MIT Open Access Articles

Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable

Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable

Laxman Dhulipala
Carnegie Mellon University
ldhulipa@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

ABSTRACT

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 13 important graph problems. We also present the optimizations and 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 provide a publicly-available benchmark suite containing our implementations.

1 INTRODUCTION

Today, the largest publicly-available graph, the Hyperlink Web graph, consists of over 3.5 billion vertices and 128 billion edges [61]. This graph presents a significant computational challenge for both distributed and shared memory systems. Indeed, very few algorithms have been applied to this graph, and those that have often take hours to run [26, 45, 54], with the fastest times requiring between 1–6 minutes using a supercomputer [84, 85]. 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. [85] 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 38.3 seconds. Another recent result by Stergiou et al. [86] 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 8.9x 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 disk-based systems [26, 45, 54], is given in Section 6.

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 [28]. 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, maximal independent set, maximal matching, graph coloring, single-source shortest paths, betweenness centrality, minimum spanning forest, k-core decomposition, approximate set cover, and triangle counting. We describe the techniques used to achieve good performance on graphs with billions of vertices and hundreds of billions of edges and share experimental results for the Hyperlink 2012 and Hyperlink 2014 Web crawls, the largest and second largest publicly-available graphs, as well as several smaller real-world graphs at various scales. Some of the algorithms we

---

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

Problem	(1)	(72h)	(SU)	Algorithm	Model	Work	Depth
Breadth-First Search (BFS)	649	10.7	60	–	TS	$O(m)$	$O(\mathrm{diam}(G)\log n)$
Integral-Weight SSSP (weighted BFS)	3770	58.1	64	[28]	PW	$O(m)$ expected	$O(\mathrm{diam}(G)\log n)$ w.h.p.[†]
General-Weight SSSP (Bellman-Ford)	4010	59.4	67	[25]	PW	$O(\mathrm{diam}(G)m)$	$O(\mathrm{diam}(G)\log n)$
Single-Source Betweenness Centrality (BC)	2260	37.1	60	[19]	FA	$O(m)$	$O(\mathrm{diam}(G)\log n)$
Low-Diameter Decomposition (LDD)	1150	19.6	58	[63]	TS	$O(m)$ expected	$O(\log^2 n)$ w.h.p.
Connectivity	2080	38.3	54	[79]	TS	$O(m)$ expected	$O(\log^3 n)$ w.h.p.
Biconnectivity	9860	165	59	[87]	FA	$O(m)$ expected	$O(\max(\mathrm{diam}(G)\log n, \log^3 n))$ w.h.p.
Strongly Connected Components (SCC)*	8130	185	43	[12]	PW	$O(m\log n)$ expected	$O(\mathrm{diam}(G)\log n)$ w.h.p.
Minimum Spanning Forest (MSF)	9520	187	50	[90]	PW	$O(m\log n)$	$O(\log^2 n)$
Maximal Independent Set (MIS)	2190	32.2	68	[11]	FA	$O(m)$ expected	$O(\log^2 n)$ w.h.p.
Maximal Matching (MM)	7150	108	66	[11]	PW	$O(m)$ expected	$O(\log^3 m/\log\log m)$ w.h.p.
Graph Coloring	8920	158	56	[40]	FA	$O(m)$	$O(\log n + L\log \Delta)$
k-core	8515	184	46	[28]	FA	$O(m)$ expected	$O(\rho\log n)$ w.h.p.
Approximate Set Cover	5630	98.4	57	[14]	PW	$O(m)$ expected	$O(\log^3 n)$ w.h.p.
Triangle Counting (TC)	—	1470	—	[81]	–	$O(m^{3/2})$	$O(\log n)$

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. Theoretical bounds for the algorithms and the variant of the MT-RAM used are shown in the last three columns. We mark times that did not finish in 5 hours with —. *SCC was run on the directed version of the graph. [†]We say that an algorithm has $O(f(n))$ cost *with high probability (w.h.p.)* if it has $O(k \cdot f(n))$ cost with probability at least $1 - 1/n^k$.

describe are based on previous results from Ligra, Ligra+, and Julienne [28, 76, 80], and other papers on efficient parallel graph algorithms [11, 40, 81]. 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+ [80] to ensure that our graph primitives for mapping, filtering, reducing and packing the neighbors of a vertex were theoretically-efficient. We note that using compression techniques is crucial for representing the symmetrized Hyperlink 2012 graph in 1TB of RAM, as storing this graph in an uncompressed format would require over 900GB to store the edges alone, whereas the graph requires 330GB in our compressed format (less than 1.5 bytes per edge). We show the running times of our algorithms on the Hyperlink 2012 graph as well as their work and depth bounds in Table 1. To make it easy to build upon or compare to our work in the future, we describe 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., [3, 8, 21, 32, 44, 49, 53, 62–64, 68, 69, 75, 87]). A major goal in parallel graph algorithm design is to find *work-efficient* algorithms with polylogarithmic depth. While many suspect that work-efficient algorithms may not exist for all

parallelizable graph problems, as inefficiency may be inevitable for problems that depend on transitive closure, many problems that are of practical importance do admit work-efficient algorithms [48]. 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 [48], 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 [12, 72]. Other problems, like k-core are P-complete [4], which rules out polylogarithmic-depth algorithms for them unless P = NC [38]. However, even k-core admits an algorithm with strong theoretical guarantees that is efficient in practice [28].

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., [36, 52, 56, 65, 76] among many others). We refer the reader to [59, 89] for surveys of existing frameworks. Several recent graph processing systems evaluate the scalability of their implementations by solving problems on massive graphs [26, 28, 45, 54, 84, 86]. All of these systems report running times either on the Hyperlink 2012 graph or Hyperlink 2014 graphs, two web crawls released by the WebDataCommons that are the largest and second largest publicly-available graphs respectively. We describe these recent systems and give a detailed comparison of how our implementations perform compare to their codes in Section 6. We review existing parallel graph algorithm benchmarks in the full version of our paper [29].

3 PRELIMINARIES

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 $n - 1$. For undirected graphs we use $N(v)$ to denote the neighbors of vertex v and $deg(v)$ to denote its degree. For directed graphs, we use $in\text{-}deg(v)$ and $out\text{-}deg(v)$ to denote the in and out-neighbors of a vertex v. We use $\text{diam}(G)$ 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. 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.

Atomic Primitives. We use three common atomic primitives in our algorithms: test-and-set (TS), fetch-and-add (FA), and priority-write (PW). A TEST-AND-SET($\&x$) checks if x is 0, and if so atomically sets it to 1 and returns *true*; otherwise it returns *false*. A FETCH-AND-ADD($\&x$) atomically returns the current value of x and then increments x. A PRIORITY-WRITE($\&x, v, p$) 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*.

Model. In the analysis of algorithms we use the following 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 the model on the CRCW PRAM equipped with the same operations with an additional $O(\log^* n)$ factor in the depth due to load-balancing. Furthermore, a PRAM algorithm using P processors and T time can be simulated in our model with PT work and T depth.

The *Multi-Threaded Random-Access Machine (MT-RAM)* [10] 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 MT-RAM extends the RAM with a fork instruction that takes a positive integer k and forks k new child threads. Each child thread receives a unique integer in the range $[1, \ldots, k]$ in its first register and otherwise has the identical state as the parent, which has a 0 in that register. They all start by running the next instruction. When a thread performs a fork, it is suspended until all the children terminate (execute an end instruction). A computation starts with a single root thread and finishes when that root thread ends. 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 subthreads 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. We augment the model with three atomic instructions that are used by our algorithms:

test-and-set (TS), fetch-and-add (FA), and priority-write (PW) and discuss our model with these operations as the TS, FA, and PW variants of the MT-RAM. 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. More details about the model can be found in [10].

Parallel Primitives. The following parallel procedures are used throughout the paper. ***Scan*** takes as input an array A of length n, an associative binary operator $\oplus$, and an identity element $\perp$ such that $\perp \oplus x = x$ for any x, and returns the array $(\perp, \perp \oplus A[0], \perp \oplus A[0] \oplus A[1], \ldots, \perp \oplus_{i=0}^{n-2} A[i])$ as well as the overall sum, $\perp \oplus_{i=0}^{n-1} A[i]$. Scan can be done in $O(n)$ work and $O(\log n)$ depth (assuming $\oplus$ takes $O(1)$ work) [44]. ***Reduce*** takes an array A and a binary associative function f and returns the sum of the elements in A with respect to f. ***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 f takes $O(1)$ work).

Ligra, Ligra+, and Julienne. We make use of the Ligra, Ligra+, and Julienne frameworks for shared-memory graph processing in this paper and review components from these frameworks here [28, 76, 80]. Ligra provides data structures for representing a graph $G = (V, E)$, ***vertexSubset***s (subsets of the vertices). We make use of the EDGEMAP function provided by Ligra, which we use for mapping over edges. EDGEMAP takes as input a graph $G(V, E)$, a vertexSubset U, and two boolean functions F and C. EDGEMAP applies F to $(u, v) \in E$ such that $u \in U$ and $C(v) = true$ (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 $F(u, v) = true$. F can side-effect data structures associated with the vertices. EDGEMAP runs in $O(\sum_{u \in U} deg(u))$ work and $O(\log n)$ depth assuming F and C take $O(1)$ work. EDGEMAP either applies a ***sparse*** or ***dense*** method based on the number of edges incident to the current frontier. Both methods run in $O(\sum_{u \in U} deg(u))$ work and $O(\log n)$ depth. We note that in our experiments we use an optimized version of the dense method which examines in-edges sequentially and stops once C returns *false*. This optimization lets us potentially examine significantly fewer edges than the $O(\log n)$ depth version, but at the cost of $O(in\text{-}deg(v))$ depth.

4 ALGORITHMS

In this section we describe I/O specifications of our benchmark, discuss related work and present the theoretically-efficient algorithm we implemented for each problem. We cite the original papers that our algorithms are based on in Table 1. We mark implementations based on prior work with a † and discuss the related work, algorithms, and implementations for these problems in the full version of our paper [29]. The full version also contains self-contained descriptions of all of our algorithms.

Shortest Path Problems[†]

Problem: Breadth-First Search (BFS)
Input: $G = (V, E)$, an unweighted graph, SRC $\in V$.
Output: D, a mapping where $D[v]$ is the shortest path distance from SRC to v in G and ∞ if v is unreachable.

Problem: 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 and ∞ if v is unreachable.

Problem: 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 and ∞ if v is unreachable. All distances must be $-\infty$
if G contains a negative-weight cycle reachable from SRC.

Problem: Single-Source Betweenness Centrality (BC)
Input: $G = (V, E)$, an undirected graph, SRC $\in V$.
Output: S, a mapping from each vertex v to the centrality contribu-
tion from all (SRC, t) shortest paths that pass through v.

Low-Diameter Decomposition†

Input: $G = (V, E)$, a directed graph, $0 < \beta < 1$.
Output: $\mathcal{L}$, a mapping from each vertex to a cluster ID representing
a $(O(\beta), O((\log n)/\beta))$ decomposition. A (β, d)-decomposition par-
titions V into $V_1, \dots, V_k$ such that the shortest path between two
vertices in V_i using only vertices in V_i is at most d, and the number
of edges (u, v) where $u \in V_i, v \in V_j, j \neq i$ 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.

Biconnectivity

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

Sequentially, biconnectivity can be solved using the Hopcroft-Tarjan
algorithm [42]. The algorithm uses depth-first search (DFS) to iden-
tify 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 [2]. Tarjan and Vishkin
present the first work-efficient algorithm for biconnectivity [87]
(as stated in the paper the algorithm is not work-efficient, but it
can be made so by using a work-efficient connectivity algorithm).
Another approach relies on the fact that biconnected graphs admit
open ear decompositions to solve biconnectivity efficiently [57, 70].

In this paper, we implement the Tarjan-Vishkin algorithm for bi-
connectivity in $O(m)$ work and $O(\max(\text{diam}(G) \log n, \log^3 n))$ depth
on the FA-MT-RAM. Our implementation first computes connectiv-
ity labels using the algorithm from Section 4, which runs in $O(m)$
work and $O(\log^3 n)$ depth w.h.p. and picks an arbitrary source ver-
tex from each component. Next, we compute a spanning forest
rooted at these sources using breadth-first search, which runs in
$O(m)$ work and $O(\text{diam}(G) \log n)$ depth. We note that the connec-
tivity algorithm can be modified to compute a spanning forest in
the same work and depth as connectivity, which would avoid the
breadth-first-search. We compute *Low*, *High*, and *Size* for each ver-
tex by running leaffix and rootfix sums on the spanning forests
produced by BFS with fetch-and-add, which requires $O(n)$ work

and $O(\text{diam}(G))$ depth. Finally, we compute an implicit represen-
tation of the biconnectivity labels for each edge, using an idea
from [7]. This step computes per-vertex labels by removing all criti-
cal 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 $O(\max(\text{diam}(G) \log n, \log^3 n))$ w.h.p.

Minimum Spanning Forest

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

Borůvka gave the first known sequential and parallel algorithm
for computing a minimum spanning forest (MSF) [18]. Significant
effort has gone into finding linear-work MSF algorithms both in
the sequential and parallel settings [21, 47, 68]. Unfortunately, the
linear-work parallel algorithms are highly involved and do not
seem to be practical. Significant effort has also gone into designing
practical parallel algorithms for MSF; we discuss relevant experi-
mental work in Section 6. 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 algo-
rithm that runs in $O(m \log n)$ work and $O(\log^2 n)$ depth on the
PW-MT-RAM. Our implementation is based on a recent implemen-
tation of Borůvka by Zhou [90] that runs on the edgelist format. We
made several changes to the algorithm which improve performance
and allow us to solve MSF on graphs stored in the CSR/CSC for-
mat, as storing an integer-weighted graph in edgelist format would
require well over 1TB of memory to represent the edges in the
Hyperlink2012 graph alone. Our code uses an implementation of
Borůvka that works over an edgelist; to make it efficient we ensure
that the size of the lists passed to it are much smaller than m. Our
approach is to perform a constant number of *filtering* steps. Each
filtering step solves an approximate k'th smallest problem in order
to extract the lightest $3n/2$ edges in the graph (or all remaining
edges) and runs Borůvka on this subset of edges. We then filter the
remaining graph, packing out any edges that are now in the same
component. This idea is similar to the theoretically-efficient algo-
rithm of Cole et al. [21], except that instead of randomly sampling
edges, we select 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, they do not affect the work and
depth asymptotically. In practice, most of the edges are removed
after 3–4 filtering steps, and so the remaining edges can be copied
into an edgelist and solved in a single Borůvka step. We also note
that as the edges are initially represented in both directions, we can
pack out the edges so that each undirected edge is only inspected
once (we noticed that earlier edgelist-based implementations stored
undirected edges in both directions).

Strongly Connected Components

Input: $G(V, E)$, a directed graph.

Output: $\mathcal{L}$, a mapping from each vertex to the label of its strongly-connected component.

Tarjan's algorithm is the textbook sequential algorithm for computing the strongly connected components (SCCs) of a directed graph [25]. As it uses depth-first search, we currently do not know how to efficiently parallelize it [2]. 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 [35], 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 [12, 24, 34, 72]. 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. [12]. Our implementation runs in in $O(m \log n)$ expected work and $O(\mathrm{diam}(G) \log n)$ depth w.h.p. on the PW-MT-RAM. One of the challenges in implementing this SCC algorithm is how to compute reachability information from multiple vertices simultaneously and how to combine the information to (1) identify SCCs and (2) refine the subproblems of visited vertices. In our implementation, we explicitly store $\mathcal{R}_F$ and $\mathcal{R}_B$, the forward and backward reachability sets for the set of centers that are active in the current phase, C_A. The sets are represented as hash tables that store tuples of vertices and center IDs, (u, c_i), representing a vertex u in the same subproblem as c_i that is visited by a directed path from c_i. We explain how to make the hash table technique practical in Section 5. 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 test-and-set's a visited flag for v, and returns it in the next frontier if the test-and-set succeeded. Each BFS requires at most $O(\mathrm{diam}(G))$ rounds as each search adds the same labels on each round as it would have had it run in isolation.

After computing $\mathcal{R}_F$ and $\mathcal{R}_B$, we deterministically assign (with respect to the random permutation of vertices generated at the start of the algorithm) vertices that we visited in this phase a new label, which is either the label of a refined subproblem or a unique label for the SCC they are contained in. We first intersect the two tables and perform, for any tuple (v, c_i) contained in the intersection, a priority-write with min on the memory location corresponding to v's SCC label with c_i as the label. Next, for all pairs (v, c_i) in $\mathcal{R}_F \oplus \mathcal{R}_B$ we do a priority-write with min on v's subproblem label, which ensures that the highest priority search that visited v sets its new subproblem.

We implemented 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 [20]. We also implemented a 'trimming' optimization that is reported in the literature [60, 83], 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.

Maximal Independent Set and Maximal Matching

Problem: 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.

Problem: 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'.

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 [3, 8, 11, 43, 49, 53]. 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 [11]. Recently, Fischer and Noever showed an improved depth bound for this algorithm [33].

In this paper, we implement the rootset-based algorithm for MIS from Blelloch et al. [11] which runs in $O(m)$ expected work and $O(\log^2 n)$ depth w.h.p. on the FA-MT-RAM. To the best of our knowledge this is the first implementation of the rootset-based algorithm; the implementations from [11] are based on processing appropriately-sized prefixes of an order generated by a random permutation P. Our implementation of the rootset-based algorithm works on a priority-DAG defined by directing edges in the graph from the higher-priority endpoint to the lower-priority endpoint. On each round, we add all roots of the DAG into the MIS, compute $N(roots)$, the neighbors of the rootset that are still active, and finally decrement the priorities of $N(N(roots))$. As the vertices in $N(roots)$ are at arbitrary depths in the priority-DAG, we only decrement the priority along an edge (u, v), $u \in N(roots)$ if $P[u] < P[v]$. The algorithm runs in $O(m)$ work as we process each edge once; the depth bound is $O(\log^2 n)$ as the priority-DAG has $O(\log n)$ depth w.h.p. [33], and each round takes $O(\log n)$ depth. We were surprised that this implementation usually outperforms the prefix-based implementation from [11], while also being simple to implement.

Our maximal matching implementation is based on the prefix-based algorithm from [11] that takes $O(m)$ expected work and $O(\log^3 m / \log \log m)$ depth w.h.p. on the PW-MT-RAM (using the improved depth shown in [33]). We had to make several modifications to run the algorithm on the large graphs in our experiments. The original code from [11] 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 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.

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.

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 [40, 88]. 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 in Ligra, which runs in $O(m + n)$ work and $O(L \log \Delta + \log n)$ depth on the FA-MT-RAM. The algorithm is implemented similarly to our rootset-based algorithm for MIS. In each round, after coloring the roots we use a fetch-and-add 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.

k-core

Input: $G = (V, E)$, an undirected graph.
Output: D, a mapping from each vertex to its coreness value.

k-cores were defined independently by Seidman [73], and by Matula and Beck [58] who also gave a linear-time algorithm for computing the *coreness* value of all vertices, i.e. the maximum k-core a vertex participates in. Anderson and Mayr showed that k-core (and therefore coreness) is in NC for $k \leq 2$, but is P-complete for $k \geq 3$ [4]. 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 [28], the authors give a parallel algorithm based on bucketing that runs in $O(m + n)$ expected work, and $\rho \log n$ depth w.h.p. 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.

Our implementation of k-core in this paper is based on the implementation from Julienne [28]. One of the challenges to implementing the peeling algorithm for k-core is efficiently computing the number of edges removed from each vertex that remains in the graph. A simple approach is to just fetch-and-add a counter per vertex, and update the bucket of the vertex based on this counter, however this incurs significant contention on real-world graphs with vertices with large degree. In order to make this step faster in practice, we implemented a work-efficient histogram which compute the number of edges removed from remaining vertices while incurring very little contention. We describe our histogram implementation in Section 5.

Approximate Set Cover[†]

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

Triangle Counting[†]

Input: $G = (V, E)$, an undirected graph.
Output: T_G, the total number of triangles in G.

5 IMPLEMENTATIONS AND TECHNIQUES

In this section, we introduce several general implementation techniques and optimizations that we use in our algorithms. Due to lack of space, we describe some techniques, such as a more cache-friendly sparse EDGEMAP that we call EDGEMAPBLOCKED, and compression techniques in the full version of our paper [29].

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 fetch-and-adds on high-degree vertices. This occurs as many social-networks and web-graphs have large maximum degree, but relatively small degeneracy, or largest non-empty core (labeled k_{max} in Table 2). For these graphs, we observed that many early rounds, which process vertices with low coreness perform a large number of fetch-and-adds on memory locations corresponding to high-degree vertices, resulting in high contention [77]. To reduce contention, we designed a work-efficient histogram implementation that can perform this step while only incurring $O(\log n)$ contention w.h.p. The **Histogram** primitive takes a sequence of (K, T) pairs, and an associative and commutative operator $R : T \times T \rightarrow T$ and computes a sequence of (K, 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 [39]. 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 w.h.p. (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 w.h.p, we incur at most $O(\log n)$ contention w.h.p. 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 [78]. 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 [9] 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 fetch-and-add-based version of k-core and saw between a 1.1–3.1x improvement from using the histogram.

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

The main challenge in implementing one round in the traversal is (1) ensuring that the table has sufficient space to store all pairs that will be added this round, and (2) efficiently iterating over all of the pairs associated with a vertex. We implement (1) by performing a parallel reduce to sum over vertices $u \in F$, the current frontier, the number of neighbors v in the same subproblem, multiplied by

the number of distinct labels currently assigned to u. This upper-bounds the number of distinct labels that could be added this round, and although we may overestimate the number of actual additions, we will never run out of space in the table. We update the number of elements currently in the table during concurrent insertions by storing a per-processor count which gets incremented whenever the processor performs a successful insertion. The counts are then summed together at the end of a round and used to update the count of the number of elements in the table.

One simple implementation of (2) is to simply allocate $O(\log n)$ space for every vertex, as the maximum number of centers that visit any vertex during a phase is at most $O(\log n)$ w.h.p. However, this will waste a significant amount of space, as most vertices are visited just a few times. Instead, our implementation stores (u, c) pairs in the table for visited vertices u, and computes hashes based only on the ID of u. As each vertex is only expected to be visited a constant number of times during a phase, the expected probe length is still a constant. Storing the pairs for a vertex in the same probe-sequence is helpful for two reasons. First, we may incur fewer cache misses than if we had hashed the pairs based on both entries, as multiple pairs for a vertex can fit in the same cache line. Second, storing the pairs for a vertex along the same probe sequence makes it extremely easy to find all pairs associated with a vertex u, as we simply perform linear-probing, reporting all pairs that have u as their key until we hit an empty cell. Our experiments show that this technique is practical, and we believe that it may have applications in similar algorithms, such as computing least-element lists or FRT trees in parallel [12, 13].

6 EXPERIMENTS

In this section, we describe our experimental results on a set of real-world graphs and also discuss related experimental work. Tables 1 and 3 show the running times for our implementations on our graph inputs. For compressed graphs, we use the compression schemes from Ligra+ [80], which we extended to ensure theoretical efficiency. We describe these modifications and also other statistics about our algorithms (e.g., number of colors used, number of SCCs, etc.) in the full version of the paper [29].

Experimental Setup. We run all of our experiments on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with $4 \times$ 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 [16]. 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

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

Table 2: Graph inputs, including vertices and edges. diam **is the diameter of the graph. For undirected graphs,** ρ **and** k_{max} **are the number of peeling rounds, and the largest non-empty core (degeneracy). We mark** diam **values where we are unable to calculate the exact diameter with** * **and report the effective diameter observed during our experiments, which is a lower bound on the actual diameter.**

ran our implementations 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 [28], and degeneracy (for undirected graphs) in Table 2. ***LiveJournal*** is a directed graph of the social network obtained from a snapshot in 2008 [17]. ***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 [51]. ***ClueWeb*** is a Web graph from the Lemur project at CMU [17]. ***Hyperlink2012*** and ***Hyperlink2014*** are directed hyperlink graphs obtained from the WebDataCommons dataset where nodes represent web pages [61]. ***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, 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.

SSSP Problems. Our BFS, weighted BFS, Bellman-Ford, and betweenness centrality implementations achieve between a 8–67x speedup across all inputs. We ran all of our shortest path experiments on the *symmetrized* versions of the graph. Our experiments show that our weighted BFS and Bellman-Ford implementations perform as well as or better than our prior implementations from Julienne [28]. Our running times for BFS and betweenness centrality are the same as the times of the implementations in Ligra [76]. 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 many 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 amount of memory when processing the largest frontiers which then caused the graph to become partly evicted from the page cache.

In an earlier paper [28], we compared the running time of our weighted BFS implementation to two existing parallel shortest path implementations from the GAP benchmark suite [6] and Galois [55], as well as a fast sequential shortest path algorithm from the DI-MACS 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 EDGEMAP-BLOCKED 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.

Connectivity Problems. Our low-diameter decomposition (LDD) implementation achieves between 17–58x 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 achieves 25–57x speedup across all inputs. We note that our implementation does not assume that vertex IDs in the graph are randomly permuted and always generates a random permutation, even on the first round, as adding vertices based on their original IDs can result in poor performance. There are several existing implementations of fast parallel connectivity algorithms [67, 78, 79, 83], however, only the implementation from [79], which presents the algorithm that we implement in this paper, is theoretically-efficient. The implementation from Shun et al. was compared to both the Multistep [83] and Patwary et al. [67] 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.

Despite our biconnectivity implementation having $O(\text{diam}(G))$ depth, our implementation achieves between a 20–59x speedup across all inputs, as the diameter of most of our graphs is extremely low. Our biconnectivity implementation is about 3–5 times slower than running connectivity on the graph, which seems reasonable as our current implementation performs two calls to connectivity, and one breadth-first search. There are a several existing implementations of biconnectivity. Cong and Bader [22] parallelize the Tarjan-Vishkin algorithm and demonstrated speedup over the Hopcroft-Tarjan (HT) algorithm. Edwards and Vishkin [31] also implement the Tarjan-Vishkin algorithm using the XMT platform, and show that their algorithm achieves good speedups. Slota and Madduri [82] 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

Application	LiveJournal-Sym			com-Orkut			Twitter-Sym			3D-Torus			ClueWeb-Sym			Hyperlink2014-Sym		
	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(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	106	2.29	46.2	250	4.50	55.5
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	736	14.4	51.1	1390	22.3	62.3
General-Weight SSSP (Bellman-Ford)	1.96	0.086	22.7	3.98	0.168	23.6	48.7	1.56	31.2	6280	133	47.2	1050	16.2	64.8	1460	22.9	63.7
Single-Source Betweenness Centrality (BC)	1.66	0.049	33.8	2.52	0.057	44.2	26.3	3.26	8.06	496	12.5	39.6	569	27.7	20.5	866	16.3	53.1
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	176	3.62	48.6	322	6.84	47.0
Connectivity	1.20	0.050	24.0	1.64	0.056	29.2	26.1	0.807	32.3	351	14.3	24.5	552	11.2	49.2	990	17.1	57.8
Biconnectivity	5.36	0.261	20.5	7.31	0.292	25.0	146	4.86	30.0	1610	59.6	27.0	2250	48.7	46.2	3520	71.5	49.2
Strongly Connected Components (SCC)*	1.61	0.116	13.8	~	~	~	13.3	0.495	26.8	~	~	~	1240	38.1	32.5	2140	51.5	41.5
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	2490	45.6	54.6	3580	71.9	49.7
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	551	8.44	65.2	1020	14.5	70.3
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	1760	31.8	55.3	2980	48.1	61.9
Graph Coloring	4.69	0.392	11.9	9.05	0.789	11.4	148	6.91	21.4	350	11.3	30.9	2050	49.8	41.1	3310	63.1	52.4
k-core	3.75	0.641	5.85	8.32	1.33	6.25	110	6.72	16.3	753	6.58	114.4	2370	62.9	37.6	3480	83.2	41.8
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	1490	28.1	53.0	2040	37.6	54.2
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	—	272	—	—	568	—

Table 3: Running times (in seconds) of our algorithms over symmetric graph inputs on a 72-core machine (with hyper-threading) where (1) is the single-thread time, (72h) is the 72 core time using hyper-threading, and (SU) is the parallel speedup (single-thread time divided by 72-core time). We mark experiments that are not applicable for a graph with ~, and experiments that did not finish within 5 hours with —. *SCC was run on the directed versions of the input graphs.

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

Our strongly connected components implementation achieves between a 13–43x speedup across all inputs. Our implementation takes a parameter β, which is the base of the exponential rate at which we grow the number of centers added. We set β between 1.1–2.0 for our experiments and note that using a larger value of β can improve the running time on smaller graphs by up to a factor of 2x. Our SCC implementation is between 1.6x faster to 4.8x slower than running connectivity on the graph. There are several existing SCC implementations that have been evaluated on real-world directed graphs [41, 60, 83]. The Hong et al. algorithm [41] is a modified version of the FWBW-Trim algorithm from McLendon et al. [60], but neither algorithm has any theoretical bounds on work or depth. Unfortunately [41] do not report running times, so we are unable to compare our performance with them. The Multistep algorithm [83] 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 17–50x 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 [5, 23, 66, 78, 90]. We compared our implementation with the union-find based MSF implementation from PBBS [78] and the implementation of Borůvka from [90], 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 [90] our implementation is between 1.2–2.9x faster.

MIS, Maximal Matching, and Graph Coloring. Our MIS and maximal matching implementations achieve between 31–70x and

25–70x speedup across all inputs. The implementations by Blelloch et al. [11] 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 [11]. 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 [40], due to synchronizing on many rounds which contain few vertices.

k-core, Approximate Set Cover, and Triangle Counting. 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 [27, 28, 46, 71]. Both the ParK algorithm [27] and Kabir and Madduri algorithm [46] 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 set cover implementation achieves between 5–57x speedup across all inputs. Our implementation is based on the implementation presented in Julienne [28]; 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 [15] 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
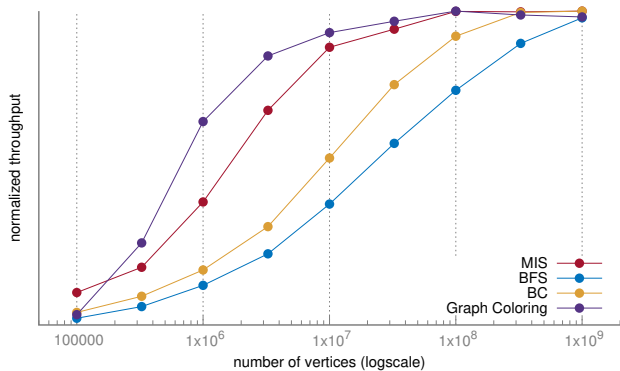
Figure 1: Log-linear plot of normalized throughput vs. vertices for MIS, BFS, BC, and coloring on the 3D-Torus graph family.

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

Table 4: Cycles stalled while the memory subsystem has an outstanding load (trillions), LLC hit rate and misses (billions), bandwidth in GB/s (bytes read and written from memory, divided by running time), and running time in seconds. All experiments are run on the ClueWeb graph using 72 cores with hyper-threading.

of 3D-Torus. On 3D-Torus, the implementation from [15] 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.

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 experimental papers that consider multicore triangle counting [1, 37, 50, 52, 74, 81]. We implement the algorithm from [81], 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 [81]. 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.

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. Our slightly worse scaling on this graph can be accounted for by the fact that we stored the graph ordered by dimension, instead of storing it using a local ordering. It would be interesting to see how much improvement we could gain by reordering the vertices.

In Figure 1 we show the normalized throughput of MIS, BFS, BC, and graph coloring for 3-dimensional tori of different sizes, where throughput is measured as the number of edges processed per second. The throughput for each application becomes saturated before our largest-scale graph for all applications except for BFS, which is saturated on a graph with 2 billion vertices. The throughput curves show that the theoretical bounds are useful in predicting how the half-lengths[3] are distributed. The half-lengths are ordered

---

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

as follows: coloring, MIS, BFS, and BC. This is the same order as sorting these algorithms by their depth with respect to this graph.

Locality. While our algorithms are efficient on the MT-RAM, we do not analyze their cache complexity, and in general they may not be efficient in a model that takes caches into account. Despite this, we observed that our algorithms have good cache performance on the graphs we tested on. In this section we give some explanation for this fact by showing that our primitives make good use of the caches. Our algorithms are also aided by the fact that these graph datasets often come in highly local orders (e.g., see the *Natural* order in [30]). Table 4 shows metrics for our experiments measured using Open Performance Counter Monitor (PCM).

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

Processing Massive Web Graphs. In Tables 1 and 3, we show the running times of our implementations on the ClueWeb, Hyperlink2014, and Hyperlink2012 graphs. To put our performance in context, we compare our 72-core running times to running times reported by existing work. Table 5 summarizes the existing results in the literature. Most results process the *directed* versions of these graphs, which have about half as many edges as the symmetrized version. Unless otherwise mentioned, all results from the literature use the directed versions of these graphs. To make the comparison easier we show our running times for BFS, SSSP (weighted BFS), BC and SCC on the directed graphs, and running times for Connectivity, k-core and TC on the symmetrized graphs in Table 5.

FlashGraph [26] 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 12x faster than their implementations, respectively. Mosaic [54] 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

Paper	Problem	Graph	Memory	Hyper-threads	Nodes	Time
Mosaic [54]	BFS*	2014	0.768	1000	1	6.55
	Connectivity*	2014	0.768	1000	1	708
	SSSP*	2014	0.768	1000	1	8.6
FlashGraph [26]	BFS*	2012	.512	64	1	208
	BC*	2012	.512	64	1	595
	Connectivity*	2012	.512	64	1	461
	TC*	2012	.512	64	1	7818
BigSparse [45]	BFS*	2012	0.064	32	1	2500
	BC*	2012	0.064	32	1	3100
Slota et al. [85]	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. [86]	Connectivity	2012	128	24000	1000	341
This paper	BFS*	2014	1	144	1	5.71
	SSSP*	2014	1	144	1	9.08
	BFS*	2012	1	144	1	16.7
	BC*	2012	1	144	1	35.2
	Connectivity	2012	1	144	1	38.3
	SCC*	2012	1	144	1	185
	k-core	2012	1	144	1	184
	TC	2012	1	144	1	1470

Table 5: System configurations (memory in terabytes, hyper-threads, and nodes) and running times (seconds) of existing results on the Hyperlink graphs. The last section shows our running times. *These problems are run on directed versions of the graph.

24 hyper-threads and 4 Xeon-Phis (244 cores with 4 threads each) for a total of 1000 hyper-threads, 768GB of RAM, and 6 NVMes. Our BFS and connectivity implementations are 1.1x and 40x faster respectively, and our SSSP implementation is 1.05x slower. Both FlashGraph and Mosaic compute weakly connected components, which is equivalent to connectivity. BigSparse [45] report disk-based running times for BFS and BC on the Hyperlink2012 graph on a 32-core machine. They solve BFS in 2500s and BC in 3100s. Our BFS and BC implementations are 149x and 88x faster than their implementations, respectively.

Slota et al. [85] 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 1.6x 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.

Stergiou et al. [86] describe a connectivity algorithm that runs in $O(\log n)$ rounds in the BSP model and report running times for the symmetrized Hyperlink2012 graph. They implement their algorithm using a proprietary in-memory/secondary-storage graph processing system used at Yahoo!, and run experiments on a 1000 node cluster. Each node contains two 6-core processors that are

2-way hyper-threaded and 128GB of RAM, for a total of 24000 hyper-threads and 128TB of RAM. Their fastest running time on the Hyperlink2012 graph is 341s on their 1000 node system. Our implementation solves connectivity on this graph in 38.3s—8.8x 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.

7 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 outperform existing implementations on the largest real-world graphs, and use much 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.

REFERENCES

[1] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 2017.

[2] A. Aggarwal, R. J. Anderson, and M.-Y. Kao. Parallel depth-first search in general directed graphs. In *STOC*, 1989.

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

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

[5] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *JPDC*, 2006.

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

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

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

[9] G. E. Blelloch. Prefix sums and their applications. *Synthesis of Parallel Algorithms*, 1993.

[10] G. E. Blelloch and L. Dhulipala. Introduction to parallel algorithms 15-853: Algorithms in the real world. 2018.

[11] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *SPAA*, 2012.

[12] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallelism in randomized incremental algorithms. In *SPAA*, 2016.

[13] G. E. Blelloch, Y. Gu, and Y. Sun. A new efficient construction on probabilistic tree embeddings. In *ICALP*, 2017.

[14] G. E. Blelloch, R. Peng, and K. Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *SPAA*, 2011.

[15] G. E. Blelloch, H. V. Simhadri, and K. Tangwongsan. Parallel and I/O efficient set covering algorithms. In *SPAA*, 2012.

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

[17] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW*, 2004.

[18] O. Borůvka. O jistém problému minimálním. *Práce Mor. Přírodověd. Spol. v Brně III*, 3, 1926.

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

[20] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer networks*, 33(1-6), 2000.

[21] R. Cole, P. N. Klein, and R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *SPAA*, 1996.

[22] G. Cong and D. A. Bader. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In *IPDPS*, 2005.

[23] G. Cong and I. G. Tanase. Composable locality optimizations for accelerating parallel forest computations. In *HPCC*, 2016.

[24] D. Coppersmith, L. Fleischer, B. Hendrickson, and A. Pinar. A divide-and-conquer algorithm for identifying strongly connected components. 2003.

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

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

[27] N. S. Dasari, R. Desh, and M. Zubair. ParK: An efficient algorithm for k-core decomposition on multicore processors. In *Big Data*, 2014.

[28] L. Dhulipala, G. E. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *SPAA*, 2017.

[29] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *CoRR*, abs/1805.05208, 2018.

[30] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *KDD*, 2016.

[31] J. A. Edwards and U. Vishkin. Better speedups using simpler parallel programming for graph connectivity and biconnectivity. In *PMAM*, 2012.

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

[33] M. Fischer and A. Noever. Tight analysis of parallel randomized greedy MIS. In *SODA*, 2018.

[34] L. K. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *IPDPS*, 2000.

[35] H. Gazit and G. L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Information Processing Letters*, 28(2), 1988.

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

[37] O. Green, L. M. Munguia, and D. A. Bader. Load balanced clustering coefficients. In *PPAA*, 2014.

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

[39] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch. A top-down parallel semisort. In *SPAA*, 2015.

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

[41] S. Hong, N. C. Rodia, and K. Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *SC*, 2013.

[42] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 1973.

[43] A. Israeli and Y. Shiloach. An improved parallel algorithm for maximal matching. *Inf. Process. Lett.*, 1986.

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

[45] S. W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind. BigSparse: High-performance external graph analytics. *CoRR*, abs/1710.07736, 2017.

[46] H. Kabir and K. Madduri. Parallel k-core decomposition on multicore platforms. In *IPDPSW*, 2017.

[47] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2), Mar. 1995.

[48] R. M. Karp and V. Ramachandran. Handbook of theoretical computer science (vol. a). chapter Parallel Algorithms for Shared-memory Machines. MIT Press, Cambridge, MA, USA, 1990.

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

[50] J. Kim, W.-S. Han, S. Lee, K. Park, and H. Yu. OPT: A new framework for overlapped and parallel triangulation in large-scale graphs. In *SIGMOD*, 2014.

[51] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

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

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

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

[55] S. Maleki, D. Nguyen, A. Lenharth, M. Garzarán, D. Padua, and K. Pingali. DSMR: A parallel algorithm for single-source shortest path problem. In *ICS*, 2016.

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

[57] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theoretical Computer Science*, 47, 1986.

[58] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3), July 1983.

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

[60] W. Mclendon Iii, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8), 2005.

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

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

[63] G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decompositions using random shifts. In *SPAA*, 2013.

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

[65] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, 2013.

[66] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan. Scalable parallel minimum spanning forest computation. In *PPoPP*, 2012.

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

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

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

[70] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In *Synthesis of Parallel Algorithms*, 1993.

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

[72] W. Schudy. Finding strongly connected components in parallel using $O(\log^2 N)$ reachability queries. In *SPAA*, 2008.

[73] S. B. Seidman. Network structure and minimum degree. *Soc. Networks*, 5(3), 1983.

[74] M. Sevenich, S. Hong, A. Welc, and H. Chafi. Fast in-memory triangle listing for large real-world graphs. In *Workshop on Social Network Mining and Analysis*, 2014.

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

[76] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, 2013.

[77] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *SPAA*, 2013.

[78] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, 2012.

[79] J. Shun, L. Dhulipala, and G. E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *SPAA*, 2014.

[80] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *DCC*, 2015.

[81] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *ICDE*, 2015.

[82] G. M. Slota and K. Madduri. Simple parallel biconnectivity algorithms for multicore platforms. In *HiPC*, 2014.

[83] G. M. Slota, S. Rajamanickam, and K. Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *IPDPS*, 2014.

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

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

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

[87] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 1985.

[88] D. J. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 1967.

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

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