their shortest path distance decrease, and updates F to be this new frontier. The map function supplied to edgeMap (Line 7–13) tests whether the distance to a neighbor can be decrased, and uses a priorityWrite to atomically lower the distance (Line 10). Emitting a neighbor to the next frontier is done using a testAndSet on *Relaxed*, an array of flags indicating whether the vertex had its current shortest path distance decrease (Line 12). Finally, at the end of a round the algorithm resets the flags for all vertices in F (Line 24). After k rounds, the algorithm has correctly computed the distances to vertices that are within k hops from the source. Since any vertex is at most $n-1$ hops away from the source, if the number of rounds in the algorithm reaches n, we know that the input graph contains a negative weight cycle. The algorithm identifies vertices reachable from these cycles using the GeneralizedBFS algorithm (Algorithm 1) to compute all vertices reachable from the current frontier (Line 20). It sets the distance to the vertices with distances that are not ∞ (i.e., reachable from a negative weight cycle) to $-\infty$ (Line 21).

For inputs without negative-weight cycles, the algorithm runs in $O(\mathsf{diam}(G)m)$ work and $O(\mathsf{diam}(G)\log n)$ depth on the PW-BF model. If the graph contains a negative-weight cycle, the algorithm runs in $O(nm)$ work and $O(n \log n)$ depth on the PW-BF model. The main change we made to this algorithm compared to the Ligra implementation was to add a GeneralizedBFS implementation and invoke it in the case where the algorithm detects a negative weight cycle. We also improve its cache-efficiency by using the block-based version of edgeMap, edgeMapBlocked, which we describe in Section 7.

Single-Source Betweenness Centrality

Betweenness centrality is a classic tool in social network analysis for measuring the importance of vertices in a network [114]. Before describing the benchmark and our implementation, we introduce several definitions. Define σ_{st} to be the total number of s–t shortest paths, $\sigma_{st}(v)$ to be the number of s–t shortest paths that pass through v, and $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$ to be the ***pair-dependency*** of s and t on v.[4] The ***betweenness centrality*** of a vertex v is equal to $\sum_{s \neq v \neq t \in V} \delta_{st}(v)$, i.e. the sum of pair-dependencies of shortest-paths passing through v. Brandes [41] proposes an algorithm to compute the betweenness centrality values based on the following notion of 'dependencies': the ***dependency*** of a vertex r on a vertex v is $\delta_r(v) = \sum_{t \in V} \delta_{rt}(v)$. The single-source betweenness centrality problem in this paper is to compute the dependency values for each vertex given a source vertex, r. The dependency values for unreachable vertices should be set to ∞.

Our implementation is based on Brandes' algorithm, and follows the approach from Ligra [136]. We note that our implementation achieves speedups over the Ligra implementation by using contention-avoiding primitives from the GBBS interface. Our algorithm runs in $O(m)$ work and $O(\text{diam}(G) \log n)$ depth on the BF model (it does not require the fetchAndAdd primitive, as in the Ligra implementation, as we explain shortly). The algorithm works in two phases, which both rely on the structure of a BFS tree rooted at r. The first phase computes σ_{rv}, i.e., the number of shortest paths from the source, r, to each vertex v. In more detail, let $P_r(v)$ be the parents of a vertex v on the previous level of the BFS tree. The first phase computes $\sigma_{rv} = \sum_{u \in P_r(v)} \sigma_{ru}$ by processing the BFS tree in level order and summing the σ_{ru} values for all parents of v in the previous level. The second phase then applies the equation $\delta_r(v) = \sum_{w : v \in P_r(w)} \frac{\sigma_{rv}}{\sigma_{rw}} \cdot (1 + \delta_r(w))$ to compute the dependencies for each vertex by processing the levels of the BFS tree in reverse order.

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

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

where $D_r(v)$ is the set of all descendent vertices through v, i.e., $w \in V$ where a shortest path from r to w passes through v. These final scores can be converted to the dependency scores by first subtracting $\frac{1}{\sigma_{rv}}$ and then multiplying by σ_{rv}, since

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

Next, we discuss the main difference between our implementation and that of Ligra. The Ligra implementation is based on using edgeMap with an map function that uses the fetchAndAdd primitive to update the number of shortest paths (σ_{rv}) in the forward phase, and to update the inverted dependencies ($\zeta_r(v)$) in the reverse phase. The Ligra implementation thus combines the generation of the next BFS frontier with aggregating the number of shortest paths passing through a vertex in the first phase, or the inverted dependency contribution of the vertex in the second phase by using the fetchAndAdd primitive. In our implementation, we observed that for certain

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

graphs, especially those with skewed degree distribution, using a FETCHANDADD to sum up the contributions incurs a large amount of contention, and significant speedups (in our experiments, up to 2x on the Hyperlink2012 graph) can be obtained by (i) separating the computation of the next frontier from the computation of the σ_{rv} and $\delta_r(v)$ values in the two phases and (ii) computing the computation of σ_{rv} and $\delta_r(v)$ using the pull-based approach described below.

The pseudocode for our betweenness centrality implementation is shown in Algorithm 4. The algorithm runs in two phases. The first phase (Lines 26–31) computes a BFS tree rooted at the source vertex r using a NGHMAP using UPDATE, and COND defined identically to the BFS algorithm in Algorithm 1. After computing the new BFS frontier, F, the algorithm maps over the vertices in it using a VERTEXMAP (Line 28), and applies the AGGREGATEPATHCONTRIBUTIONS procedure for each vertex. This procedure (Lines 9–11) performs a reduction over all in-neighbors of the vertex to pull path-scores from vertices that are completed, i.e. $Completed[v] = \text{true}$ (Line 11). The algorithm then applies a second VERTEXMAP over F to mark these vertices as completed (Line 29). The frontier is then saved for use in the second phase (Line 30). At the end of the second phase we reset the *Status* values (Line 32).

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

Widest Path (Bottleneck Path)

The Widest Path, or Bottleneck Path benchmark in GBBS is to compute $\forall v \in V$ the maximum over all paths of the minimum weight edge on the path between a source vertex, *src*, and v. The algorithm is an important primitive, used for example in the Ford-Fulkerson maximum flow algorithm [49, 66], as well as other flow algorithms [19]. Sequentially, the algorithm can be solved as quickly as SSSP by using a modified version of Dijkstra's algorithm. We note that faster algorithms are known sequentially for sparse graphs [60]. For positive integer-weighted graphs, the problem can also be solved using the work-efficient bucketing data structure from Julienne [52]. The buckets, which represent the width classes, are initialized with the out-neighbors of the source, u, and the buckets are traversed using the *decreasing* order (from the largest bucket to the smallest bucket). Unlike the other applications in Julienne, using widest path is interesting since the bucket containing a vertex (the vertex priorities) only increase (in other applications in Julienne, the priorities can only decrease). The problem can also be solved using the Bellman-Ford approach described above by performing computations over the (max, min) semi-ring instead of the (min, +) semi-ring. Other than these changes, the pseudocode for the problem is identical to that of Algorithms 2 and 3.

$O(k)$-Spanner

Computing graph spanners is a fundamental problem in combinatorial graph algorithms and graph theory [120]. A graph H is a **k-*spanner*** of a graph G if $\forall u, v \in V$ connected by a path, $\text{dist}_G(u, v) \leq \text{dist}_H(u, v) \leq k \cdot \text{dist}_G(u, v)$ (equivalently, such a subgraph is called a spanner with ***stretch*** k). The spanner problem studied in this paper is to compute an $O(k)$ spanner for a given k.

Sequentially, classic results give elegant constructions of $(2k-1)$-spanners using $O(n^{1+1/k})$ edges, which are essentially the best possible assuming the girth conjecture [149]. In this paper, we

ALGORITHM 4: Betweenness Centrality

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

implement the spanner algorithm recently proposed by Miller, Peng, Xu, and Vladu (MPXV) [110]. The construction results in an $O(k)$-spanner with expected size $O(n^{1+1/k})$, and runs in $O(m)$ work and $O(k \log n)$ depth on the BF model.

The MPXV spanner algorithm (Algorithm 5) uses the low-diameter decomposition (LDD) algorithm, which will be described in Section 6.2. It takes as input a parameter k which controls the stretch of the spanner. The algorithm first computes an LDD with $\beta = \log n/(2k)$ (Line 3). The stretch of each LDD cluster is $O(k)$ *whp*, and so the algorithm includes all tree edges generated by the LDD in the spanner (Line 4). The algorithm handles inter-cluster edges by taking a single

ALGORITHM 5: $O(k)$-Spanner

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

inter-cluster edge between a boundary vertex and each neighboring cluster (Line 5). Our implementation uses a parallel hash table to select a single inter-cluster edge between two neighboring clusters.

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

6.2 Connectivity Problems

Low-Diameter Decomposition

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

Low-diameter decompositions (LDD) were first introduced in the context of distributed computing [11], and were later used in metric embedding, linear-system solvers, and parallel algorithms. Awerbuch presents a simple sequential algorithm based on ball growing that computes an $(\beta, O((\log n)/\beta)$ decomposition [11]. Miller, Peng, and Xu (MPX) [111] present a work-efficient parallel algorithm that computes a $(\beta, O((\log n)/\beta)$ decomposition. For each $v \in V$, the algorithm draws a start time, δ_v, from an exponential distribution with parameter β. The clustering is done by assigning each vertex u to the cluster v which minimizes $\text{dist}_G(u, v) - \delta_v$. This algorithm can be implemented by running a set of parallel breadth-first searches as follows. The first breadth-first search starts at the vertex with the largest start time, $\delta_{\max}$, and breadth-first searches start from other $v \in V$ once $\delta_{\max} - \delta_v$ steps have elapsed. In this paper, we present an implementation of the MPX algorithm which computes an $(2\beta, O(\log n/\beta))$ decomposition in $O(m)$ work and $O(\log^2 n)$ depth *whp* on the BF model. Our implementation is based on the non-deterministic LDD implementation from Shun et al. [140] (designed as part of a parallel connectivity implementation). The main changes in our implementation are to separate the LDD code from the connectivity implementation.

Algorithm 6 shows pseudocode for the modified version of the Miller-Peng-Xu algorithm from [140], which computes a $(2\beta, O(\log n/\beta))$ decomposition in $O(m)$ work and $O(\log^2 n)$ depth *whp* on the BF model. The algorithm allows ties to be broken arbitrarily when two searches visit a vertex in the same time-step, and one can show that this only affects the number of cut edges by a constant factor [140]. The LDD algorithm starts by first permuting the vertices into $O(\log n/\beta)$ batches, stored in an array B (Line 19). This partitioning simulates sampling from the exponential

ALGORITHM 6: Low Diameter Decomposition

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

distribution by randomly permuting the vertices in parallel (Line 15) and dividing the vertices in the permutation into $O(\log n/\beta)$ many batches (Line 16). After partitioning the vertices, the LDD algorithm performs a sequence of rounds, where in each round all vertices that are not already clustered in the next batch are added as new cluster centers. Each cluster then tries to acquire unclustered vertices adjacent to it (thus increasing its radius by 1). This procedure is sometimes referred to as *ball-growing* in the literature [12, 35, 111].

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

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

Connectivity

The connectivity problem is to compute a connectivity labeling of an undirected graph, i.e., a mapping from each vertex to a label such that two vertices have the same label if and only if there is a path between them in the graph. Connectivity can easily be solved sequentially in linear work using breadth-first or depth-first search. Parallel algorithms for connectivity have a long history; we refer readers to [140] for a review of the literature. Early work on parallel connectivity discovered many natural algorithms which perform $O(m \log n)$ work and poly-logarithmic depth [13, 122, 127, 135]. A number of optimal parallel connectivity algorithms were discovered in subsequent years [45, 67, 75, 76, 121, 123, 140], but to the best of our knowledge the recent algorithm by Shun et al. is the only linear-work polylogarithmic-depth parallel algorithm that is practical and has been studied experimentally [140].

In this paper, we implement the connectivity algorithm from Shun et al. [140], which runs in $O(m)$ expected work and $O(\log^3 n)$ depth *whp* on the BF model. The implementation uses the work-efficient algorithm for low-diameter decomposition (LDD) described above. One change we made to the implementation from [140] was to separate the LDD and contraction steps from the connectivity algorithm. Refactoring these sub-routines allowed us to express the main connectivity algorithm in about 50 lines of code.

The connectivity algorithm from Shun et al. [140] (Algorithm 7) takes as input an undirected graph G and a parameter $0 < \beta < 1$. It first runs the LDD algorithm, Algorithm 6 (Line 2), which decomposes the graph into clusters each with diameter $O(\log n/\beta)$, and βm inter-cluster edges in expectation. Next, it builds G' by contracting each cluster to a single vertex and adding inter-cluster edges while removing duplicate edges, self-loops, and isolated vertices (Line 3). It then checks if the contracted graph is empty (Line 4); if so, the current clusters are the components, and it returns the mapping from vertices to clusters (Line 5). Otherwise, it recurses on the contracted graph (Line 6) and returns the connectivity labeling produced by assigning each vertex to the label assigned to its cluster in the recursive call (Lines 7 and 8).

ALGORITHM 7: Connectivity

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

Spanning Forest

The spanning forest problem is to compute a subset of edges in the graph that represent a spanning forest. Finding spanning forests in parallel has been studied largely in conjunction with connectivity algorithms, since most parallel connectivity algorithms can naturally be modified to output a spanning forest (see [140] for a review of the literature).

Our spanning forest algorithm (Algorithm 8) is based on the connectivity algorithm from Shun et al. [140] which we described earlier. Our algorithm runs in runs in $O(m)$ expected work and $O(\log^3 n)$ depth *whp* on the BF model. The main difference in the spanning forest algorithm compared to the connectivity algorithm is to include all LDD edges at each level of the recursion (Line 4). These LDD edges are extracted using the *Parents* array returned by the LDD algorithm given in Algorithm 6. Recall that this array has size proportional to the number of vertices, and all

ALGORITHM 8: Spanning Forest

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

entries initialized to ∞. The LDD algorithm uses this array to store the BFS parent of each vertex v that joins a different vertex's cluster ($Clusters[v] \neq v$). The LDD edges are retrieved by checking for each index $i \in [0, n)$ whether $Parents[i] \neq \infty$ and if so taking $(i, Parents[i])$ as an LDD edge.

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

Biconnectivity

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

Sequentially, biconnectivity can be solved using the Hopcroft-Tarjan algorithm [80]. The algorithm uses depth-first search (DFS) to identify articulation points and requires $O(m + n)$ work to label all edges with their biconnectivity label. It is possible to parallelize the sequential algorithm using a parallel DFS, however, the fastest parallel DFS algorithm is not work-efficient [3]. Tarjan and Vishkin present the first work-efficient algorithm for biconnectivity [148] (as stated in the paper the algorithm is not work-efficient, but it can be made so by using a work-efficient connectivity algorithm). The same paper also introduces the Euler-tour technique, which can be used to compute subtree functions on rooted trees in parallel in $O(n)$ work and $O(\log^2 n)$ depth on the BF model. Another approach relies on the fact that biconnected graphs admit open ear decompositions to solve biconnectivity efficiently [103, 126].

In this paper, we implement the Tarjan-Vishkin algorithm for biconnectivity in $O(m)$ expected work and $O(\max(\mathsf{diam}(G) \log n, \log^3 n))$ depth on the FA-BF model. Our implementation first computes connectivity labels using our connectivity algorithm, which runs in $O(m)$ expected work and $O(\log^3 n)$ depth *whp* and picks an arbitrary source vertex from each component. Next, we compute a spanning forest rooted at these sources using breadth-first search, which runs in $O(m)$ work and

ALGORITHM 9: Biconnectivity

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

$O(\text{diam}(G) \log n)$ depth. We compute the subtree functions *Low*, *High*, and *Size* for each vertex by running leaffix and rootfix sums on the spanning forests produced by BFS with fetchAndAdd, which requires $O(n)$ work and $O(\text{diam}(G) \log n)$ depth. Finally, we compute an implicit representation of the biconnectivity labels for each edge, using an idea from [23]. This step computes per-vertex labels by removing all critical edges and computing connectivity on the remaining graph. The resulting vertex labels can be used to assign biconnectivity labels to edges by giving tree edges the connectivity label of the vertex further from the root in the tree, and assigning non-tree edges the label of either endpoint. Summing the cost of each step, the total work of this algorithm is $O(m)$ in expectation and the total depth is $O(\max(\text{diam}(G) \log n, \log^3 n))$ *whp*.

Algorithm 9 shows the Tarjan-Vishkin biconnectivity algorithm. It first computes a spanning forest of G and roots the trees in this forest arbitrarily (Lines 15 and 16). Next, the algorithm computes a preorder numbering, *Preorder*, with respect to the roots (Line 17). It then computes the subtree functions $Low(v)$ and $High(v)$ for each $v \in V$, which are the minimum and maximum preorder numbers respectively of all non-tree edges (u, w) where u is a vertex in v's subtree (Line 18). It also computes $Size(v)$, the size of each vertex's subtree. Observe that one can determine whether the parent of a vertex u, p_u is an articulation point by checking $Preorder[p_u] \leq Low(u)$ and $High(u) < Preorder[p_u] + Size[p_u]$. Following [23], we refer to this set of tree edges (u, p_u), where p_u is an articulation point, as ***critical edges*** (Line 9). The last step of the algorithm is to compute a connectivity labeling of the graph with all critical edges removed. Our algorithm removes the critical edges using the packGraph primitive (see Section 4).

Given this final connectivity labeling, the biconnectivity label of an edge (u, v) is the connectivity label of the vertex that is further from the root of the tree. The query data structure can thus report biconnectivity labels of edges in $O(1)$ time using $2n$ words of memory; each vertex just stores its connectivity label, and the vertex ID of its parent in the rooted forest (for an edge (u, v) either one vertex is the parent of the other, which determines the vertex further from the root, or

neither is the parent of the other, which implies that both are the same distance from the root). The same query structure can also report whether an edge is a bridge in $O(1)$ time. We refer the reader to [23] for more details. The low space usage of this query structure is important for our implementations as storing a biconnectivity label per-edge explicitly would require a prohibitive amount of memory for large graphs.

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

Minimum Spanning Forest

The minimum spanning forest problem is to compute a spanning forest of the graph with minimum possible total edge weight. Borůvka gave the first known sequential and parallel algorithm for computing a minimum spanning forest (MSF) [40]. Significant effort has gone into finding linear-work MSF algorithms both in the sequential and parallel settings [45, 87, 121]. Unfortunately, these linear-work parallel algorithms are highly involved and do not seem to be practical. Significant effort has also gone into designing practical parallel algorithms for MSF; we discuss relevant experimental work in Section 8. Due to the simplicity of Borůvka, many parallel implementations of MSF use variants of it.

In this paper, we present an implementation of Borůvka's algorithm that runs in $O(m \log n)$ work and $O(\log^2 n)$ depth *whp* on the PW-BF model. Our implementation is based on a recent implementation of Borůvka by Zhou [155] that runs on the edgelist format (graphs represented as a sequence of edges, see Section 3). We made several changes to the algorithm which improve performance and allow us to solve MSF on very large graphs stored in the CSR/CSC format (defined in Section 3). Storing an integer-weighted graph in edgelist format would require well over 1TB of memory to represent the edges in the Hyperlink2012 graph alone.

Algorithm 10 shows the pseudocode for our implementation of Borůvka's algorithm designed for the CSR/CSC format. Our implementation uses an implementation of Borůvka (Lines 2–21) that works over an edgelist as a subroutine; to make it efficient in practice, we ensure that the size of the lists passed to it are much smaller than m. The edgelist-based implementation is based on shortcutting using pointer-jumping instead of contraction. The main MSF algorithm (Lines 22–33) maintains a *Parents* array that represents the connected components that have been found by the algorithm so far. Initially, each vertex is in its own component (Line 25). The main algorithm performs a constant number of *filtering* steps on a small number of the lowest-weight edges that are extracted from the graph. Each filtering step first solves an approximate k-th smallest problem in order to determine a weight threshold, which is either the weight of approximately the $3n/2$-th lightest edge, or the max edge weight if the maximum number of filtering rounds are reached (Line 27). This step can be easily implemented using the vertex primitives in Section 4 and binary search. Edges lighter than the threshold are extracted using the EXTRACTEDGES primitive, defined in Section 4 (Line 29). The algorithm then runs Borůvka on this subset of edges (Line 30), which we describe next. Borůvka returns edges that are in the minimum spanning forest, and additionally compresses the *Parents* array based on the new forest edges. Lastly, the main algorithm removes edges that are now contained in the same component using the PACKGRAPH primitive (Line 31).

ALGORITHM 10: Minimum Spanning Forest

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

The edgelist-based Borůvka implementation (Lines 2–21) takes as input the number of vertices and a prefix of the lowest weight edges currently in the graph. The forest is initially empty (Line 3). The algorithm runs over a series of rounds. Within a round, the algorithm first initializes an array P of (weight, index) pairs for all vertices (Line 5). Next, it loops in parallel over all edges in E and perform priorityWrites to P based on the weight on both endpoints of the edge (Lines 8 and 9). This step writes the weight and index-id of a minimum-weight edge incident to a vertex v into $P[v]$. Next, for each vertex u that found an MSF edge incident to it, i.e., $P[u] \neq (\infty, \infty)$ (Line 10), the algorithm determines v, the neighbor of u along this MSF edge (Lines 11–12). If v also selected (u, v, w) as its MSF edge, the algorithm deterministically sets the vertex with lower id to be the root of the tree (Line 14) and the vertex with higher id to point to lower one (Line 16). Otherwise, u joins v's component (Line 16). Lastly, the algorithm performs several clean-up steps. First, it updates the forest with all newly identified MSF edges (Line 17). Next, it performs pointer-jumping (see Section 3) to compress trees created in *Parents* (Line 18). Note that the pointer-jumping step can be work-efficiently implemented in $O(\log n)$ depth *whp* on the BF model [31]. Finally, it relabels the

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

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

We point out that the filtering idea used in our main algorithm is similar to the theoretically-efficient algorithm of Cole et al. [45], except that instead of randomly sampling edges, our filtering procedure selects a linear number of the lowest weight edges. Each filtering step costs $O(m)$ work and $O(\log m)$ depth, but as we only perform a constant number of steps before processing the rest of the remaining graph, the filtering steps do not affect the work and depth asymptotically. In practice, most of the edges are removed after 3–4 filtering steps, and so the remaining edges can be copied into an edgelist and solved in a single Borůvka step. We also note that as the edges are initially represented in both directions, we can pack out the edges so that each undirected edge is only inspected once (we noticed that earlier edgelist-based implementations stored undirected edges in both directions).

Strongly Connected Components

The strongly connected components problem is to compute a labeling L that maps each vertex to a unique label for its strongly connected component (i.e., $L[u] = L[v]$ iff there is a directed path from u to v and from v to u). Tarjan's algorithm is the textbook sequential algorithm for computing the strongly connected components (SCCs) of a directed graph [49]. As it uses depth-first search, we currently do not know how to efficiently parallelize it [3]. The current theoretical state-of-the-art for parallel SCC algorithms with polylogarithmic depth reduces the problem to computing the transitive closure of the graph. This requires $\tilde{O}(n^3)$ work using combinatorial algorithms [68], which is significantly higher than the $O(m + n)$ work done by sequential algorithms. As the transitive-closure based approach performs a significant amount of work even for moderately sized graphs, subsequent research on parallel SCC algorithms has focused on improving the work while potentially sacrificing depth [33, 48, 65, 131]. Conceptually, these algorithms first pick a random pivot and use a reachability-oracle to identify the SCC containing the pivot. They then remove this SCC, which partitions the remaining graph into several disjoint pieces, and recurse on the pieces.

In this paper, we present the first implementation of the SCC algorithm from Blelloch et al. [33], shown in Algorithm 11. We refer the reader to Section 6.2 of [33] for proofs of correctness and its work and depth bounds. The algorithm is similar in spirit to randomized quicksort. The algorithm first sets the initial label for all vertices as ∞ and marks all vertices as not done (Lines 3 and 4). Next, it randomly permutes the vertices and partitions them into $\log n$ batches whose sizes increase geometrically (Line 2). This pseudocode for Partition is given in Algorithm 6. Specifically, B_i contains all vertices that are part of the i-th batch. The variable d is a counter tracking the number of vertices that the algorithm has finished processing. It processes the batches one at a time.

For each batch, it first computes *Centers*, which are the vertices in this batch that are not yet done (Line 7). The next step calls *MarkReachable* from the centers on both G and the transposed graph, G^{T} (Lines 8–9). *MarkReachable* takes the set of centers and uses a variant of a breadth-first search to compute the sets *OutL* (*InL*), which for the j'th center $c_j \in B_i$ includes all $(v, d + j)$ pairs for vertices v that c_j can reach through its out-edges (in-edges). We describe this procedure in

ALGORITHM 11: Strongly Connected Components

```
1:  procedure SCC(G(V, E))
2:      B := Partition(V, 1)   ▷ permute and group vertices in O(log n) batches of increasing size (see Alg. 6)
3:      L[0, ..., n) := ∞
4:      Done[0, ..., n) := false
5:      d := 0                 ▷ counter used to assign a uniquelabel to each center based on its position in B
6:      for i ∈ [0, |B|) do
7:          Centers :={v ∈ B_i | !Done[i]}          ▷ vertices starting in the i-th batch that are not yet done
8:          OutL := MarkReachable(G, Centers)      ▷ pairs (u, d + j) s.t. the j-th center in B_i reaches u in G
9:          InL := MarkReachable(G^T, Centers)     ▷ pairs (u, d + j) s.t. the j-th center in B_i reaches u in G^T
10:         for (u, l) ∈ InL ∩ OutL in parallel do
11:             Done[u] := true                                   ▷ mark this vertex as done
12:             priorityWrite(&L[u], l, <)  ▷ final value is l' = d + j' where j' = arg min_j{B_i[j] in u's SCC }
13:         packGraph(G, fn (u, v) → return                       ▷ preserve edges within the same subproblem
14:             |InL[u]| = |InL[v]| and |OutL[u]| = |OutL[v]|)
15:         d := d + |B_i|                  ▷ increment d by the number of finished centers in the i-th batch
16:     return L
```

more detail below. Finally, the algorithm computes all (u, l) pairs in the intersection of *InL* and *OutL* in parallel (Line 10). For each pair, the algorithm first marks the vertex as done (Line 11). It then performs a priorityWrite to atomically try and update the label of the vertex to l (Line 12). After the parallel loop on Line 10 finishes, the label for a vertex u that had some vertex in its SCC appear as a center in this batch will be set to $l' = d + j$, where $j' = \arg\min_j\{B_i[j]$ in u's SCC $\}$, i.e., it the unique label for the vertex with minimum rank in the permutation B contained in u's SCC.

The last step of the algorithm refines the subproblems in the graph by partitioning it, i.e., deleting all edges which the algorithm identifies as not being in the same SCC. In our implementation, this step is implemented using the packGraph primitive (Line 13), which considers every directed edge in the graph and only preserves edges (u, v) where the number of centers reaching u and v in *InL* are equal (respectively the number of centers reaching them in *OutL*). We note that the algorithm described in Blelloch et al. [33] suggests that to partition the graph, each reachability search can check whether any edge (u, v) where one endpoint is reachable in the search, and the other is not, can be cut (possibly cutting some edges multiple times). The benefit of our approach is that we can perform a single parallel scan over the edges in the graph and pack out a removed edge exactly once. Our implementation runs in $O(m \log n)$ expected work and $O(\mathsf{diam}(G) \log n)$ depth *whp* on the PW-BF model.

One of the challenges in implementing this SCC algorithm is how to compute reachability information from multiple vertices (the centers) simultaneously. Our implementation explicitly materializes the forward and backward reachability sets for the set of centers that are active in the current phase. The sets are represented as hash tables that store tuples of vertices and labels, (u, l), representing a vertex u in the same subproblem as the vertex c with label l that is visited by a directed path from c. We explain how to make the hash table technique practical in Section 7.3. The reachability sets are computed by running simultaneous breadth-first searches from all active centers. In each round of the BFS, we apply edgeMap to traverse all out-edges (or in-edges) of the current frontier. When we visit an edge (u, v) we try to add u's center IDs to v. If u succeeds in adding any IDs, it testAndSet's a visited flag for v, and returns it in the next frontier if the testAndSet succeeded. Each BFS requires at most $O(\mathsf{diam}(G))$ rounds as each search adds the same labels in each round as it would have had it run in isolation.

We also implement an optimized search for the first phase, which just runs two regular BFSs over the in-edges and out-edges from a single pivot and stores the reachability information in bit-vectors instead of hash-tables. It is well known that many directed real-world graphs have a single massive strongly connected component, and so with reasonable probability the first vertex in the permutation will find this giant component [43]. Our implementation also supports a *trimming optimization* that is used by some papers in the literature [106, 144], which eliminates trivial SCCs by removing any vertices that have zero in- or out-degree. We implement a procedure that recursively trims until no zero in- or out-degree vertices remain, or until a maximum number of rounds are reached, although in practice we found that a single trimming step is sufficient to remove the majority of trivial vertices on our graph inputs.

6.3 Covering Problems

Maximal Independent Set

The maximal independent set problem is to compute a subset of vertices U such that no two vertices in U are neighbors, and all vertices in $V \setminus U$ have a neighbor in U. Maximal independent set (MIS) and maximal matching (MM) are easily solved in linear work sequentially using greedy algorithms. Many efficient parallel maximal independent set and matching algorithms have been developed over the years [5, 25, 32, 82, 89, 98]. Blelloch et al. show that when the vertices (or edges) are processed in a random order, the sequential greedy algorithms for MIS and MM can be parallelized efficiently and give practical algorithms [32]. Recently, Fischer and Noever showed an improved depth bound for these MIS and MM algorithms [64].

In this paper, we implement the rootset-based algorithm for MIS from Blelloch et al. [32] which runs in $O(m)$ work and $O(\log^2 n)$ depth *whp* on the FA-BF model (using the improved depth analysis of Fischer and Noever [64]). To the best of our knowledge this is the first implementation of the rootset-based algorithm; the implementations from [32] are based on processing appropriately-sized prefixes of an order generated by a random permutation P, and have linear expected work and a larger depth bound. Our implementation of the rootset-based algorithm works on a priority-DAG defined by directing edges in the graph from the higher-priority endpoint to the lower-priority endpoint. In each round, we add all roots of the DAG into the MIS, compute $N(Roots)$, the neighbors of the rootset that are still active, and finally decrement the priorities of $N(N(Roots))$. As the vertices whose priorities we decrement are at arbitrary depths in the priority-DAG, we only decrement the priority along an edge (u, v) if $P[u] < P[v]$ (we could also explicitly run the algorithm on the graph directed according to P, which would avoid this check). The algorithm runs in $O(m)$ work as we process each vertex and edge once; the depth bound is $O(\log^2 n)$ as the priority-DAG has $O(\log n)$ depth *whp* [64], and each round takes $O(\log n)$ depth. We were surprised that this implementation usually outperforms the prefix-based implementation from [32], while also being simple to implement.

Our implementation of the rootset-based MIS algorithm is shown in Algorithm 12. The algorithm first randomly orders the vertices with a random permutation P (Line 1). It then computes an array *Priority* where each vertex is associated with the count of its number of neighbors that have higher priority than it with respect to the permutation P. This computation is done using the countNghs primitive from Section 4 (Line 16). Next, on Line 17 we compute the initial rootset, *Roots*, which is the set of all vertices that initially have priority 0. In each round, the algorithm adds the roots to the independent set, I (Line 21), and computes the set of covered (i.e., removed) vertices, which are neighbors of the rootset that are still active ($Priority[v] > 0$). This step is done using edgeMap over *Roots*, where the map and condition function are defined similarly to BFS, returning true for a neighboring vertex if and only if it has not been visited before (the testAnd-

ALGORITHM 12: Maximal Independent Set

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

Set to *Flags* succeeds). The algorithm also sets the *Priority* values of these vertices to 0 (Line 23), which prevents them from being considered as potential roots in the remainder of the algorithm. Next, the algorithm updates the number of finished vertices (Line 24). Finally, the algorithm computes the next set of roots using a second edgeMap. The map function (Lines 9–12) decrements the priority of all neighbors v visited over an edge (u, v) where $u \in Covered$ and $P[u] < P[v]$ using a fetchAndAdd that returns true for a neighbor v if this edge decrements its priority to 0.

Maximal Matching

The maximal matching problem is to compute a subset of edges $E' \subseteq E$ such that no two edges in E' share an endpoint, and all edges in $E \setminus E'$ share an endpoint with some edge in E'. Our maximal matching implementation is based on the prefix-based algorithm from [32] that takes $O(m)$ expected work and $O(\log^2 m)$ depth *whp* on the PW-BF model (using the improved depth shown in [64]). We had to make several modifications to run the algorithm on the large graphs in our experiments. The original code from [32] uses an edgelist representation, but we cannot directly use this implementation as uncompressing all edges would require a prohibitive amount of memory for large graphs. Instead, as in our MSF implementation, we simulate the prefix-based approach by performing a constant number of *filtering* steps. Each filter step packs out $3n/2$ of the highest priority edges, randomly permutes them, and then runs the edgelist based algorithm on the prefix. After computing the new set of edges that are added to the matching, we filter the remaining graph and remove all edges that are incident to matched vertices. In practice, just 3–4 filtering steps are sufficient to remove essentially all edges in the graph. The last step uncompresses

ALGORITHM 13: Maximal Matching

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

any remaining edges into an edgelist and runs the prefix-based algorithm. The filtering steps can be done within the work and depth bounds of the original algorithm.

Our implementation of the prefix-based maximal matching algorithm from Blelloch et al. [32] is shown in Algorithm 13. The algorithm first creates the array *matched*, sets all vertices to be unmatched, and initializes the matching to empty (Line 11). The algorithm runs a constant number of filtering rounds, as described above, where each round fetches some number of highest priority edges that are still active (i.e., neither endpoint is incident to a matched edge). First, it calculates the number of edges to extract (Line 15). It then extracts the highest priority edges using the packGraph primitive. The function supplied to packGraph checks whether an edge e is one of the highest priority edges, and if so, emits it in the output edgelist, P and removes this edge from the graph. Our implementation calculates edge priorities by hashing the edge pair. It selects whether an edge is in the prefix by comparing each edge's priority with the priority of approximately the *toExtract*-th smallest priority, computed using approximate median.

Next, the algorithm applies the parallel greedy maximal matching algorithm (Lines 2–9) on it. The parallel greedy algorithm first randomly permutes the edges in the prefix (Line 4). It then repeatedly finds the set of edges that have the lowest rank in the prefix amongst all other edges incident to either endpoint (Line 6), adds them to the matching (Line 7), and filters the edges based on the newly matched edges (Line 8). The edges matched by the greedy algorithm are returned to the MaximalMatching procedure (Line 9). We refer to [32, 64] for a detailed description of the prefix-based algorithm that we implement, and a proof of the work and depth of the ParallelGreedyMM algorithm.

The last steps within a round are to filter the remaining edges in the graph based on the newly matched edges using the packGraph primitive (Line 20). The supplied predicate does not return

any edges in the output edgelist, and packs out any edge incident to the partial matching, W. Lastly, the algorithm adds the newly matched edges to the matching Line 21. We note that applying a constant number of filtering rounds before executing ParallelGreedyMM does not affect the work and depth bounds.

Graph Coloring

The graph coloring problem is to compute a mapping from each $v \in V$ to a color such that for each edge $(u, v) \in E$, $C(u) \neq C(v)$, using at most $\Delta + 1$ colors. As graph coloring is NP-hard to solve optimally, algorithms like greedy coloring, which guarantees a $(\Delta + 1)$-coloring, are used instead in practice, and often use much fewer than $(\Delta + 1)$ colors on real-world graphs [77, 151]. Jones and Plassmann (JP) parallelize the greedy algorithm using linear work, but unfortunately adversarial inputs exist for the heuristics they consider that may force the algorithm to run in $O(n)$ depth. Hasenplaugh et al. introduce several heuristics that produce high-quality colorings in practice and also achieve provably low-depth regardless of the input graph. These include LLF (largest-log-degree-first), which processes vertices ordered by the log of their degree and SLL (smallest-log-degree-last), which processes vertices by removing all lowest log-degree vertices from the graph, coloring the remaining graph, and finally coloring the removed vertices. For LLF, they show that it runs in $O(m + n)$ work and $O(L \log \Delta + \log n)$ depth, where $L = \min\{\sqrt{m}, \Delta\} + \log^2 \Delta \log n / \log \log n$ in expectation.

In this paper, we implement a synchronous version of Jones-Plassmann using the LLF heuristic, which runs in $O(m + n)$ work and $O(L \log \Delta + \log n)$ depth on the FA-BF model. The algorithm is implemented similarly to our rootset-based algorithm for MIS. In each round, after coloring the roots we use a fetchAndAdd to decrement a count on our neighbors, and add the neighbor as a root on the next round if the count is decremented to 0.

Algorithm 14 shows our synchronous implementation of the parallel LLF-Coloring algorithm from [77]. The algorithm first computes priorities for each vertex in parallel using the countNghs primitive (Line 14). This step computes the number of neighbors of a vertex that must run before it by applying the *countFn* predicate (Line 13). This predicate function returns true for a (u, v) edge to a neighbor v if the log-degree of v is greater than u, or, if the log-degrees are equal whether v has a lower-rank in a permutation on the vertices (Line 1) than v. Next, the algorithm computes the vertexSubset *Roots* (Line 15) which consists of all vertices that have no neighbors that are still uncolored that must be run before them based on *countFn*. Note that *Roots* is an independent set. The algorithm then loops while some vertex remains uncolored. Within the loop, it first assigns colors to the roots in parallel (Line 18) by setting each root to the first unused color in its neighborhood (Lines 5–6). Finally, it updates the number of finished vertices by the number of roots (Line 19) and computes the next rootset by applying edgeMap on the rootset with a map function that decrements the priority over all (u, v) edges incident to *Roots* where $Priority[v] > 0$. The map function returns true only if the priority decrement decreases the priority of the neighboring vertex to 0 (Line 8).

Approximate Set Cover

The set cover problem can be modeled by a bipartite graph where sets and elements are vertices, with an edge between a set and an element if and only if the set covers that element. The approximate set cover problem is as follows: given a bipartite graph $G = (V = (S, E), A)$ representing an unweighted set cover instance, compute a subset $S' \subseteq S$ such that $\cup_{s \in S'} N(s) = E$ and $|S'|$ is an $O(\log n)$-approximation to the optimal cover. Like graph coloring, the set cover problem is NP-hard to solve optimally, and a sequential greedy algorithm computes an H_n-approximation in $O(m)$ time for unweighted sets, and $O(m \log m)$ time for weighted sets, where $H_n = \sum_{k=1}^{n} 1/k$ and m is

ALGORITHM 14: LLF Graph Coloring

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

the sum of the sizes of the sets (or the number of edges in the graph). There has been significant work on finding work-efficient parallel algorithms that achieves an H_n-approximation [24, 36, 37, 91, 124].

Algorithm 15 shows pseudocode for the Blelloch et al. algorithm [36] which runs in $O(m)$ work and $O(\log^3 n)$ depth on the PW-BF model. Our presentation here is based on the bucketing-based implementation from Julienne [52], with one significant change regarding how sets acquire elements which we discuss below. The algorithm first buckets the sets based on their degree, placing a set covering D elements into $\lfloor \log_{1+\epsilon} D \rfloor$-th bucket (Line 24). It then processes the buckets in decreasing order (Lines 26–38). In each round, the algorithm extracts the highest bucket (*Sets*) (Line 26) and packs out the adjacency lists of vertices in this bucket to remove edges to neighbors that are covered in prior rounds (Line 27). The output is an augmented vertexSubset, *SetsD*, containing each set along with its new degree after packing out all dead edges. It then maps over *SetsD*, updating the degree in D for each set with the new degree (Line 28). The algorithm then filters *SetsD* to build a vertexSubset*Active*, which contains sets that have sufficiently high degree to continue in this round (Line 29).

The next few steps of the algorithm implement one step of MaNIS (Maximal Nearly-Independent Set) [36], to compute a set of sets from *Active* that have little overlap. First, the algorithm assigns a random priority to each currently active set using a random permutation, storing the priorities in the array π (Lines 30–31). Next, it applies edgeMap (Line 32) where the map function (Line 12) uses a priority-write on each (s, e) edge to try and acquire an element e using the priority of the visiting set, $\pi[s]$. It then computes the number of elements each set successfully acquired using the srcCount primitive (Line 33) with the predicate WonElm (Line 10) that checks whether the minimum value stored at an element is the unique priority for the set. The final MaNIS step maps over the vertices and the number of elements they successfully acquired (Line 34) with the map function WonEnough (Lines 13–16) which adds sets that covered enough elements to the cover.

ALGORITHM 15: Approximate Set Cover

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

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

Our implementation of approximate set cover in this paper is based on the implementation from Julienne [52], and we refer to this paper for more details about the bucketing-based implementation. The main change we made in this paper is to ensure that we correctly set random priorities

for active sets in each round of the algorithm. Both the implementation in Julienne as well as an earlier implementation of the algorithm [37] use the original IDs of sets instead of picking random priorities for all sets that are active on a given round. This approach can cause very few vertices to be added in each round on meshes and other graphs with a large amount of symmetry. Instead, in our implementation, for $\mathcal{A}_S$, the active sets on a round, we generate a random permutation of $[0, \ldots, |\mathcal{A}_S| - 1]$ and write these values into a pre-allocated dense array with size proportional to the number of sets (Lines 30—31). We give experimental details regarding this change in Section 8.

6.4 Substructure Problems

k-core

A ***k-core*** of a graph is a maximal subgraph H where the degree of every vertex in H is $\geq k$. The ***coreness*** of a vertex is the maximum k-core a vertex participates in. The k-core problem in this paper is to compute a mapping from each vertex to its coreness value. k-cores were defined independently by Seidman [132], and by Matula and Beck [104] who also gave a linear-time algorithm for computing the coreness value of all vertices. Anderson and Mayr showed that k-core (and therefore coreness) is in NC for $k \leq 2$, but is P-complete for $k \geq 3$ [7]. The Matula and Beck algorithm is simple and practical—it first bucket-sorts vertices by their degree, and then repeatedly deletes the minimum-degree vertex. The affected neighbors are moved to a new bucket corresponding to their induced degree. As each edge in each direction and vertex is processed exactly once, the algorithm runs in $O(m + n)$ work. In [52], the authors gave a parallel algorithm based on bucketing that runs in $O(m + n)$ expected work, and $\rho \log n$ depth *whp.* ρ is the peeling-complexity of the graph, defined as the number of rounds to peel the graph to an empty graph where each peeling step removes all minimum degree vertices.

Algorithm 16 shows pseudocode for the work-efficient k-core algorithm from Julienne [52] which computes the coreness values of all vertices. The algorithm initializes the initial coreness value of each vertex to its degree (Line 36), and inserts the vertices into a bucketing data-structure based on their degree (Line 4). In each round, while all of the vertices have not yet been processed the algorithm performs the following steps. It first removes (or peels) the vertices in the minimum bucket, k (Line 7). Next, it computes the number of edges removed from each neighbor using the nghCount primitive. The apply function supplied to the primitive (Lines 10–18) takes a pair of a vertex, and the number of incident edges removed (v, *edgesRemoved*), updates the current coreness of the vertex v and emits a vertex and bucket identifier into the output vertexSubset if and only if the vertex needs to move to a new bucket (the return value of the getBucket primitive). The output is an augmented vertexSubset where each vertex is augmented with the bucket (a value of type bktdest) that it moves to. The last step is to update the buckets of affected neighbors (Line 20). Once all buckets have been processed (all cores have been peeled), the algorithm returns the array *Coreness*, which contains the final coreness values of each vertex at the end of the algorithm.

Approximate Densest Subgraph

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

The problem is a classic graph optimization problem that admits exact polynomial-time solutions using either a reduction to flow [70] or LP-rounding [44]. In his paper, Charikar also gives a simple $O(m + n)$ work 2-approximation algorithm based on computing a degeneracy ordering of

ALGORITHM 16: k-core (Coreness)

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

the graph, and taking the maximum density subgraph over all suffixes of the degeneracy order.[5] The problem has also received attention in parallel models of computation [17, 18]. Bahmani et al. give a $(2+\epsilon)$-approximation running in $O(\log_{1+\epsilon} n)$ rounds of MapReduce [18]. Subsequently, Bahmani et al. [17] showed that a $(1+\epsilon)$-approximation can be found in $O(\log n/\epsilon^2)$ rounds of MapReduce by using the multiplicative-weights approach on the dual of the natural LP for densest subgraph. To the best of our knowledge, it is open whether the densest subgraph problem can be exactly solved in NC.

In this paper, we implement the elegant $(2+\epsilon)$-approximation algorithm of Bahmani et al. (Algorithm 17). Our implementation of the algorithm runs in $O(m+n)$ work and $O(\log_{1+\epsilon} n \log n)$ depth. The algorithm starts with a candidate subgraph, S, consisting of all vertices, and an empty approximate densest subgraph $S_{\max}$ (Lines 4–5). It also maintains an array with the induced degree of each vertex in the array D, which is initially just its degree in G (Line 3). The main loop iteratively peels vertices with degree below the density threshold in the current candidate subgraph (Lines 6–16). Specifically, it first finds all vertices with induced degree less than $2(1+\epsilon)\mathcal{D}(S)$ (Line 7). Next, it calls NGHCOUNT (see Section 4), which computes for each neighbor of R the number of incident edges removed by deleting vertices in R from the graph, and updates the neighbor's degree in D (Line 17). Finally, it removes vertices in R from S (Line 14). If the density of the updated subgraph S is greater than the density of $S_{\max}$, the algorithm updates $S_{\max}$ to be S.

Bahmani et al. show that this algorithm removes a constant factor of the vertices in each round, but do not consider the work or total number of operations performed by their algorithm. We briefly sketch how the algorithm can be implemented in $O(m+n)$ work and $O(\log_{1+\epsilon} n \log n)$ depth. Instead of computing the density of the current subgraph by scanning all edges, we maintain

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

ALGORITHM 17: Approximate Densest Subgraph

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

it explicitly using an array, D (Line 3) which tracks the degrees of vertices still in S, and update D as vertices are removed from S. Each round of the algorithm does work proportional to vertices in S to compute R (Line 7) but since S decreases by a constant factor in each round the work of these steps to obtain R is $O(n)$ over all rounds. Updating D can be done by computing the number of edges going between R and S which are removed, which only requires scanning edges incident to vertices in R using nghCount (Line 13). Therefore, the edges incident to each vertex are scanned exactly once (in the round when it is included in R) and so the algorithm performs $O(m + n)$ work. The depth is $O(\log_{1+\epsilon} n \log n)$ since there are $O(\log_{1+\epsilon} n)$ rounds each of which perform a filter and nghCount which both run in $O(\log n)$ depth.

We note that an earlier implementation of our algorithm used the edgeMap primitive combined with fetchAndAdd to decrement degrees of neighbors of R. We found that since a large number of vertices are removed in each round, using fetchAndAdd can cause significant contention, especially on graphs containing vertices with high degrees. Our implementation uses a work-efficient histogram procedure to implement nghCount (see Section 7) which updates the degrees while incurring very little contention.

Triangle Counting

The triangle counting problem is to compute the global count of the number of triangles in the graph. Triangle counting has received significant recent attention due to its numerous applications in Web and social network analysis. There have been dozens of papers on sequential triangle counting (see e.g., [6, 83, 93, 117, 118, 129, 130], among many others). The fastest algorithms rely on matrix multiplication and run in either $O(n^\omega)$ or $O(m^{2\omega/(1+\omega)})$ work, where ω is the best matrix multiplication exponent [6, 83]. The fastest algorithm that does not rely matrix multiplication requires $O(m^{3/2})$ work [93, 129, 130], which also turns out to be much more practical. Parallel algorithms with $O(m^{3/2})$ work have been designed [1, 97, 142], with Shun and Tangwongsan [142] showing an algorithm that requires $O(\log n)$ depth on the BF model.[6] The implementation from [142] parallelizes Latapy's *compact-forward* algorithm, which creates a directed graph DG where an edge

[6]The algorithm in [142] was described in the Parallel Cache Oblivious model, with a depth of $O(\log^{3/2} n)$.

ALGORITHM 18: Triangle Counting

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

$(u, v) \in E$ is kept in DG iff $d(u) < d(v)$. Although triangle counting can be done directly on the undirected graph in the same work and depth asymptotically, directing the edges helps reduce work, and ensures that every triangle is counted exactly once.

In this paper we implement the triangle counting algorithm described in [142] (Algorithm 18). The algorithm first uses the filterGraph primitive (Line 4) to direct the edges in the graph from lower-degree to higher-degree, breaking ties lexicographically (Line 2). It then maps over all vertices in the graph in parallel (Line 6), and for each vertex performs a sum-reduction over its out-neighbors, where the value for each neighbor is the intersection size between the directed neighborhoods $N^+(u)$ and $N^+(v)$ (Line 7).

We note that we had to make several significant changes to the implementation in order to run efficiently on large compressed graphs. First, we parallelized the creation of the directed graph; this step creates a directed graph encoded in the parallel-byte format in $O(m)$ work and $O(\log n)$ depth using the filterGraph primitive. We also parallelized the merge-based intersection algorithm to make it work in the parallel-byte format. We give more details on these techniques in Section 7.

6.5 Eigenvector Problems

PageRank

PageRank is a centrality algorithm first used at Google to rank webpages [42]. The algorithm takes a graph $G = (V, E)$, a damping factor $0 \leq \gamma \leq 1$ and a constant ϵ which controls convergence. Initially, the PageRank of each vertex is $1/n$. In each iteration, the algorithm updates the PageRanks of the vertices using the following equation:

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

The PageRank algorithm terminates once the l_1 norm of the differences between PageRank values between iterations is below ϵ. The algorithm implemented in this paper is an extension of the implementation of PageRank described in Ligra [136]. The main changes are using a contention-avoiding reduction primitive, which we describe in more detail below. Some PageRank implementations used in practice actually use an algorithm called PageRank-Delta [96], which modifies PageRank by only activating a vertex if its PageRank value has changed sufficiently. However, the work and depth of this algorithm are the same as that of PageRank in the worst case, and therefore we chose to implement the classic algorithm.

We show pseudocode for our PageRank implementation in Algorithm 19. The initial PageRank values are set to $1/n$ (Line 1). The algorithm initializes a frontier containing all vertices (Line 5), and sets the error (the l_1 norm between consecutive PageRank vectors) to ∞ (Line 12). The algorithm then iterates the PageRank update step while the error is above the threshold ϵ (Lines 13–16). The update is implemented using the nghReduce primitive (see Section 4 for details on the primitive).

ALGORITHM 19: PageRank

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

The *condFn* function (Line 6) specifies that value should be aggregated for each vertex with non-zero in-degree. The *mapFn* function pulls a PageRank contribution of $P_{curr}[u]/d(u)$ for each in-neighbor u in the frontier (Line 7). Finally, after the contributions to each neighbor have been summed up, the *applyFn* function is called on a pair of a neighboring vertex v, and its contribution (Lines 8–11). The apply step updates the next PageRank value for the vertex using the PageRank equation above (Line 9) and updates the difference in PageRank values for this vertex in the *diffs* vector (Line 10). The last steps in the loop applies a parallel reduction over the differences vector to update the current error (Line 15) and finally swaps the current and next PageRank vectors (Line 16).

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

7 IMPLEMENTATIONS AND TECHNIQUES

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

7.1 A Work-efficient Histogram Implementation

Our initial implementation of the peeling-based algorithm for k-core algorithm suffered from poor performance due to a large amount of contention incurred by fetchAndAdds on high-degree vertices. This issue 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 3). For these graphs, we observed that many early rounds, which process vertices with low coreness perform a large number of fetchAndAdds on memory locations corresponding to high-degree vertices, resulting in high contention [138]. To reduce contention, we designed a work-efficient histogram implementation that can perform this step while only incurring $O(\log n)$ contention *whp*. The ***Histogram*** primitive takes a sequence of $(\mathbf{K}, \mathbf{T})$ pairs, and an associative and commutative operator $R : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$ and computes a sequence of $(\mathbf{K}, \mathbf{T})$ pairs, where each key k only appears once, and its associated value t is the sum of all values associated with keys k in the input, combined with respect to R.

A useful example of histogram to consider is summing for each $v \in N(F)$ for a vertexSubset F, the number of edges (u, v) where $u \in F$ (i.e., the number of incoming neighbors from the frontier). This operation can be implemented by running histogram on a sequence where each $v \in N(F)$ appears once per (u, v) edge as a tuple $(v, 1)$ using the operator +. One theoretically efficient implementation of histogram is to simply semisort the pairs using the work-efficient semisort algorithm from [74]. The semisort places pairs from the sequence into a set of *heavy* and *light* buckets, where heavy buckets contain a single key that appears many times in the input sequence, and light buckets contain at most $O(\log^2 n)$ distinct keys (k, v) keys, each of which appear at most $O(\log n)$ times *whp* (heavy and light keys are determined by sampling). We compute the reduced value for heavy buckets using a standard parallel reduction. For each light bucket, we allocate a hash table, and hash the keys in the bucket in parallel to the table, combining multiple values for the same key using R. As each key appears at most $O(\log n)$ times *whp* we incur at most $O(\log n)$ contention *whp*. The output sequence can be computed by compacting the light tables and heavy arrays.

While the semisort implementation is theoretically efficient, it requires a likely cache miss for each key when inserting into the appropriate hash table. To improve cache performance in this step, we implemented a work-efficient algorithm with $O(n^\epsilon)$ depth based on radix sort. Our implementation is based on the parallel radix sort from PBBS [139]. As in the semisort, we first sample keys from the sequence and determine the set of heavy-keys. Instead of directly moving the elements into light and heavy buckets, we break up the input sequence into $O(n^{1-\epsilon})$ blocks, each of size $O(n^\epsilon)$, and sequentially sort the keys within a block into light and heavy buckets. Within the blocks, we reduce all heavy keys into a single value and compute an array of size $O(n^\epsilon)$ which holds the starting offset of each bucket within the block. Next, we perform a segmented-scan [26] over the arrays of the $O(n^{1-\epsilon})$ blocks to compute the sizes of the light buckets, and the reduced values for the heavy-buckets, which only contain a single key. Finally, we allocate tables for the light buckets, hash the light keys in parallel over the blocks and compact the light tables and heavy keys into the output array. Each step runs in $O(n)$ work and $O(n^\epsilon)$ depth. Compared to the original semisort implementation, this version incurs fewer cache misses because the light keys per block are already sorted and consecutive keys likely go to the same hash table, which fits in cache. We compared our times in the histogram-based version of k-core and the fetchAndAdd-based version of k-core and saw between a 1.1–3.1x improvement from using the histogram.

7.2 edgeMapBlocked

One of the core primitives used by our algorithms is edgeMap (described in Section 3). 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 $d(u), u \in U$ to compute offsets, allocates an array of size $\sum_{u \in U} d(u)$, and iterates over all $u \in U$ in parallel, writing the ID of the neighbor to the array

ALGORITHM 20: EDGEMAPBLOCKED

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

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 $\sum_{u \in U} d(u)$. However, EDGEMAPSPARSE will always perform $\sum_{u \in U} d(u)$ 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. The output is obtained by applying a prefix-sum over the number of live neighbors per-block, and compacting the block outputs into the output array.

We now describe a theoretically efficient implementation of EDGEMAPBLOCKED (Algorithm 20). As in EDGEMAPSPARSE, we first compute an array of offsets O (Line 2) by prefix summing the degrees of $u \in U$. We process the edges incident to this frontier in blocks of size *bsize*. As we cannot afford to explicitly write out the edges incident to the current frontier to block them, we instead logically assign the edges to blocks. Each block searches for a range of vertices to process with *bsize* edges; the i-th block binary searches the offsets array to find the vertex incident to the start of the $(i \cdot bsize)$-th edge, storing the result into $B[i]$ (Lines 4–5). The vertices that block i must process are therefore between $B[i]$ and $B[i+1]$. We note that multiple blocks can be assigned to process the edges incident to a high-degree vertex. Next, we allocate an intermediate array I of size d_U (Line 6), but do not initialize the memory, and an array A that stores the number of live neighbors found by each block (Line 7). Next, we process the blocks in parallel by sequentially applying F to each edge in the block and compactly writing any live neighbors to $I[i \cdot bsize]$ (Line 9), and write the number of live neighbors to $A[i]$ (Line 10). Finally, we do a prefix sum on A, which gives offsets into an array of size proportional to the number of live neighbors, and copy the live neighbors in parallel to R, the output array (Line 11).

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

7.3 Techniques for Overlapping Searches

In this section, we describe how we compute and update the reachability labels for vertices that are visited in a phase of our SCC algorithm. Recall that each phase performs a graph traversal from the set of active centers on this round, C_A, and computes for each center c, all vertices in the weakly-connected component for the subproblem of c that can be reached by a directed path from it. We store this reachability information as a set of (u, l_i) pairs in a hash-table, which represent the fact that u can be reached by a directed path from c_i (l_i is a unique label for the center c_i, see Algorithm 11). A phase performs two graph traversals from the centers to compute $\mathcal{R}_F$ and $\mathcal{R}_B$, the out-reachability set and in-reachability sets respectively. Each traversal allocates an initial hash table and runs rounds of EDGEMAP until no new label information is added to the table.

The main challenge in implementing one round in the traversal is (1) ensuring that the table has sufficient space to store all pairs that will be added this round, and (2) efficiently iterating over all of the pairs associated with a vertex. We implement (1) by performing a parallel reduce to sum over vertices $u \in F$, the current frontier, the number of neighbors v in the same subproblem, multiplied by the number of distinct labels currently assigned to u. This quantity upper-bounds the number of distinct labels that could be added this round, and although we may overestimate the number of actual additions, we will never run out of space in the table. We update the number of elements stored in the table during concurrent insertions by storing a per-processor count which gets incremented whenever the processor performs a successful insertion. The counts are then summed together at the end of a round and used to update the count of the number of elements in the table.

One simple implementation of (2) is to simply allocate $O(\log n)$ space for every vertex, as one can show that the maximum number of centers that visit any vertex during a phase is at most $O(\log n)$ *whp*. However, this approach will waste a significant amount of space, as most vertices are visited just a few times (a constant number of times per round, in expectation). Instead, our implementation stores (u, l) pairs in the table for visited vertices u, and computes hashes based only on the ID of u. As each vertex is only expected to be visited a constant number of times during a phase, the expected probe length is still a constant. Storing the pairs for a vertex in the same probe-sequence is helpful for two reasons. First, we may incur fewer cache misses than if we had hashed the pairs based on both entries, as multiple pairs for a vertex can fit in the same cache line. Second, storing the pairs for a vertex along the same probe sequence makes it easy to find all pairs associated with a vertex u; the idea is to simply perform linear-probing, reporting all pairs that have u as their key until we hit an empty cell. Our experiments confirm that this technique is practical, and we believe that it may have applications in similar algorithms, such as computing least-element lists and FRT trees in parallel [33, 34].

7.4 Primitives on Compressed Graphs

Most of the algorithms studied in this paper are concisely expressed using fundamental primitives such as map, map-reduce, filter, pack, and intersection (see Section 4). To run our algorithms without any modifications on compressed graphs, we wrote new implementations of these primitives using using the parallel-byte format from Ligra+, some of which required some new techniques in order to be theoretically efficient. We first review the byte and parallel-byte formats from [141]. In byte coding, we store a vertex's neighbor list by difference encoding consecutive vertices, with the first vertex difference encoded with respect to the source. Decoding is done by sequentially uncompressing each difference, and summing the differences into a running sum which gives the ID of the next neighbor. As this process is sequential, graph algorithms using the byte format that map over the neighbors of a vertex will require $\Omega(\Delta)$ depth, where Δ is the maximum degree of a vertex in the graph. The parallel-byte format from Ligra+ breaks the neighbors of a high-degree

vertex into blocks, where each block contains a fixed number of neighbors. Each block is difference encoded with respect to the source. As each block can have a different compressed size, it also stores offsets that point to the start of each block. The format stores the blocks in a neighbor list L in sorted order.

We now describe efficient implementations of primitives used by our algorithms. All descriptions are given for neighbor lists coded in the parallel-byte format, and we assume for simplicity that the block size (the number of neighbors stored in each block) is $O(\log n)$. The ***Map*** primitive takes as input neighbor list L, and a map function F, and applies F to each ID in L. This primitive can be implemented with a parallel-for loop across the blocks, where each iteration decodes its block sequentially. Our implementation of map runs in $O(|L|)$ work and $O(\log n)$ depth. ***Map-Reduce*** takes as input a neighbor list L, a map function $F : \mathbf{vtx} \rightarrow \mathbf{T}$ and a binary associative function R and returns the sum of the mapped elements with respect to R. We perform map-reduce similarly by first mapping over the blocks, then sequentially reducing over the mapped values in each block. We store the accumulated value on the stack or in an heap-allocated array if the number of blocks is large enough. Finally, we reduce the accumulated values using R to compute the output. Our implementation of map-reduce runs in $O(|L|)$ work and $O(\log n)$ depth.

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

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

8 EXPERIMENTS

In this section, we describe our experimental results on a set of real-world graphs and also discuss related experimental work. Tables 5 and 6 show the running times for our implementations on

Table 3. Graph Inputs, Including Vertices and Edges

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

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

our graph inputs. For compressed graphs, we use the compression schemes from Ligra+ [141], which we extended to ensure theoretical efficiency (see Section 7.4). We describe statistics about our input graphs and algorithms (e.g., number of colors used, number of SCCs, etc.) in Section A.

8.1 Experimental Setup and Graph Inputs

Experimental Setup. We ran all of our experiments on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4 × 2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use Cilk Plus to express parallelism and are compiled with the `g++` compiler (version 5.4.1) with the `-O3` flag. By using Cilk's work-stealing scheduler we are able obtain an expected running time of $W/P + O(D)$ for an algorithm with W work and D depth on P processors [38]. We note that our codes can also be easily run using other parallel runtimes, such as OpenMP, TBB, or a homegrown scheduler based on the Arora-Blumofe-Plaxton deque [10] that we implemented ourselves [27]. For the parallel experiments, we use the command `numactl -i all` to balance the memory allocations across the sockets. All of the speedup numbers we report are the running times of our parallel implementation on 72-cores with hyper-threading over the running time of the implementation on a single thread.

Graph Data. To show how our algorithms perform on graphs at different scales, we selected a representative set of real-world graphs of varying sizes. Most of the graphs are Web graphs and social networks—low diameter graphs that are frequently used in practice. To test our algorithms on large diameter graphs, we also ran our implementations on 3-dimensional tori where each vertex is connected to its 2 neighbors in each dimension.

We list the graphs used in our experiments, along with their size, approximate diameter, peeling complexity [52], and degeneracy (for undirected graphs) in Table 3. ***LiveJournal*** is a directed graph of the social network obtained from a snapshot in 2008 [39]. ***com-Orkut*** is an undirected graph of the Orkut social network. ***Twitter*** is a directed graph of the Twitter network, where edges represent the follower relationship [92]. ***ClueWeb*** is a Web graph from the Lemur project at CMU [39]. ***Hyperlink2012*** and ***Hyperlink2014*** are directed hyperlink graphs obtained from the

Table 4. Compressed Graph Inputs, Including Memory Required to Store the Graph in an Uncompressed CSR Format, Memory Required to Store the Graph in the Parallel Byte-compressed CSR Format, and the Savings Obtained Over the Uncompressed Format by the Compressed Format

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

The number of vertices and edges in these graphs are given in Table 3.

WebDataCommons dataset where nodes represent web pages [107]. ***3D-Torus*** is a 3-dimensional torus with 1B vertices and 6B edges. We mark symmetric (undirected) versions of the directed graphs with the suffix -Sym. We create weighted graphs for evaluating weighted BFS, Borůvka, widest path, and Bellman-Ford by selecting edge weights between $[1, \log n)$ uniformly at random. We process LiveJournal, com-Orkut, Twitter, and 3D-Torus in the uncompressed format, and ClueWeb, Hyperlink2014, and Hyperlink2012 in the compressed format.

Table 4 lists the size in gigabytes of the compressed graph inputs used in this paper both with and without compression, and reports the savings obtained by using compression. Note that the largest graph studied in this paper, the directed Hyperlink2012 graph, barely fits in the main memory of our machine in the uncompressed format, but would leave hardly any memory to be used for an algorithm analyzing this graph. Using compression significantly reduces the memory required to represent each graph (between 2.21–2.85x, and 2.53x on average). We converted the graphs listed in Table 4 directly from the WebGraph format to the compressed format used in this paper by modifying a sequential iterator method from the WebGraph framework [39].

8.2 SSSP Problems

Our BFS, weighted BFS, Bellman-Ford, and betweenness centrality implementations achieve between a 13–67x speedup across all inputs. We ran all of our shortest path experiments on the *symmetrized* versions of the graph. Our widest path implementation achieves between 38–72x speedup across all inputs, and our spanner implementation achieves between 31–65x speedup across all inputs. We ran our spanner code with $k = 4$. Our experiments show that our weighted BFS and Bellman-Ford implementations perform as well as or better than our prior implementations from Julienne [52]. Our running times for BFS and betweenness centrality are the same as the times of the implementations in Ligra [136]. We note that our running times for weighted BFS on the Hyperlink graphs are larger than the times reported in Julienne. This is because the shortest-path experiments in Julienne were run on directed version of the graph, where the average vertex can reach significantly fewer vertices than on the symmetrized version. We set a flag for our weighted BFS experiments on the ClueWeb and Hyperlink graphs that lets the algorithm switch to a dense edgeMap once the frontiers are sufficiently dense, which lets the algorithm run within half of the RAM on our machine. Before this change, our weighted BFS implementation would request a large amount of memory when processing the largest frontiers which then caused the graph to become partly evicted from the page cache. For widest path, the times we report are for the Bellman-Ford version of the algorithm, which we were surprised to find is consistently 1.1–1.3x faster than our

Table 5. Running Times (in seconds) of Our Algorithms Over Symmetric Graph Inputs on a 72-core Machine (with Hyper-threading) Where (1) is the Single-thread Time, (72h) is the 72 Core Time Using Hyper-threading, and (SU) is the Parallel Speedup (Single-thread Time Divided by 72-core Time)

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

We mark experiments that are not applicable for a graph with ~, and experiments that did not finish within 5 hours with —. *SCC was run on the directed versions of the input graphs.

algorithm based on bucketing. We observe that our spanner algorithm is only slightly more costly than computing connectivity on the same input.

In an earlier paper [52], we compared the running time of our weighted BFS implementation to two existing parallel shortest path implementations from the GAP benchmark suite [22] and Galois [100], as well as a fast sequential shortest path algorithm from the DIMACS shortest path challenge, showing that our implementation is between 1.07–1.1x slower than the Δ-stepping implementation from GAP, and 1.6–3.4x faster than the Galois implementation. Our old version of Bellman-Ford was between 1.2–3.9x slower than weighted BFS; we note that after changing it to use the EDGEMAPBLOCKED optimization, it is now competitive with weighted BFS and is between 1.2x faster and 1.7x slower on our graphs with the exception of 3D-Torus, where it performs 7.3x slower than weighted BFS, as it performs $O(n^{4/3})$ work on this graph.

8.3 Connectivity Problems

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

Our work-efficient implementation of connectivity and spanning forest achieve 34–65x speedup and 31–67x speedup across all inputs, respectively. We note that our implementation does not

Table 6. Running Times (in seconds) of Our Algorithms Over Symmetric Graph Inputs on a 72-core Machine (with Hyper-threading) Where (1) is the Single-thread Time, (72h) is the 72 Core Time Using Hyper-threading, and (SU) is the Parallel Speedup (Single-thread Time Divided by 72-core Time)

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

We mark experiments that are not applicable for a graph with ~, and experiments that did not finish within 5 hours with —. *SCC was run on the directed versions of the input graphs.

assume that vertex IDs in the graph are randomly permuted and always generates a random permutation, even on the first round, as adding vertices based on their original IDs can result in poor performance (for example on 3D-Torus). There are several existing implementations of fast parallel connectivity algorithms [119, 139, 140, 144], however, only the implementation from [140], which presents the connectivity algorithm that we implement in this paper, is theoretically-efficient. The implementation from Shun et al. was compared to both the Multistep [144] and Patwary et al. [119] implementations, and shown to be competitive on a broad set of graphs. We compared our connectivity implementation to the work-efficient connectivity implementation from Shun et al. on our uncompressed graphs and observed that our code is between 1.2–2.1x faster in parallel. Our spanning forest implementation is slightly slower than connectivity due to having to maintain a mapping between the current edge set and the original edge set.

Despite our biconnectivity implementation having $O(\mathsf{diam}(G)\log n)$ depth, our implementation achieves between a 20–59x speedup across all inputs, as the diameter of most of our graphs is extremely low. Our biconnectivity implementation is about 3–5x slower than running connectivity on the graph, which seems reasonable as our current implementation performs two calls to connectivity, and one breadth-first search. There are a several existing implementations of biconnectivity. Cong and Bader [46] parallelize the Tarjan-Vishkin algorithm and demonstrated speedup over the Hopcroft-Tarjan (HT) algorithm. Edwards and Vishkin [61] also implement the Tarjan-Vishkin algorithm using the XMT platform, and show that their algorithm achieves good speedups. Slota and

Madduri [143] present a BFS-based biconnectivity implementation which requires $O(mn)$ work in the worst-case, but behaves like a linear-work algorithm in practice. We ran the Slota and Madduri implementation on 36 hyper-threads allocated from the same socket, the configuration on which we observed the best performance for their code, and found that our implementation is between 1.4–2.1x faster than theirs. We used a DFS-ordered subgraph corresponding to the largest connected component to test their code, which produced the fastest times. Using the original order of the graph affects the running time of their implementation, causing it to run between 2–3x slower as the amount of work performed by their algorithm depends on the order in which vertices are visited.

Our strongly connected components implementation achieves between a 13–43x speedup across all inputs. Our implementation takes a parameter β, which is the base of the exponential rate at which we grow the number of centers added. We set β between 1.1–2.0 for our experiments and note that using a larger value of β can improve the running time on smaller graphs by up to a factor of 2x. Our SCC implementation is between 1.6x faster to 4.8x slower than running connectivity on the undirected version of the graph. There are several existing SCC implementations that have been evaluated on real-world directed graphs [79, 106, 144]. The Hong et al. algorithm [79] is a modified version of the FWBW-Trim algorithm from McLendon et al. [106], but neither algorithm has any theoretical bounds on work or depth. Unfortunately [79] do not report running times, so we are unable to compare our performance with them. The Multistep algorithm [144] has a worst-case running time of $O(n^2)$, but the authors point-out that the algorithm behaves like a linear-time algorithm on real-world graphs. We ran our implementation on 16 cores configured similarly to their experiments and found that we are about 1.7x slower on LiveJournal, which easily fits in cache, and 1.2x faster on Twitter (scaled to account for a small difference in graph sizes). While the multistep algorithm is slightly faster on some graphs, our SCC implementation has the advantage of being theoretically-efficient and performs a predictable amount of work.

Our minimum spanning forest implementation achieves between 17–54x speedup over the implementation running on a single thread across all of our inputs. Obtaining practical parallel algorithms for MSF has been a longstanding goal in the field, and several existing implementations exist [14, 47, 116, 139, 155]. We compared our implementation with the union-find based MSF implementation from PBBS [139] and the implementation of Borůvka from [155], which is one of the fastest implementations we are aware of. Our MSF implementation is between 2.6–5.9x faster than the MSF implementation from PBBS. Compared to the edgelist based implementation of Borůvka from [155] our implementation is between 1.2–2.9x faster.

8.4 Covering Problems

Our MIS and maximal matching implementations achieve between 31–70x and 25–66x speedup across all inputs. The implementations by Blelloch et al. [32] are the fastest existing implementations of MIS and maximal matching that we are aware of, and are the basis for our maximal matching implementation. They report that their implementations are 3–8x faster than Luby's algorithm on 32 threads, and outperform a sequential greedy MIS implementation on more than 2 processors. We compared our rootset-based MIS implementation to the prefix-based implementation, and found that the rootset-based approach is between 1.1–3.5x faster. Our maximal matching implementation is between 3–4.2x faster than the implementation from [32]. Our implementation of maximal matching can avoid a significant amount of work, as each of the filter steps can extract and permute just the $3n/2$ highest priority edges, whereas the edgelist-based version in PBBS must permute all edges. Our coloring implementation achieves between 11–56x speedup across all inputs. We note that our implementation appears to be between 1.2–1.6x slower than the

asynchronous implementation of JP in [77], due to synchronizing on many rounds which contain few vertices.

Our approximate set cover implementation achieves between 5–58x speedup across all inputs. Our implementation is based on the implementation presented in Julienne [52]; the one major modification was to regenerate random priorities for sets that are active on the current round. We compared the running time of our implementation with the parallel implementation from [37] which is available in the PBBS library. We ran both implementations with $\epsilon = 0.01$. Our implementation is between 1.2x slower to 1.5x faster than the PBBS implementation on our graphs, with the exception of 3D-Torus. On 3D-Torus, the implementation from [37] runs 56x slower than our implementation as it does not regenerate priorities for active sets on each round causing worst-case behavior. Our performance is also slow on this graph, as nearly all of the vertices stay active (in the highest bucket) during each round, and using $\epsilon = 0.01$ causes a large number of rounds to be performed.

8.5 Substructure Problems

Our k-core implementation achieves between 5–46x speedup across all inputs, and 114x speedup on the 3D-Torus graph as there is only one round of peeling in which all vertices are removed. There are several recent papers that implement parallel algorithms for k-core [50, 52, 86, 128]. Both the ParK algorithm [50] and Kabir and Madduri algorithm [86] implement the peeling algorithm in $O(k_{\max}n + m)$ work, which is not work-efficient. Our implementation is between 3.8–4.6x faster than ParK on a similar machine configuration. Kabir and Madduri show that their implementation achieves an average speedup of 2.8x over ParK. Our implementation is between 1.3–1.6x faster than theirs on a similar machine configuration.

Our approximate densest subgraph implementation achieves between 55–71x speedup across all inputs. We ran our implementation with $\epsilon = 0.001$, which in our experiments produced subgraphs with density roughly equal to those produced by the 2-approximation algorithm based on degeneracy ordering, or setting ϵ to 0. To the best of our knowledge, there are no prior existing shared-memory parallel algorithms for this problem.

Our triangle counting (TC) implementation achieves between 39–81x speedup across all inputs. Unfortunately, we are unable to report speedup numbers for TC on our larger graphs as the single-threaded times took too long due to the algorithm performing $O(m^{3/2})$ work. There are a number of experimental papers that consider multicore triangle counting [1, 72, 90, 97, 133, 142]. We implement the algorithm from [142], and adapted it to work on compressed graphs. We note that in our experiments we intersect directed adjacency lists sequentially, as there was sufficient parallelism in the outer parallel-loop. There was no significant difference in running times between our implementation and the implementation from [142]. We ran our implementation on 48 threads on the Twitter graph to compare with the times reported by EmptyHeaded [1] and found that our times are about the same.

8.6 Eigenvector Problems

Our PageRank (PR) implementation achieves between 47–74x speedup across all inputs. We note that the running times we report are for a single iteration of PageRank. Our implementation is based on the implementation from Ligra [136], and uses a damping factor $\gamma = 0.85$. We note that the modification made to carry out dense iterations using a reduction over the in-neighbors of a vertex was important to decrease contention and improve parallelism, and provided between 2–3x speedup over the Ligra implementation in practice. Many graph processing systems implement PageRank. The optimizing compiler used by GraphIt generates a highly-optimized implementation that is currently the fastest shared-memory implementation known to us [153]. We note that

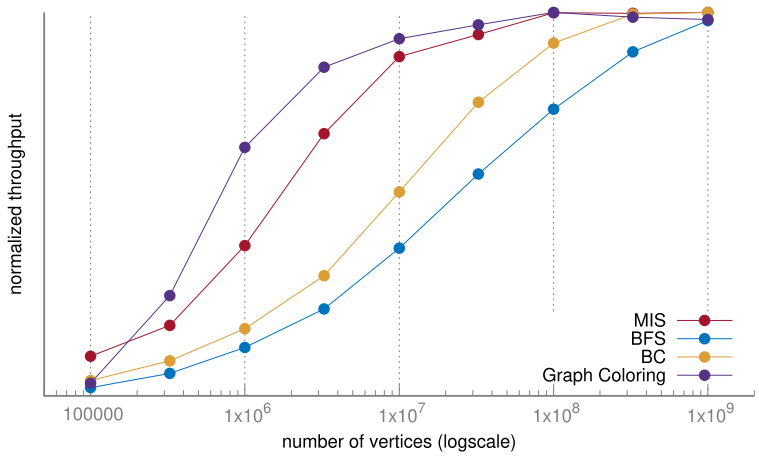

Fig. 7. Log-linear plot of normalized throughput vs. vertices for MIS, BFS, BC, and coloring on the 3D-Torus graph family.

our implementation is about 1.8x slower than the implementation in GraphIt for LiveJournal and Twitter when run on the same number of threads as in their experiments, which is likely due to a partitioning optimization used by GraphIt that eliminates a large amount of cross-socket traffic and thus improves performance on multi-socket systems.

8.7 Performance on 3D-Torus

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

In Figure 7 we show the normalized throughput of MIS, BFS, BC, and graph coloring for 3-dimensional tori of different sizes, where throughput is measured as the number of edges processed per second. The throughput for each application becomes saturated before our largest-scale graph for all applications except for BFS, which is saturated on a graph with 2 billion vertices. The throughput curves show that the theoretical bounds are useful in predicting how the half-lengths[7] are distributed. The half-lengths are ordered as follows: coloring, MIS, BFS, and BC. This is the same order as sorting these algorithms by their depth with respect to this graph.

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

Table 7. Cycles Stalled While the Memory Subsystem has an Outstanding Load (Trillions), LLC Hit Rate and Misses (Billions), Bandwidth in GB/s (Bytes Read and Written from Memory, Divided by Running Time), and Running Time in Seconds

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

All experiments are run on the ClueWeb graph using 72 cores with hyper-threading.

8.8 Locality

While our algorithms are efficient in the specific variants of the binary-forking model that we consider, we do not analyze their cache complexity, and in general they may not be efficient in a model that takes caches into account. Despite this fact, we observed that our algorithms have good cache performance on the graphs we tested on. In this section we give some explanation for this fact by showing that our primitives make good use of the caches. Our algorithms are also aided by the fact that these graph datasets often come in highly local orders (e.g., see the *Natural* order in [56]).

We ran a set of experiments to study the locality of a subset of our algorithms on the ClueWeb graph. Table 7 shows locality metrics for our experiments, which we measured using the Open Performance Counter Monitor (PCM). We found that using a work-efficient histogram is 3.5x faster than using fetchAndAdd in our k-core implementation, which suffers from high contention on this graph. Using a histogram reduces the number of cycles stalled due to memory by more than 7x. We also ran our wBFS implementation with and without the edgeMapBlocked optimization, which reduces the number of cache-lines read from and written to when performing a sparse edgeMap. The blocked implementation reads and writes 2.1x fewer bytes than the unoptimized version, which translates to a 1.7x faster running time. We note that we disabled the dense optimization for this experiment to directly compare the two implementations of a sparse edgeMap.

8.9 Processing Massive Web Graphs

In Table 6, we show the running times of our implementations on the ClueWeb, Hyperlink2014, and Hyperlink2012 graphs. To put our performance on these massive graphs in context, we compare our 72-core running times to running times reported by existing work. Table 8 summarizes state-of-the-art existing results in the literature. Most results process the *directed* versions of these graphs, which have about half as many edges as the symmetrized version. Unless otherwise mentioned, all results from the literature use the directed versions of these graphs. To make the comparison easier we show our running times for BFS, SSSP (weighted BFS), BC and SCC on the directed graphs, and running times for Connectivity, k-core and TC on the symmetrized graphs in Table 8.

FlashGraph [154] 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 [99] 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

Table 8. System Configurations (Memory in Terabytes, Hyper-threads, and Nodes) and Running Times (in seconds) of Existing Results on the Hyperlink Graphs

Paper	Problem	Graph	Memory	Hyper-threads	Nodes	Time
Mosaic [99]	BFS*	2014	0.768	1000	1	6.55
	Connectivity*	2014	0.768	1000	1	708
	SSSP*	2014	0.768	1000	1	8.6
FlashGraph [154]	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 [85]	BFS*	2012	0.064	32	1	900
	BC*	2012	0.064	32	1	800
Slota et al. [146]	Largest-CC*	2012	16.3	8192	256	63
	Largest-SCC*	2012	16.3	8192	256	108
	Approx k-core*	2012	16.3	8192	256	363
Stergiou et al. [147]	Connectivity	2012	128	24000	1000	341
Gluon [51]	BFS	2012	24	69632	256	380
	Connectivity	2012	24	69632	256	75.3
	PageRank	2012	24	69632	256	158.2
	SSSP	2012	24	69632	256	574.9
This paper	BFS*	2014	1	144	1	5.71
	SSSP*	2014	1	144	1	9.08
	Connectivity	2014	1	144	1	11.2
	BFS*	2012	1	144	1	16.7
	BC*	2012	1	144	1	35.2
	Connectivity	2012	1	144	1	25.0
	SCC*	2012	1	144	1	185
	SSSP	2012	1	144	1	58.1
	k-core	2012	1	144	1	184
	PageRank	2012	1	144	1	462
	TC	2012	1	144	1	1168

The last section shows our running times. *These problems are run on directed versions of the graph.

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 [85] 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. [146] 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. [51] 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 incomparable.

Stergiou et al. [147] describe a connectivity algorithm that runs in $O(\log n)$ rounds in the BSP model and report running times for the Hyperlink2012-Sym 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.

9 CONCLUSION AND FUTURE WORK

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

Future Work. There are many directions for future work stemming from this work. One is to continue to extend GBBS with other graph problems that were not considered in this paper. For example, the recent work of Shi et al. [134] extends GBBS with work-efficient clique-counting algorithms and work-efficient algorithms for low out-degree orientation. It would be interesting to include parallel implementations for other classic graph problems as part of GBBS, such as planarity testing and embedding, planar separator, higher connectivity, among many others.

It would also be interesting to study practical implementations for dynamic graph problems in the parallel batch-dynamic setting. Recent work has proposed theoretically-efficient parallel batch-dynamic algorithms for many fundamental problems such as dynamic connectivity [2, 55] and dynamic k-clique counting [57], among other problems. It would be interesting to study the

practicality of these algorithms using an efficient parallel batch-dynamic data structure for dynamic graphs, such as Aspen [54], and to include these problems as part of GBBS.

Another direction is to extend GBBS to important application domains of graph algorithms, such as graph clustering. Although clustering is quite different from the problems studied in this paper since there is usually no single "correct" way to cluster a graph or point set, we believe that our approach will be useful for building theoretically-efficient and scalable single-machine clustering algorithms, including density-based clustering [62], affinity clustering [20], and hierarchical agglomerative clustering (HAC) on graphs [102]. The recent work of Tseng et al. [150] presents a work-efficient parallel structural graph clustering algorithm which is incorporated into GBBS.

Lastly, it would be interesting to understand the portability of our implementations in different architectures and computational settings. Recent work in this direction has found that implementations developed in this paper can be efficiently implemented in a setting where the graph is stored in NVRAM, and algorithms have access to a limited amount of DRAM [29, 58]. The experimental results for their NVRAM system, called *Sage*, shows that applying the implementations from this paper in conjunction with an optimized EDGEMAP primitive designed for NVRAMs achieves superior performance on an NVRAM-based machine compared to the state-of-the-art NVRAM implementations of Gill et al. [69], providing promising evidence for the portability of our approach.

APPENDIX

A GRAPH STATISTICS

In this section, we list graph statistics computed for the graphs from Section 8.[8] These statistics include the number of connected components, strongly connected components, colors used by the LLF and LF heuristics, number of triangles, and several others. These numbers will be useful for verifying the correctness or quality of our algorithms in relation to future algorithms that also run on these graphs. Although some of these numbers were present in Table 3, we include in the tables below for completeness. We provide details about the statistics that are not self-explanatory.

- *Effective Directed Diameter*: the maximum number of levels traversed during a graph traversal algorithm (BFS or SCC) on the unweighted directed graph.
- *Effective Undirected Diameter*: the maximum number of levels traversed during a graph traversal algorithm (BFS) on the unweighted directed graph.
- *Size of Largest (Connected/Biconnected/Strongly-Connected) Component*: The number of *vertices* in the largest (connected/biconnected/strongly-connected) component. Note that in the case of biconnectivity, we assign labels to edges, so a vertex participates in a component for each distinct edge label incident to it.
- *Num. Triangles*: The number of closed triangles in G, where each triangle (u, v, w) is counted exactly once.
- *Num. Colors Used by (LF/LLF)*: The number of colors used is just the maximum color ID assigned to any vertex.
- *(Maximum Independent Set/Maximum Matching/Approximate Set Cover) Size*: We report the sizes of these objects computed by our implementations. For MIS and maximum matching we report this metric to lower-bound the size of the maximum independent set and maximum matching supported by the graph. For approximate set cover, we run our code on instances similar to those used in prior work (e.g., Blelloch et al. [37] and Dhulipala et al. [52]) where the elements are vertices and the sets are the neighbors of each vertex

[8] Similar statistics can be found on the SNAP website (https://snap.stanford.edu/data/) and the Laboratory for Web Algorithmics website (http://law.di.unimi.it/datasets.php).

Table 9. Graph Statistics for the LiveJournal Graph

Statistic	Value
Num. Vertices	4,847,571
Num. Directed Edges	68,993,773
Num. Undirected Edges	85,702,474
Effective Directed Diameter	16
Effective Undirected Diameter	20
Num. Connected Components	1,876
Num. Biconnected Components	1,133,883
Num. Strongly Connected Components	971,232
Size of Largest Connected Component	4,843,953
Size of Largest Biconnected Component	3,665,291
Size of Largest Strongly Connected Component	3,828,682
Num. Triangles	285,730,264
Num. Colors Used by LF	323
Num. Colors Used by LLF	327
Maximal Independent Set Size	2,316,617
Maximal Matching Size	1,546,833
Set Cover Size	964,492
$k_{\max}$ (Degeneracy)	372
ρ (Num. Peeling Rounds in k-core)	3,480

Table 10. Graph Statistics for the Com-Orkut Graph

Statistic	Value
Num. Vertices	3,072,627
Num. Directed Edges	—
Num. Undirected Edges	234,370,166
Effective Directed Diameter	—
Effective Undirected Diameter	9
Num. Connected Components	187
Num. Biconnected Components	68,117
Num. Strongly Connected Components	—
Size of Largest Connected Component	3,072,441
Size of Largest Biconnected Component	3,003,914
Size of Largest Strongly Connected Component	—
Num. Triangles	627,584,181
Num. Colors Used by LF	86
Num. Colors Used by LLF	98
Maximal Independent Set Size	651,901
Maximal Matching Size	1,325,427
Set Cover Size	105,572
$k_{\max}$ (Degeneracy)	253
ρ (Num. Peeling Rounds in k-core)	5,667

As com-Orkut is an undirected graph, some of the statistics are not applicable and we mark the corresponding values with –.

Table 11. Graph Statistics for the Twitter Graph

Statistic	Value
Num. Vertices	41,652,231
Num. Directed Edges	1,468,365,182
Num. Undirected Edges	2,405,026,092
Effective Directed Diameter	65
Effective Undirected Diameter	23
Num. Connected Components	2
Num. Biconnected Components	1,936,001
Num. Strongly Connected Components	8,044,729
Size of Largest Connected Component	41,652,230
Size of Largest Biconnected Component	39,708,003
Size of Largest Strongly Connected Component	33,479,734
Num. Triangles	34,824,916,864
Num. Colors Used by LF	1,081
Num. Colors Used by LLF	1,074
Maximal Independent Set Size	26,564,540
Maximal Matching Size	9,612,260
Set Cover Size	1,736,761
$k_{\max}$ (Degeneracy)	2,488
ρ (Num. Peeling Rounds in k-core)	14,963

Table 12. Graph Statistics for the ClueWeb Graph

Statistic	Value
Num. Vertices	978,408,098
Num. Directed Edges	42,574,107,469
Num. Undirected Edges	74,774,358,622
Effective Directed Diameter	821
Effective Undirected Diameter	132
Num. Connected Components	23,794,336
Num. Biconnected Components	81,809,602
Num. Strongly Connected Components	135,223,661
Size of Largest Connected Component	950,577,812
Size of Largest Biconnected Component	846,117,956
Size of Largest Strongly Connected Component	774,373,029
Num. Triangles	1,995,295,290,765
Num. Colors Used by LF	4,245
Num. Colors Used by LLF	4,245
Maximal Independent Set Size	459,052,906
Maximal Matching Size	311,153,771
Set Cover Size	64,322,081
$k_{\max}$ (Degeneracy)	4,244
ρ (Num. Peeling Rounds in k-core)	106,819

Table 13. Graph Statistics for the Hyperlink2014 Graph

Statistic	Value
Num. Vertices	1,724,573,718
Num. Directed Edges	64,422,807,961
Num. Undirected Edges	124,141,874,032
Effective Directed Diameter	793
Effective Undirected Diameter	207
Num. Connected Components	129,441,050
Num. Biconnected Components	132,198,693
Num. Strongly Connected Components	1,290,550,195
Size of Largest Connected Component	1,574,786,584
Size of Largest Biconnected Component	1,435,626,698
Size of Largest Strongly Connected Component	320,754,363
Num. Triangles	4,587,563,913,535
Num. Colors Used by LF	4154
Num. Colors Used by LLF	4158
Maximal Independent Set Size	1,333,026,057
Maximal Matching Size	242,469,131
Set Cover Size	23,869,788
$k_{\max}$ (Degeneracy)	4,160
ρ (Num. Peeling Rounds in k-core)	58,711

Table 14. Graph Statistics for the Hyperlink2012 Graph

Statistic	Value
Num. Vertices	3,563,602,789
Num. Directed Edges	128,736,914,167
Num. Undirected Edges	225,840,663,232
Effective Directed Diameter	5275
Effective Undirected Diameter	331
Num. Connected Components	144,628,744
Num. Biconnected Components	298,663,966
Num. Strongly Connected Components	1,279,696,892
Size of Largest Connected Component	3,355,386,234
Size of Largest Biconnected Component	3,023,064,231
Size of Largest Strongly Connected Component	1,827,543,757
Num. Triangles	9,648,842,110,027
Num. Colors Used by LF	10,566
Num. Colors Used by LLF	10,566
Maximal Independent Set Size	1,799,823,993
Maximal Matching Size	2,434,644,438
Set Cover Size	372,668,619
$k_{\max}$ (Degeneracy)	10,565
ρ (Num. Peeling Rounds in k-core)	130,728

in the undirected graph. In the case of the social network and hyperlink graphs, this optimization problem naturally captures the minimum number of users or Web pages whose neighborhoods must be retrieved to cover the entire graph.

- k_{max} *(Degeneracy)*: The value of k of the largest non-empty k-core.

ACKNOWLEDGMENTS

Thanks to Jessica Shi and Tom Tseng for their work on GBBS and parts of this paper, and thanks to the reviewers and Lin Ma for helpful comments. This research was supported in part by NSF grants #CCF-1408940, #CCF-1533858, #CCF-1629444, and #CCF-1845763, DOE grant #DE-SC0018947, and a Google Faculty Research Award.

REFERENCES

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A relational engine for graph processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44.

[2] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. 2019. 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*. 381–392.

[3] Alok Aggarwal, Richard J. Anderson, and M.-Y. Kao. 1989. Parallel depth-first search in general directed graphs. In *ACM Symposium on Theory of Computing (STOC)*. 297–308.

[4] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *IEEE International Symposium on Workload Characterization, IISWC*. 44–55.

[5] Noga Alon, László Babai, and Alon Itai. 1986. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms* 7, 4 (1986), 567–583.

[6] N. Alon, R. Yuster, and U. Zwick. 1997. Finding and counting given length cycles. *Algorithmica* 17, 3 (1997), 209–223.

[7] Richard Anderson and Ernst W. Mayr. 1984. *A* P*-complete Problem and Approximations to It.* Technical Report.

[8] Alexandr Andoni, Clifford Stein, and Peilin Zhong. 2020. Parallel approximate undirected shortest paths via low hop emulators. In *ACM Symposium on Theory of Computing (STOC)*. ACM, 322–335.

[9] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (01 Apr 2001).

[10] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (2001), 115–144.

[11] Baruch Awerbuch. 1985. Complexity of network synchronization. *J. ACM* 32, 4 (1985), 804–823.

[12] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. 1992. Low-diameter graph decomposition is in NC. In *Scandinavian Workshop on Algorithm Theory*. 83–93.

[13] Baruch Awerbuch and Y. Shiloach. 1983. New connectivity and MSF algorithms for ultracomputer and PRAM. In *International Conference on Parallel Processing (ICPP)*. 175–179.

[14] David A. Bader and Guojing Cong. 2006. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J. Parallel Distrib. Comput.* 66, 11 (2006), 1366–1378.

[15] David A. Bader and Kamesh Madduri. 2005. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *IEEE International Conference on High-Performance Computing (HiPC)*. 465–476.

[16] David A. Bader and Kamesh Madduri. 2006. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *International Conference on Parallel Processing (ICPP)*. 523–530.

[17] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. 2014. Efficient primal-dual graph algorithms for MapReduce. In *International Workshop on Algorithms and Models for the Web-Graph*. 59–78.

[18] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest subgraph in streaming and MapReduce. *Proc. VLDB Endow.* 5, 5 (2012), 454–465.

[19] Georg Baier, Ekkehard Köhler, and Martin Skutella. 2005. The k-splittable flow problem. *Algorithmica* 42, 3–4 (2005), 231–248.

[20] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. 2017. Affinity clustering: Hierarchical clustering at scale. In *Advances in Neural Information Processing Systems*. 6864–6874.

[21] Scott Beamer, Krste Asanović, and David Patterson. 2013. Direction-optimizing breadth-first search. *Scientific Programming* 21, 3–4 (2013), 137–148.

[22] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP benchmark suite. *CoRR* abs/1508.03619 (2015).

[23] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 711–722.

[24] Bonnie Berger, John Rompel, and Peter W. Shor. 1994. Efficient NC algorithms for set cover with applications to learning and geometry. *J. Computer and System Sciences* 49, 3 (Dec. 1994), 454–477.

[25] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. 2013. Efficient parallel and external matching. In *European Conference on Parallel Processing (Euro-Par)*. 659–670.

[26] Guy E. Blelloch. 1993. Prefix sums and their applications. In *Synthesis of Parallel Algorithms*, John Reif (Ed.). Morgan Kaufmann.

[27] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

[28] Guy E. Blelloch and Laxman Dhulipala. 2018. Introduction to Parallel Algorithms. http://www.cs.cmu.edu/realworld/slidesS18/parallelChap.pdf. Carnegie Mellon University.

[29] Guy E. Blelloch, Laxman Dhulipala, Phillip B. Gibbons, Yan Gu, Charlie McGuffey, and Julian Shun. 2021. The read-only semi-external model. In *SIAM/ACM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 70–84.

[30] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic algorithms can be fast. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 181–192.

[31] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.

[32] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 308–317.

[33] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2016. Parallelism in randomized incremental algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 467–478.

[34] Guy E. Blelloch, Yan Gu, and Yihan Sun. 2017. Efficient construction on probabilistic tree embeddings. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*. 26:1–26:14.

[35] Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L Miller, Richard Peng, and Kanat Tangwongsan. 2014. Nearly-linear work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. *Theory of Computing Systems* 55, 3 (2014), 521–554.

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

[37] Guy E. Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Parallel and I/O efficient set covering algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 82–90.

[38] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748.

[39] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph framework I: Compression techniques. In *International World Wide Web Conference (WWW)*. 595–601.

[40] Otakar Borůvka. 1926. O jistém problému minimálním. *Práce Mor. Přírodověd. Spol. v Brně III* 3 (1926), 37–58.

[41] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology* 25, 2 (2001), 163–177.

[42] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. In *International World Wide Web Conference (WWW)*. 107–117.

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

[44] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*. 84–95.

[45] Richard Cole, Philip N. Klein, and Robert E. Tarjan. 1996. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 243–250.

[46] Guojing Cong and David A. Bader. 2005. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 9–18.

[47] Guojing Cong and Ilie Gabriel Tanase. 2016. Composable locality optimizations for accelerating parallel forest computations. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*. 190–197.

[48] Don Coppersmith, Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. 2003. *A Divide-and-conquer Algorithm for Identifying Strongly Connected Components*. Technical Report RC23744. IBM Research.

[49] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3 ed.). MIT Press.
[50] Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. 2014. ParK: An efficient algorithm for k-core decomposition on multicore processors. In *IEEE International Conference on Big Data (BigData)*. 9–16.
[51] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 752–768.
[52] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2017. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 293–304.
[53] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 293–304.
[54] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 918–934.
[55] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. 2020. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1300–1319.
[56] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing graphs and indexes with recursive graph bisection. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 1535–1544.
[57] Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. 2021. Parallel batch-dynamic k-clique counting. In *SIAM/ACM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 129–143.
[58] Laxman Dhulipala, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. 2020. Sage: Parallel semi-asymmetric graph algorithms for NVRAMs. *Proc. VLDB Endow.* 13, 9 (2020), 1598–1613.
[59] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The graph based benchmark suite (GBBS). In *International Workshop on Graph Data Management Experiences and Systems (GRADES) and Network Data Analytics (NDA)*. 11:1–11:8.
[60] Ran Duan, Kaifeng Lyu, and Yuanhang Xie. 2018. Single-source bottleneck path algorithm faster than sorting for sparse graphs. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*. 43:1–43:14.
[61] James A. Edwards and Uzi Vishkin. 2012. Better speedups using simpler parallel programming for graph connectivity and biconnectivity. In *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. 103–114.
[62] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 226–231.
[63] Jeremy T. Fineman. 2018. Nearly work-efficient parallel algorithm for digraph reachability. In *ACM Symposium on Theory of Computing (STOC)*. 457–470.
[64] Manuela Fischer and Andreas Noever. 2018. Tight analysis of parallel randomized greedy MIS. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2152–2160.
[65] Lisa K. Fleischer, Bruce Hendrickson, and Ali Pinar. 2000. On identifying strongly connected components in parallel. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 505–511.
[66] Lester Randolph Ford and Delbert R. Fulkerson. 2009. Maximal flow through a network. In *Classic Papers in Combinatorics*. Springer, 243–248.
[67] Hillel Gazit. 1991. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. on Computing* 20, 6 (Dec. 1991), 1046–1067.
[68] Hillel Gazit and Gary L. Miller. 1988. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Inform. Process. Lett.* 28, 2 (1988), 61–65.
[69] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single machine graph analytics on massive datasets using intel optane DC persistent memory. *Proc. VLDB Endow.* 13, 8 (2020), 1304–13.
[70] A. V. Goldberg. 1984. *Finding a Maximum Density Subgraph*. Technical Report UCB/CSD-84-171. Berkeley, CA, USA.
[71] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 17–30.
[72] Oded Green, Luis M. Munguia, and David A. Bader. 2014. Load balanced clustering coefficients. In *Workshop on Parallel programming for Analytics Applications (PPAA)*. 3–10.
[73] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. 1995. *Limits to Parallel Computation: P-completeness Theory*. Oxford University Press, Inc.

[74] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 24–34.

[75] Shay Halperin and Uri Zwick. 1996. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. *J. Comput. Syst. Sci.* 53, 3 (1996), 395–416.

[76] Shay Halperin and Uri Zwick. 2001. Optimal randomized EREW PRAM algorithms for finding spanning forests. 39, 1 (2001), 1–46.

[77] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2014. Ordering heuristics for parallel graph coloring. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 166–177.

[78] Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, Bozhi You, Keshav Pingali, and Vijaya Ramachandran. 2019. A round-efficient distributed betweenness centrality algorithm. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 272–286.

[79] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 92:1–92:11.

[80] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM* 16, 6 (1973), 372–378.

[81] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1317–1328.

[82] Amos Israeli and Y. Shiloach. 1986. An improved parallel algorithm for maximal matching. *Inform. Process. Lett.* 22, 2 (1986), 57–60.

[83] Alon Itai and Michael Rodeh. 1977. Finding a minimum circuit in a graph. In *ACM Symposium on Theory of Computing (STOC)*. 1–10.

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

[85] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *ACM International Symposium on Computer Architecture (ISCA)*. 411–424.

[86] H. Kabir and K. Madduri. 2017. Parallel k-core decomposition on multicore platforms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1482–1491.

[87] David R. Karger, Philip N. Klein, and Robert E. Tarjan. 1995. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM* 42, 2 (March 1995), 321–328.

[88] Richard M. Karp and Vijaya Ramachandran. 1990. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science (Vol. A)*, Jan van Leeuwen (Ed.). MIT Press, Cambridge, MA, USA, 869–941.

[89] Richard M. Karp and Avi Wigderson. 1984. A fast parallel algorithm for the maximal independent set problem. In *ACM Symposium on Theory of Computing (STOC)*. 266–272.

[90] Jinha Kim, Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, and Hwanjo Yu. 2014. OPT: A new framework for overlapped and parallel triangulation in large-scale graphs. In *ACM International Conference on Management of Data (SIGMOD)*. 637–648.

[91] Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. 2015. Fast greedy algorithms in MapReduce and streaming. *ACM Trans. Parallel Comput.* 2, 3 (2015), 14:1–14:22.

[92] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media? In *International World Wide Web Conference (WWW)*. 591–600.

[93] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.* 407, 1–3 (2008), 458–473.

[94] Charles E. Leiserson and Tao B. Schardl. 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 303–314.

[95] Jason Li. 2020. Faster parallel algorithm for approximate shortest path. In *ACM Symposium on Theory of Computing (STOC)*. 308–321.

[96] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.* 5, 8 (April 2012).

[97] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. 340–349.

[98] Michael Luby. 1986. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.* (1986), 1036–1053.

[99] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *European Conference on Computer Systems (EuroSys)*. 527–543.

[100] Saeed Maleki, Donald Nguyen, Andrew Lenharth, María Garzarán, David Padua, and Keshav Pingali. 2016. DSMR: A parallel algorithm for single-source shortest path problem. In *Proceedings of the 2016 International Conference on Supercomputing (ICS)*. 32:1–32:14.

[101] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *ACM International Conference on Management of Data (SIGMOD)*. 135–146.

[102] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge university press.

[103] Yael Maon, Baruch Schieber, and Uzi Vishkin. 1986. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theoretical Computer Science* 47 (1986), 277–298.

[104] David W. Matula and Leland L. Beck. 1983. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM* 30, 3 (July 1983), 417–427.

[105] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.* 48, 2, Article 25 (Oct. 2015), 39 pages.

[106] William Mclendon Iii, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. 2005. Finding strongly connected components in distributed graphs. *J. Parallel Distrib. Comput.* 65, 8 (2005), 901–910.

[107] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science* 1, 1 (2015), 33–47.

[108] Ulrich Meyer and Peter Sanders. 2000. Parallel shortest path for arbitrary graphs. In *European Conference on Parallel Processing (Euro-Par)*. 461–470.

[109] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms* 49, 1 (2003), 114–152.

[110] Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. 2015. Improved parallel algorithms for spanners and hopsets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 192–201.

[111] Gary L. Miller, Richard Peng, and Shen Chen Xu. 2013. Parallel graph decompositions using random shifts. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 196–203.

[112] Gary L. Miller and Vijaya Ramachandran. 1992. A new graph triconnectivity algorithm and its parallelization. *Combinatorica* 12, 1 (1992), 53–76.

[113] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding graph computing in the context of industrial solutions. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 69:1–69:12.

[114] Mark E. J. Newman. 2003. The structure and function of complex networks. *SIAM Rev.* 45, 2 (2003), 167–256.

[115] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *ACM Symposium on Operating Systems Principles (SOSP)*. 456–471.

[116] Sadegh Nobari, Thanh-Tung Cao, Panagiotis Karras, and Stéphane Bressan. 2012. Scalable parallel minimum spanning forest computation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 205–214.

[117] Mark Ortmann and Ulrik Brandes. 2014. Triangle listing algorithms: Back from the diversion. In *Algorithm Engineering and Experiments (ALENEX)*. 1–8.

[118] Rasmus Pagh and Francesco Silvestri. 2014. The input/output complexity of triangle enumeration. In *ACM Symposium on Principles of Database Systems (PODS)*. 224–233.

[119] M. M. A. Patwary, P. Refsnes, and F. Manne. 2012. Multi-core spanning forest algorithms using the disjoint-set data structure. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 827–835.

[120] David Peleg and Alejandro A Schäffer. 1989. Graph spanners. *Journal of Graph Theory* 13, 1 (1989), 99–116.

[121] Seth Pettie and Vijaya Ramachandran. 2002. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. on Computing* 31, 6 (2002), 1879–1895.

[122] C. A. Phillips. 1989. Parallel graph contraction. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 148–157.

[123] Chung Keung Poon and Vijaya Ramachandran. 1997. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *International Symposium on Algorithms and Computation (ISAAC)*. 212–222.

[124] Sridhar Rajagopalan and Vijay V. Vazirani. 1999. Primal-dual RNC approximation algorithms for set cover and covering integer programs. *SIAM J. on Computing* 28, 2 (Feb. 1999), 525–540.

[125] Vijaya Ramachandran. 1989. A framework for parallel graph algorithm design. In *International Symposium on Optimal Algorithms*. 33–40.

[126] Vijaya Ramachandran. 1993. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In *Synthesis of Parallel Algorithms*, John H Reif (Ed.). Morgan Kaufmann Publishers Inc.
[127] J. Reif. 1985. *Optimal Parallel Algorithms for Integer Sorting and Graph Connectivity.* Technical Report TR-08-85. Harvard University.
[128] Ahmet Erdem Sariyuce, C. Seshadhri, and Ali Pinar. 2018. Parallel local algorithms for core, truss, and nucleus decompositions. *Proc. VLDB Endow.* 12, 1 (2018), 43–56.
[129] T. Schank. 2007. *Algorithmic Aspects of Triangle-Based Network Analysis.* Ph.D. Dissertation. Universitat Karlsruhe.
[130] Thomas Schank and Dorothea Wagner. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *Workshop on Experimental and Efficient Algorithms (WEA).* 606–609.
[131] Warren Schudy. 2008. Finding strongly connected components in parallel using $O(\log^2 N)$ reachability queries. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 146–151.
[132] Stephen B. Seidman. 1983. Network structure and minimum degree. *Soc. Networks* 5, 3 (1983), 269–287.
[133] Martin Sevenich, Sungpack Hong, Adam Welc, and Hassan Chafi. 2014. Fast in-memory triangle listing for large real-world graphs. In *Workshop on Social Network Mining and Analysis.* Article 2, 2:1–2:9 pages.
[134] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2020. Parallel clique counting and peeling algorithms. *arXiv preprint arXiv:2002.10047* (2020).
[135] Yossi Shiloach and Uzi Vishkin. 1982. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms* 3, 1 (1982), 57–67.
[136] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP).* 135–146.
[137] Julian Shun and Guy E. Blelloch. 2014. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 96–107.
[138] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing contention through priority updates. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 299–300.
[139] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: The problem based benchmark suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).*
[140] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2014. A simple and practical linear-work parallel algorithm for connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 143–153.
[141] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *Data Compression Conference (DCC).* 403–412.
[142] Julian Shun and Kanat Tangwongsan. 2015. Multicore triangle computations without tuning. In *IEEE International Conference on Data Engineering (ICDE).* 149–160.
[143] George M. Slota and Kamesh Madduri. 2014. Simple parallel biconnectivity algorithms for multicore platforms. In *IEEE International Conference on High-Performance Computing (HiPC).* 1–10.
[144] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 550–559.
[145] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2015. *Supercomputing for Web Graph Analytics.* Technical Report SAND2015-3087C. Sandia National Lab.
[146] G. M. Slota, S. Rajamanickam, and K. Madduri. 2016. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 293–302.
[147] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting label propagation for distributed connected components. In *International Conference on Web Search and Data Mining (WSDM).* 540–546.
[148] Robert E. Tarjan and Uzi Vishkin. 1985. An efficient parallel biconnectivity algorithm. *SIAM J. on Computing* 14, 4 (1985), 862–874.
[149] Mikkel Thorup and Uri Zwick. 2005. Approximate distance oracles. *J. ACM* 52, 1 (2005), 1–24.
[150] Tom Tseng, Laxman Dhulipala, and Julian Shun. 2021. Parallel index-based structural graph clustering and its approximation. *To appear in ACM International Conference on Management of Data (SIGMOD)* (2021).
[151] Dominic J. A. Welsh and Martin B. Powell. 1967. An upper bound for the chromatic number of a graph and its application to timetabling problems. *Comput. J.* 10, 1 (1967), 85–86.
[152] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big graph analytics platforms. *Foundations and Trends in Databases* 7, 1–2 (2017), 1–195.
[153] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A high-performance graph DSL. *Object-Oriented Programming Systems, Languages,and Applications (OOPSLA)* (2018), 121:1–121:30.

[154] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *USENIX Conference on File and Storage Technologies (FAST)*. 45–58.

[155] Wei Zhou. 2017. *A Practical Scalable Shared-Memory Parallel Algorithm for Computing Minimum Spanning Trees*. Master's thesis. KIT.

Received May 2019; accepted September 2020