Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing

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

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

Julian Shun
UC Berkeley
jshun@eecs.berkeley.edu

ABSTRACT

Existing graph-processing frameworks let users develop efficient implementations for many graph problems, but none of them support efficiently bucketing vertices, which is needed for *bucketing-based* graph algorithms such as Δ-stepping and approximate set-cover. Motivated by the lack of simple, scalable, and efficient implementations of bucketing-based algorithms, we develop the Julienne framework, which extends a recent shared-memory graph processing framework called Ligra with an interface for maintaining a collection of buckets under vertex insertions and bucket deletions.

We provide a theoretically efficient parallel implementation of our bucketing interface and study several bucketing-based algorithms that make use of it (either bucketing by remaining degree or by distance) to improve performance: the peeling algorithm for k-core (coreness), Δ-stepping, weighted breadth-first search, and approximate set cover. The implementations are all simple and concise (under 100 lines of code). Using our interface, we develop the first work-efficient parallel algorithm for k-core in the literature with nontrivial parallelism.

We experimentally show that our bucketing implementation scales well and achieves high throughput on both synthetic and real-world workloads. Furthermore, the bucketing-based algorithms written in Julienne achieve up to 43x speedup on 72 cores with hyper-threading over well-tuned sequential baselines, significantly outperform existing work-inefficient implementations in Ligra, and either outperform or are competitive with existing special-purpose parallel codes for the same problem. We experimentally study our implementations on the largest publicly available graphs and show that they scale well in practice, processing real-world graphs with billions of edges in seconds, and hundreds of billions of edges in a few minutes. As far as we know, this is the first time that graphs at this scale have been analyzed in the main memory of a single multicore machine.

1 INTRODUCTION

Both the size and availability of real-world graphs has increased dramatically over the past decade. Due to the need to process this data quickly, many frameworks for processing massive graphs have been developed for both distributed-memory and shared-memory parallel machines such as Pregel [36], GraphLab [32, 33], PowerGraph [22], and Ligra [51]. Implementing algorithms using frameworks instead of as one-off programs enables users to easily take advantage of optimizations already implemented by the framework, such as direction-optimization, compression and parallelization over both the vertices and edges of a set of vertices [5, 55].

The performance of algorithms in these frameworks is often determined by the total amount of work performed. Unfortunately, the simplest algorithms to implement in existing frameworks are often work-inefficient, i.e., they perform asymptotically more work than the most efficient sequential algorithm. While work-inefficient algorithms can exhibit excellent self-relative speedup, their absolute performance can be an order of magnitude worse than the running time of the baseline sequential algorithm, even on a very large number of cores [38].

Many commonly implemented graph algorithms in existing frameworks are *frontier-based* algorithms. Frontier-based algorithms proceed in rounds, where each round performs some computation on vertices in the current frontier, and frontiers can change from round to round. For example, in breadth-first search (BFS), the frontier on round i is the set of vertices at distance i from the source of the search. In label propagation implementations of graph connectivity [22, 51], the frontier on each round consists of vertices whose labels changed in the previous round.

However, several fundamental graph algorithms cannot be expressed as frontier-based algorithms. These algorithms, which we call *bucketing-based* algorithms, maintain vertices in a set of ordered buckets. In each round, the algorithm extracts the vertices contained in the lowest (or highest) bucket and performs some computation on these vertices. It can then update the buckets containing either the extracted vertices or their neighbors. Frontier-based algorithms are a special case of bucketing-based algorithms, specifically they are bucketing-based algorithms that only use one bucket.

As an example, consider the weighted breadth-first search (wBFS) algorithm, which solves the single-source shortest path problem (SSSP) with nonnegative, integral edge weights in parallel [18]. Like BFS, wBFS processes vertices level by level, where level i contains all vertices at distance exactly i from *src*, the source vertex. The i'th round relaxes the neighbors of vertices in level i and updates any distances that change. Unlike a BFS, where the unvisited neighbors of the current level are in the next level, the neighbors of a level in wBFS can be spread across multiple levels. Because of this, wBFS maintains the levels in an ordered set of buckets. On round i, if a vertex v can decrease the distance to a neighbor u it places u in bucket $i+d(v,u)$. Finding the vertices in a given level can then easily be done using the bucket structure. We can show that the work of this algorithm is $O(r_{src} + |E|)$ and the depth is $O(r_{src} \log |V|)$ where r_{src} is the eccentricity from *src* (see Section 2). However, without

SPAA '17, , July 24-26, 2017, Washington DC, USA

ACM ISBN 978-1-4503-4593-4/17/07…$15.00
https://doi.org/10.1145/3087556.3087580

bucketing, the algorithm has to scan all vertices in each round to compute the current level, which makes it perform $O(r_{src}|V| + |E|)$ work and the same depth, which is not work-efficient.

In this paper, we study four bucketing-based graph algorithms—k-core[1], Δ-stepping, weighted breadth-first search (wBFS), and approximate set-cover. To provide simple and theoretically-efficient implementations of these algorithms, we design and implement a work-efficient interface for bucketing in the Ligra shared-memory graph processing framework [51]. Our extended framework, which we call ***Julienne***, enables us to write short (under 100 lines of code) implementations of the algorithms that are efficient and achieve good parallel speedup (up to 43x on 72 cores with two-way hyper-threading). Furthermore we are able to process the largest publicly-available real-world graph containing over 225 billion edges in the memory of a single multicore machine [39]. This graph must be compressed in order to be processed even on a machine with 1TB of main memory. Because Julienne supports the compression features of Ligra+, we were able to run our codes on this graph without extra modifications [55]. All of our implementations either outperform or are competitive with hand-optimized codes for the same problem. We summarize the cost bounds for the algorithms developed in this paper in Table 1.

Using our framework, we obtain the first work-efficient algorithm for k-core with nontrivial parallelism. The sequential requires performs $O(n + m)$ work [4], however the best prior parallel algorithms [16, 20, 41, 44, 51] require at least $O(k_{max}n+m)$ work where k_{max} is the largest core number in the graph—this is because these algorithms scan all remaining vertices when computing vertices in a particular core. By using bucketing, our algorithm only scans the edges of vertices with minimum degree, which makes it work-efficient. On a graph with 225B edges using 72 cores with two-way hyper-threading, our work-efficient implementation takes under 4 minutes to complete, whereas the work-inefficient implementation does not finish even after 3 hours.

Contributions. The main contributions of this paper are as follows.

(1) A simple interface for dynamically maintaining sets of identifiers in buckets.
(2) A theoretically efficient parallel algorithm that implements our bucketing interface, and four applications implemented using the interface.
(3) The first work-efficient implementation of k-core with non-trivial parallelism.
(4) Experimental results on the largest publicly available graphs, showing that our codes achieve high performance while remaining simple. To the best of our knowledge, this work is the first time graphs at the scale of billions of vertices and hundreds of billions of edges have been analyzed in minutes in the memory of a single shared-memory server.

[1]The definitions of k-core and coreness (see Secton 4.1) have been used interchangeably in the literature, however they are not the same problem, as pointed out in [46]. In this paper we use k-core to refer to the coreness problem. Note that computing a particular k-core from the coreness numbers requires finding the largest induced subgraph among vertices with coreness at least k, which can be done efficiently in parallel.

Algorithm	Work	Depth	Parameters
k-core	$O(\|E\| + \|V\|)$	$O(\rho \log \|V\|)$	ρ: peeling complexity, see Section 4.1.
wBFS	$O(r_{src} + \|E\|)$	$O(r_{src} \log \|V\|)$	r_{src}: eccentricity from the source vertex *src*, see Section 2.
Δ-stepping	$O(w_\Delta)$	$O(d_\Delta \log \|V\|)$	w_Δ, d_Δ: work and number of rounds of the original Δ-stepping algorithm.
Approximate Set Cover	$O(M)$	$O(\log^3 M)$	M: sum of the sizes of the sets.

Table 1: Cost bounds for the algorithms developed in this paper. The work bounds are in expectation and the depth bounds are with high probability.

2 PRELIMINARIES

We denote a directed unweighted graph by $G(V, E)$ where V is the set of vertices and E is the set of (directed) 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. We use r_s to denote the eccentricity, or longest shortest path distance between a vertex s and any vertex v reachable from s. We assume that there are no self-edges or duplicate edges in the graph.

We analyze algorithms in the work-depth model, where the ***work*** is the number of operations used by the algorithm and the ***depth*** is the length of the longest sequential dependence in the computation [25]. We allow for concurrent reads and writes in the model. A ***compare-and-swap*** (CAS) is an atomic instruction that takes three arguments—a memory location (*loc*), an old value (*oldV*) and a new value (*newV*). If the value currently stored at *loc* is equal to *oldV* it atomically updates *newV* at *loc* and returns *true*. Otherwise, *loc* is not modified and the CAS returns *false*. A ***writeMin*** is an atomic instruction that takes two arguments—a memory location (*loc*) and an old value (*val*), and atomically updates the value stored at *loc* to be the minimum of the stored value and *val*, returning *true* if the stored value was atomically updated and *false* otherwise. We assume that both CAS and writeMin take $O(1)$ work and note that both primitives are very efficient in practice [52].

The following parallel procedures are used throughout the paper. ***Scan*** takes as input an array X 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 $(\perp, \perp \oplus X[0], \perp \oplus X[0] \oplus X[1], \ldots, \perp \oplus_{i=0}^{n-2} X[i])$ as well as the overall sum, $\perp \oplus_{i=0}^{n-1} X[i]$. Scan can be done in $O(n)$ work and $O(\log n)$ depth (assuming $\oplus$ takes $O(1)$ work) [25]. ***Reduce*** takes an array A and a binary associative function f and returns the "sum" of elements with respect to f. ***Filter*** takes an array A and a function f returning a boolean and returns a new array containing $e \in A$ for which $f(e)$ is true, in the same order as in A. Both reduce and filter can be done in $O(n)$ work and $O(\log n)$ depth (assuming f takes $O(1)$ work). A ***semisort*** takes an input array of elements, where each element has an associated key and

reorders the elements so that elements with equal keys are contiguous, but elements with different keys are not necessarily ordered. The purpose is to collect equal keys together, rather than sort them. A semisort can be done in $O(n)$ expected work and $O(c \log n)$ depth with probability $1-1/n^c$ (i.e., ***with high probability (w.h.p.)***) [23].

2.1 Ligra Framework

In this section, we review the Ligra framework for shared-memory graph processing [51]. Ligra provides data structures for representing a graph $G = (V, E)$, and ***vertexSubset***s (subsets of the vertices). It provides the functions VERTEXMAP, used for mapping over vertices, and EDGEMAP, used for mapping over edges. ***VERTEXMAP*** takes as input a vertexSubset U and a function F returning a boolean. It applies F to all vertices in U and returns a vertexSubset containing $U' \subseteq U$ where. $u \in U'$ if and only if $F(u) = true$. F can side-effect data structures associated with the vertices. ***EDGEMAP*** takes as input a graph $G(V, E)$, a vertexSubset U, and two functions F and C which both return a boolean. EDGEMAP applies F to $(u, v) \in E$ s.t. $u \in U$ and $C(v) = true$ (call this subset of edges E_a), and returns a vertexSubset U' where $u \in V$ if and only if $(u, v) \in E_a$ and $F(u, v) = true$. As in VERTEXMAP, F can side-effect data structures associated with the vertices.

Additional Primitives. We add several primitives to Julienne in addition to those provided by Ligra that simplify the expression of our algorithms. We include an option type MAYBE(**T**). We extend the vertexSubset data structure to allow vertices in the subset to have associated values. We denote a vertexSubset with associated value type **T** as ***vertexSubset***$_{\mathbf{T}}$. A vertexSubset$_T$ can be supplied to any functions that accept a vertexSubset. We also add a function call operator to vertexSubset which returns a (vertex, data) pair.

We provide a new primitive, ***EDGEMAPREDUCE***, which takes a graph G, vertexSubset S, a map function $M : \mathbf{vtx} \rightarrow \mathbf{T}$, an associative and commutative reduce function $R : \mathbf{T} \times \mathbf{T} \rightarrow \mathbf{T}$, and an update function $U : \mathbf{vtx} \times \mathbf{T} \rightarrow$ MAYBE(**O**), and returns a vertexSubset$_O$. EDGEMAPREDUCE performs the following logic common to many graph algorithms: M is applied to each neighbor of S in parallel. The mapped values are reduced to a single value per neighbor using R (in an arbitrary ordering since R is associative and commutative). Finally, U is called on the neighboring vertex v and the reduced value for v. The output is a vertexSubset$_{\mathbf{O}}$, where all vertices for which U returned NONE are filtered out. In our applications, we use ***EDGEMAPSUM***, which specializes M to 1 and R to SUM.

We provide a primitive, ***EDGEMAPFILTER***, which takes a graph G, vertexSubset U, and a predicate P, and outputs a vertexSubset$_{\mathbf{int}}$, where each vertex $u \in U$ has an associated count for the number of neighbors that satisfied P. EDGEMAPFILTER also takes an optional parameter PACK which lets applications remove edges to all neighbors that do not satisfy P by mutating G.

3 BUCKETING

The bucket structure maintains a dynamic mapping from identifiers to bucket_ids. The purpose of the structure is to provide efficient access to the inverse map—given a bucket_id, b, retrieve all identifiers currently mapped to b.

3.1 Interface

The bucket structure uses several types that we now define. An **identifier** is a unique integer representing a bucketed object. An identifier is mapped to a **bucket_id**, a unique integer for each bucket. The order that buckets are traversed in is given by the **bucket_order** type. **bucket_dest** 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 bucket_ids and O, a bucket_order. Initially, some identifiers may not be mapped to a bucket, so we add NULLBKT, a special bucket_id which lets D indicate this. Buckets in the structure are accessed monotonically in the order specified by O. While the interface can easily be modified to support random-access to buckets, we do not know of any algorithms that require it. Although we currently only use identifiers to represent vertices, our interface is not specific to storing and retrieving vertices, and may have applications other than graph algorithms. Even in the context of graphs, we envision algorithms where identifiers represent other objects such as edges, triangles, or graph motifs.

After the structure is created, NEXTBUCKET can be used to access the next non-empty bucket in non-decreasing (resp. non-increasing) order while UPDATEBUCKETS updates the bucket_ids for multiple identifiers. To iterate through the buckets, the structure internally maintains a variable CUR which stores the value of the current bucket being processed. Note that the CUR bucket can potentially be returned more than once by NEXTBUCKET if identifiers are inserted back into CUR. The GETBUCKET primitive is how users indicate that an identifier is moving buckets. We added this primitive to allow implementations to perform certain optimizations without extra involvement from the user. We describe these optimizations and present the rationale for the GETBUCKET primitive in Section 3.3.

The full list of functions is therefore:

- **MAKEBUCKETS**(n : *int*,
  D : identifier $\mapsto$ bucket_id
  O : bucket_order : buckets
  Creates a bucket structure containing n identifiers in the range $[0, n)$ where the bucket_id for identifier i is $D(i)$. The structure iterates over the buckets in order O which is either INCREASING or DECREASING.
- **NEXTBUCKET**() : (bucket_id, identifiers)
  Returns the bucket_id of the next non-empty bucket and the set of identifiers contained in it. When no identifiers are left in the bucket structure, the pair (NULLBKT, {}) is returned.
- **GETBUCKET**(PREV : bucket_id,
  NEXT : bucket_id) : bucket_dest
  Computes a bucket_dest for an identifier moving from bucket_id PREV to NEXT. Returns NULLBKT if the identifier does not need to be updated, or if NEXT< CUR.
- **UPDATEBUCKETS**(F : *int* $\mapsto$ (identifier, bucket_dest),
  k : *int*)
  Updates k identifiers in the bucket structure. The i'th identifier and its bucket_dest are given by $F(i)$.

3.2 Algorithms

We first discuss a sequential algorithm implementing the interface and analyze its cost. The sequential algorithm shares the same underlying ideas as the parallel algorithm, so we go through it in some detail. Both algorithms in this section represent buckets exactly and so the bucket_dest and bucket_id types are identical (in particular GETBUCKET just returns NEXT).

Sequential Bucketing. We represent each bucket using a dynamic array, and the set of buckets using a dynamic array B (B_i is the dynamic array for bucket i). For simplicity, we describe the algorithm in the case when buckets are processed in INCREASING order. The structure is initialized by computing the initial number of buckets by iterating over D and allocating a dynamic array of this size. Next, we iterate over the identifiers, inserting identifier i into bucket $B_{D(i)}$ if $D(i)$ is not NULLBKT, resizing if necessary. Updates are handled lazily. When UPDATEBUCKETS is called, we leave the identifier in B_{PREV} and just insert it into B_{NEXT}, opening new buckets if NEXT is outside the current range of buckets. As discussed in Section 3.1, buckets are extracted by maintaining a variable CUR which is initially the first bucket. When NEXTBUCKET is called, we check to see whether B_{CUR} is empty. If it is, we increment CUR and repeat. Otherwise, we compact B_{CUR}, only keeping identifiers $i \in B_{\text{CUR}}$ where $D(i) = \text{CUR}$, and return the resulting set of identifiers if it is nonempty, and repeat if it is empty.

We now discuss the total work done by the sequential algorithm. The work done by initialization is $O(n + T)$ work where T is the largest bucket used by the structure, as T is an upper bound on the number of buckets when the structure was initialized. Now, suppose the structure receives K calls to UPDATEBUCKETS after being initialized, each of which updates a set S_i of identifiers where $0 \leq i < K$. By amortizing the cost of creating new buckets against T, and noticing that each update that didn't create a new bucket can be done in $O(1)$ work, the total work across all calls to UPDATEBUCKETS is $O(T + \sum_{i=0}^{K} |S_i|)$.

We now argue that the total work done over all calls to NEXTBUCKET is also $O(T + \sum_{i=0}^{K} |S_i|)$. If CUR is empty, we increment it and repeat, which can happen at most T times. Otherwise, there are some number of identifiers $i \in A_{\text{CUR}}$. By charging each identifier, which can either be dead ($D(i) \neq \text{CUR}$) or live ($D(i) == \text{CUR}$), to the operation that inserted it into the current bucket, we obtain the bound. Summing the work for each primitive gives the following lemma.

LEMMA 3.1. *The total work performed by sequential bucketing when there are n identifiers, T total buckets and K calls to* UPDATEBUCKETS *each of which updates a set S_i of identifiers is $O(n + T + \sum_{i=0}^{K} |S_i|)$.*

As discussed in Section 3.1 a given bucket can be returned multiple times by NEXTBUCKET, and the same identifiers can be reinserted into the structure multiple times using UPDATEBUCKETS, so the total work of the bucket structure can potentially be much larger than $O(n)$. Some of our applications have the property that $\sum_{i=0}^{K} |S_i| = O(m)$, while also bounding T, the total number of buckets, as $O(n)$. For these applications, the cost of using the bucket-structure is $O(m + n)$.

Parallel Bucketing. In this section we describe a work-efficient parallel algorithm for our interface. The algorithm performs initialization, K calls to UPDATEBUCKETS, and L calls to NEXTBUCKET in the same work as the sequential algorithm and $O((K + L) \log n)$ depth w.h.p. As before, we maintain a dynamic array B of buckets. We initialize the structure by calculating the number of initial buckets in parallel using reduce in $O(n)$ work and $O(\log n)$ depth and allocating a dynamic array containing the initial number of buckets. Inserting identifiers into B can be done by then calling UPDATEBUCKETS(D, n). NEXTBUCKET performs a filter to keep $i \in A_{\text{CUR}}$ with $D(i) == \text{CUR}$ in parallel which can be done in $O(k)$ work and $O(\log k)$ depth on a bucket containing k identifiers.

We now describe our parallel implementation of UPDATEBUCKETS, which on a set of k updates inserts the identifiers into their new buckets in $O(k)$ expected work and $O(\log n)$ depth w.h.p. The key to achieving these bounds is a work-efficient parallel *semisort* (as described in Section 2).

Our algorithm first creates an array of (identifier, bucket_id) pairs and then calls the semisort routine, using bucket_ids as keys. The output of the semisort is an array of (identifier, bucket_id) pairs where all pairs with the same bucket_id are contiguous. Next, we map an indicator function over the semisorted pairs which outputs 1 if the index is the start of a distinct bucket_id and 0 otherwise. We then pack this mapped array to produce an array of indices corresponding to the start of each distinct bucket. Both steps can be done in $O(k)$ work and $O(\log k)$ depth. Using the offsets, we calculate the number of identifiers moving to each bucket and, in parallel, resize all buckets that have identifiers moving to them. Because all identifiers moving to a particular bucket are stored contiguously in the output of the semisort, we can simply copy them to the newly resized bucket in parallel.

Semisorting the pairs requires $O(k)$ expected work and $O(\log n)$ depth w.h.p. As in the sequential algorithm, the expected work done by K calls to UPDATEBUCKETS where the i'th call updates a set S_i of identifiers is $O(\sum_{i=0}^{K} |S_i|)$. Finally, because each substep of the routine requires at most $O(\log n)$ depth, each call to UPDATEBUCKETS runs in $O(\log n)$ depth w.h.p. As NEXTBUCKET also runs in $O(\log n)$ depth, we have that a total of K calls to UPDATEBUCKETS, and L calls to NEXTBUCKET runs in $O((K + L) \log n)$ depth w.h.p. This gives the following lemma.

LEMMA 3.2. *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 w.h.p.*

3.3 Optimizations

In practice, while many of our applications initialize the bucket structure with a large number of buckets (even $O(n)$ buckets), they only process a small fraction of them. In other applications like wBFS, the number of buckets needed by the algorithm is initially unknown. However, as the eccentricity of Web graphs and social networks tends to be small, few buckets are usually needed [58].

To make our code more efficient in situations where few buckets are being accessed, or identifiers are moved many times, we let the user specify a parameter n_B. We then only represent a range of n_B buckets (initially the first n_B buckets), and store identifiers

in the remaining buckets in an 'overflow' bucket. We only move an identifier that is logically moving from its current bucket to a new bucket if its new bucket is in the current range, or if it is not yet in any bucket. This optimization is enabled by the GETBUCKET primitive, which has the user supply both the current bucket_id and next bucket_id for the identifier. Once the current range is finished, we remove identifiers in the overflow bucket and insert them back into the structure, where the n_B buckets are now used to represent the next range of n_B buckets in the algorithm.

The main benefit of this optimization is a potential reduction in the number of identifiers UPDATEBUCKETS must move as a small value of n_B can cause most of the movement to occur in the overflow bucket. We tried supporting this implementation strategy without requiring the GETBUCKET primitive by having the bucket structure maintain an extra internal mapping from identifiers to bucket_ids. However, we found that the cost of maintaining this array of size $O(n)$ was significant (about 30% more expensive) in our applications, due to the cost of an extra random-access read and write per identifier in UPDATEBUCKETS.

Additionally, while implementing UPDATEBUCKETS using a semisort is theoretically efficient, we found that it was slow in practice due to the extra data movement that occurs when shuffling the updates. Instead, our implementation of UPDATEBUCKETS directly writes identifiers to their destination buckets and avoids the shuffle phase. We first break the array of updates into n/M blocks of length M (we set M to 2048 in our implementation). Next, we count the number of identifiers going to each bucket in each block and store these per-block histograms in an array. We then scan the array with a stride of n_B to compute the total number of identifiers moving to each bucket and resize the buckets. Finally, we iterate over each block again, compute a unique offset into the target bucket using the scanned value, and insert the identifier into the target bucket at this location. The total depth of this implementation of UPDATEBUCKETS is $O(M + \log n)$ as each block is processed sequentially and the scan takes $O(\log n)$. For small values of n_B (our default value is 128), we found that this implementation is much faster than a semisort.

3.4 Performance

In this section we study the performance of our parallel implementation of bucketing on a synthetic workload designed to simulate how our applications use the bucket structure.

Experimental Setup. We run 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.

Microbenchmark. The microbenchmark simulates the behavior of a bucketing-based algorithm such as k-core and Δ-stepping. On each round, these applications extract a bucket containing a set S of identifiers (vertices), and update the buckets for identifiers in $N(S)$. The microbenchmark simulates this behavior on a degree-8 random graph. Given two inputs, b, the number of initial buckets, and n, the number of identifiers, it starts by bucketing the identifiers uniformly at random and iterating over the buckets in INCREASING order. On each round, it extracts a set S of identifiers and for each extracted identifier, it picks 8 randomly chosen neighbors $\{v_0, \ldots, v_7\}$, checks whether the bucket for v_i is greater than CUR, and if so updates its bucket to $\max(\text{CUR}, D(v_i)/2)$. If $D(v_i) \leq \text{CUR}$, it sets v_i's bucket to NULLBKT which ensures that identifiers extracted from the bucket structure are never reinserted.

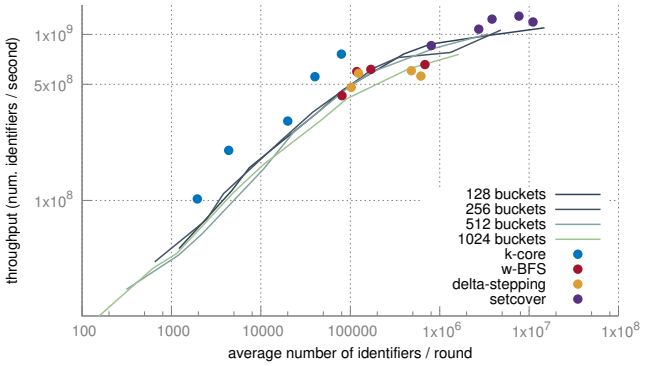


Figure 1: Log-log plot of throughput (billions of identifiers per second) vs. average number of identifiers processed per round.

We profile the performance of the bucket structure while varying b, the number of buckets. As our applications request at most about 1000 buckets, we run the microbenchmark to see how it performs when b is in the range $[128, 256, 512, 1024]$. For a given number of buckets, we vary the number of identifiers to generate different data points. The throughput of the bucket structure is calculated as the total number of identifiers extracted by NEXTBUCKET, plus the number of identifiers that move from their current bucket to a new bucket. Because identifiers moving to the NULLBKT-bucket are inexpensively handled by the bucket structure, (such requests are ignored by UPDATEBUCKETS and do not incur any random reads or writes) we exclude these requests from our total count.

We plot the throughput achieved by the structure vs. the average number of identifiers per round in Figure 1. The average number of identifiers per round is the total number of identifiers that are extracted and updated, divided by the number of rounds required to process all of the buckets. Using this data, we calculated the peak throughput supported by the bucket structure, and the half-performance length[2] which are approximately 1 billion identifiers per second, and an average of 500,000 identifiers per round, respectively.

Applications. We also plot points corresponding to the throughput and average number of identifiers per round achieved by our applications when run on our graphs in Figure 1. We observe that the benchmark throughput is a useful guideline for throughput achieved by our applications. We note that the average number of identifiers per round in k-core is noticeably lower than our other applications—this is because of the large number of rounds necessary to compute the coreness of each vertex using the peeling algorithm in our graphs (up to about 130,000). We discuss more details about our algorithms in Section 4 and their performance in Section 5.

[2]The number of identifiers when the system achieves half of its peak performance.

4 APPLICATIONS

In this section, we describe four bucketing-based algorithms and discuss how our framework can be used to produce theoretically efficient implementations of them.

4.1 k-core and Coreness

A k-core of an undirected graph G is a maximal connected subgraph where every vertex has induced-degree at least k. k-cores are widely studied in the context of data mining and social network analysis because participation in a large k-core is indicative of the importance of a node in the graph. The coreness problem is to compute for each $v \in V$ the maximum k-core v is in. We call this value the *coreness* of a vertex and denote it as $\lambda(v)$.

The notion of a k-core was introduced independently by Seidman [48], and by Matula and Beck [37] (who used the term k-linkage) and identifies the subgraphs of G that satisfy the induced degree property as the k-cores of G. Anderson and Mayr showed that the decision problem for k-core can be solved in NC for $k \leq 2$, but is P-complete[3] for $k \geq 3$ [3]. Since being defined, k-cores and coreness values have found many applications from graph mining, network visualization, fraud detection, and studying biological networks [2, 50, 60].

Matula and Beck give the first algorithm which computes all coreness values. Their algorithm bucket-sorts vertices by their degree, and then repeatedly deletes the minimum-degree vertex. The affected neighbors are then moved to a new bucket corresponding to their induced degree. The total work of their algorithm is $O(m+n)$. Batagelj and Zaversnik (BZ) give an implementation of the Matula-Beck algorithm that runs in the same time bounds [4].

While the sequential algorithm requires $O(m+n)$ work, all existing parallel algorithms with non-trivial parallelism take at least $O(m + k_{max}n)$ work where k_{max} is the largest core number in the graph [16, 20, 41, 44, 51]. This is because the implementations do not bucket the vertices and must scan all remaining vertices when computing each core number. Our parallel algorithm as well as some existing parallel algorithms are based on a peeling procedure, where on each iteration of the procedure, vertices below a certain degree are removed from the graph. The peeling process on random (hyper)graphs has been studied and it has been shown that $O(\log n)$ rounds of peeling suffices [1, 26], although for arbitrary graphs the number of rounds could be linear in the worst case. We note that computing a particular k-core from the coreness numbers requires finding the largest induced subgraph among vertices with coreness at least k, which can be done efficiently in parallel [14, 54].

The pseudocode for our implementation is shown in Algorithm 1. D holds the initial bucket for each vertex, which is initially its degree in G. The bucket structure is created on line 12 by supplying n, D and the INCREASING keyword, as lowest degree vertices are removed first. On line 14, the next non-empty bucket is extracted from the structure, with k updated to be the bucket id (this could be the same as the previous round if there are still vertices with that coreness number). The bucket contains all vertices with degree k. As these vertices now have their coreness set, we update *finished* with the number of vertices in the current bucket on line 15. We

[3]There is no polylogarithmic depth algorithm for this problem unless P = NC.

Algorithm 1 Coreness

```
1:  D = {deg(v_0), ..., deg(v_{n-1})}          ▷ initialized to initial degrees
2:  k = 0                                      ▷ the core number being processed
3:  procedure UPDATE(v, edgesRemoved)
4:      inducedD = D[v], newD = ∞
5:      if (inducedD > k) then
6:          newD = max(inducedD − edgesRemoved, k),  D[v] = newD
7:          bkt = B.GET_BUCKET(inducedD, newD)
8:          if (bkt ≠ NULLBKT) then
9:              return SOME(bkt)
10:     return NONE
11: procedure CORENESS(G)
12:     B = MAKEBUCKETS(G.n, D, INCREASING),  finished = 0
13:     while (finished < G.n) do
14:         (k, ids) = B.NEXTBUCKET()
15:         finished = finished + |ids|
16:         moved = EDGEMAPSUM(G, ids, UPDATE)
17:         B.UPDATEBUCKETS(moved, |moved|)
18:     return D
```

call EDGEMAPSUM on line 16, with the UPDATE function (lines 3–10) to count the number edges removed for each vertex. For a neighbor v, UPDATE updates $D[v]$. It returns a MAYBE(bucket_dest) by calling GETBUCKET on the previous induced-degree of v and the new induced-degree (if the new induced-degree falls below k, it will be set to k so that it can be placed in the current bucket). The result of EDGEMAPSUM is a $\text{vertexSubset}_{\text{bucket\_dest}}$. On line 17 we update the buckets for vertices that have changed buckets, and repeat. The algorithm terminates once all of the vertices have been extracted from the bucket structure.

We now analyze the complexity of our algorithm by plugging in quantities into Lemma 3.2. We can bound $\sum_{i=0}^{K} |S_i| \leq 2m$, as in the worst case each removed edge will cause an independent request to the bucket structure. Furthermore, the total number of buckets, T is at most n, as vertices are initialized into a bucket corresponding to their degree. Plugging these quantities into Lemma 3.2 gives us $O(m+n)$ expected work, which makes our algorithm work-efficient.

To analyze the depth of our algorithm, we define ρ to be the ***peeling-complexity*** of a graph, or the number of steps needed to peel the graph completely. A step in the peeling process removes all vertices with minimum degree, decrements the degrees of all adjacent neighbors and repeats. On graphs with peeling-complexity ρ, our algorithm runs in $O(\rho \log n)$ depth w.h.p., as each peeling-step potentially requires a call to the bucket structure to update the buckets for affected neighbors. While ρ can be as large as n in the worst-case, in practice ρ is significantly smaller than n. Our algorithm is the first work-efficient algorithm for coreness with non-trivial parallelism. The bounds are summarized in the following theorem.

THEOREM 4.1. *Our algorithm for coreness requires $O(m+n)$ expected work and $O(\rho \log n)$ depth with high probability, where ρ is the peeling-complexity of the graph.*

Our serial implementation of coreness is based on an implementation of the BZ algorithm written in Khaouid et al. [28]. We re-wrote their code in C++ and integrated it into the Ligra+ framework (an extension of Ligra that supports graph compression) [55], which lets us run our implementation on our largest graphs.

4.2 Δ-stepping and wBFS

The ***single-source shortest path (SSSP)*** problem takes as input a weighted graph $G = (V, E, w(E))$ and a source vertex *src*, and computes the shortest path distance from *src* to each vertex in V, with unreachable vertices having distance ∞. On graphs with non-negative edge weights, the problem can be solved in $O(m + n \log n)$ work by using Dijkstra's algorithm [19] with Fibonacci heaps [21]. While Dijkstra's algorithm cannot be used on graphs with negative edge-weights, the Bellman-Ford algorithm can, but at the cost of an increased worst-case work-bound of $O(mn)$ [15]. Bellman-Ford often performs very well in parallel, but is work-inefficient for graphs with only non-negative edge weights.

Both Dijkstra and Bellman-Ford work by relaxing vertices. We denote the shortest path to each vertex by *SP*. A relaxation occurs over a directed edge (u, v) when vertex u checks whether $SP(u) + w(u, v) < SP(v)$, updating $SP(v)$ to the smaller value if this is the case. In Dijkstra's algorithm, only the vertex, v, that is closest to the source is relaxed—as the graph is assumed to have non-negative edge-weights, we are guaranteed that $SP(v)$ is correct, and so each vertex only relaxes its outgoing edges once. In the simplest form of Bellman-Ford, all vertices relax their neighbors in each step, and so each step costs $O(m)$. The number of steps needed for Bellman-Ford to converge is proportional to the largest number of hops in a shortest path from *src* to any $v \in V$, which can be as large as $O(n)$.

Weighed breadth-first search (wBFS) is a version of Dijkstra's algorithm that works well for small integer edge weights and low-diameter graphs [18]. As described in Section 1, wBFS keeps a bucket for each possible distance and goes through them one by one from the lowest. Each bucket acts like a frontier as in BFS, but when we process a vertex v in a frontier i instead of placing its unvisited neighbors in the next frontier $i+1$ we place each neighbor u in the bucket $i + d(v, u)$. wBFS turns out to be a special case of Δ-stepping, and hence we return to it later.

The Δ-stepping algorithm provides a way to trade-off between the work-efficiency of Dijkstra's algorithm and the increased parallelism of Bellman-Ford [40]. In Δ-stepping, computation is broken up into a number of steps. On step i, vertices in the annulus at distance $[i\Delta, (i + 1)\Delta)$ are relaxed until no further distances change. The algorithm then proceeds to the next annulus, repeating until the shortest-path distances for all reachable vertices are set. Note that when $\Delta = \infty$, this algorithm is equivalent to Bellman-Ford.

While Bellman-Ford is easy to implement in parallel, previous work has identified the difficulty in producing a scalable implementation of bucketing [24], which is required in the Δ-stepping algorithm [40]. Due to the difficulty of bucketing in parallel, many implementations of SSSP in graph-processing frameworks use the Bellman-Ford algorithm [22, 51]. Implementations of Δ-stepping do exist, but the algorithms are not easily expressed in existing frameworks, so they are either provided as primitives in a graph processing framework [42, 59] or are stand-alone implementations [6, 17, 24, 34, 35]. There are other parallel algorithms for SSSP, but for some of the algorithms, there is low parallelism [11, 43], and for others no parallel implementations exist [8, 13, 29, 49, 56]. Note that there is currently no parallel algorithm for single-source shortest paths with non-negative edge weights that matches the work of the sequential algorithm and has polylogarithmic depth. Our bucketing interface allows us to give a simple implementation of Δ-stepping with work matching that of the original algorithm [40].

Algorithm 2 Δ-stepping

```
1:  SP = {∞, ..., ∞}                                   ▷ initialized to all ∞
2:  Fl = {0, ..., 0}                                   ▷ initialized to all 0
3:  procedure GetBucketNum(i) return ⌊SP[i]/Δ⌋
4:  procedure Update(s, d, w)
5:      nDist = SP[s] + w,  oDist = SP[d],  res = None
6:      if (nDist < oDist) then
7:          if (CAS(&Fl[d], 0, 1) then
8:              res = Some(oDist)   ▷ the distance at the start of this round
9:          writeMin(&SP[d], nDist)
10:     return res
11: procedure Reset(v, oldDist)
12:     Fl[v] = 0,  newDist = SP[d]
13:     return B.get_bucket(⌊oldDist/Δ⌋, ⌊newDist/Δ⌋)
14: procedure Δ-stepping(G, Δ, src)
15:     SP[r] = 0
16:     B = makeBuckets(G.n, GetBucketNum, increasing)
17:     while ((id, ids) = B.nextBucket() and id ≠ nullbkt) do
18:         Moved = edgeMap(G, ids, Update)
19:         NewBuckets = vertexMap(Moved, Reset)
20:         B.updateBuckets(NewBuckets, |NewBuckets|)
21:     return SP
```

The pseudocode for our implementation is shown in Algorithm 2. Shortest-path distances are stored in an array *SP*, which are initially all ∞, except for the source, *src* which has an entry of 0. We also maintain an array of flags, *Fl*, which are used by edgeMap to remove duplicates. The bucket structure is created by specifying n, *SP*, and the keyword increasing (line 16). The i'th bucket represents the annulus of vertices between distance $[i\Delta, (i + 1)\Delta)$ from the source. Each Δ-step processes the closest unfinished annulus and so the buckets are processed in increasing order. On line 17 we extract the next bucket, and terminate if it is nullbkt. Otherwise, we explore the outgoing edges of the set of vertices in the bucket using edgeMap. In the Update function passed to edgeMap (lines 4–10), a neighboring vertex, d, is visited over the edge (s, d, w). s checks whether it relaxes d, i.e., $SP[s] + w < SP[d]$. If it can, it first uses a CAS to test whether it is the unique neighbor of d that read its value before any modifications in this round (line 7) setting this distance to be the return value (line 8) if the CAS succeeds. s then uses an atomic writeMin operation to update the distance to d (line 9). Unsuccessful visitors return None, which signals that they did not capture the old value of d. The result of edgeMap is a vertexSubset where the value stored for each vertex is the distance before any modifications in this round.

Next, we call vertexMap (line 19), which calls the Reset function (lines 11–13) on each visited neighbor, v, that had its distance updated. Reset first resets the flag for v (line 12) to enable v to be correctly visited again on a future round. It then calculates the new bucket for v (line 13) and returns this value. The output is another vertexSubset called *NewBuckets* containing the neighbors and their new buckets. Finally, on line 20, we update the buckets containing each neighbor that had its distance lowered, by calling UpdateBuckets on the vertexSubset *NewBuckets*. We repeat these steps until the bucket structure is empty. While we describe visitors from the current frontier CAS'ing values in a separate array of flags, *Fl*,

our actual implementation uses the highest-bit of SP to represent Fl, as this reduces the number of random-memory accesses and improves performance in practice.

The original description of Δ-stepping by Meyer and Sanders [40] separates edges into *light edges* and *heavy edges*, where light edges are of length at most Δ. Inside each annulus, light edges may be processed multiple times but heavy edges only need to be processed once, which reduces the amount of redundant work. We implemented this optimization but did not find a significant improvement in performance for our input graphs. Note that this optimization can fit into our framework by creating two graphs, one containing just the light edges and the other just the heavy edges. Light edges can be processed multiple times until the bucket number changes, at which point we relax the heavy edges once for the vertices in the bucket.

We will now argue that our implementation of Δ-stepping (with the light-heavy edge optimization) does the same amount of work as the original algorithm. The original algorithm takes at most $(d_c/\Delta)l_{\max}$ rounds to finish, where d_c is the maximum distance in the graph and $l_{\max}$ is the maximum number of light edges in a path with total weight at most Δ. Our implementation takes the same number of rounds to finish because we are relaxing exactly the same vertices as the original algorithm on each round. Using our work-efficient bucketing implementation, by Lemma 3.2 the work per round is linear in the number of vertices and outgoing edges processed, which matches that of the original algorithm. The depth of our algorithm is $O(\log n)$ times the number of rounds w.h.p.

When edge weights are integers, and $\Delta = 1$, Δ-stepping becomes wBFS. This is because there can only be one round within each step. In this case we have the following strong bound on work-efficiency.

Theorem 4.2. *Our algorithm for wBFS (equivalent to Δ-stepping with integral weights and $\Delta = 1$) when run on a graph with m edges and eccentricity r_{src} from the source src, runs in $O(r_{src} + m)$ expected work and $O(r_{src} \log n)$ depth w.h.p.*

Proof. The work follows directly from the fact we do no more work than the sequential algorithm, charging only $O(1)$ work per bucket insertion and removal, which is proportional to the number of edges (every edge does at most one insertion and is later removed). The depth comes from the number of rounds and the fact that each round takes $O(\log n)$ depth w.h.p. for the bucketing. □

4.3 Approximate Set Cover

The ***set cover*** problem takes as input a universe $\mathcal{U}$ of ground elements, $\mathcal{F}$ a collection of sets of $\mathcal{U}$ s.t. $\bigcup \mathcal{F} = \mathcal{U}$ and a cost function $c : \mathcal{F} \to \mathbb{R}_+$. The problem is to find the cheapest collection of sets $\mathcal{A} \subseteq \mathcal{F}$ that covers U, where the cost of a solution $\mathcal{A}$ is $c(\mathcal{A}) = \sum_{S \in \mathcal{A}} c(S)$. This problem can be modeled as a bipartite graph where sets and elements are vertices, with an edge connecting a set to an element if and only if the set covers that element. Finding the cheapest collection of sets is an NP-complete problem, and a sequential greedy algorithm [27] gives a H_n-approximation, where $H_n = \sum_{k=1}^{n} 1/k$, in $O(m)$ work for unweighted sets and $O(m \log m)$ work for weighted sets, where m is the sum of the sizes of the sets, or equivalently the number of edges in the bipartite graph. Parallel algorithms have been designed for approximating the set cover [7, 9, 10, 12, 30, 45, 57], and Blelloch et al. [9] present a work-efficient parallel algorithm for the problem, which takes $O(m)$ work and $O(\log^3 m)$ depth, and gives a $(1+\epsilon)H_n$-approximation to the set cover problem. Blelloch et al. [10] later present a multicore implementation of the parallel set cover algorithm. Their code, however, is special-purpose, not being part of any general framework, and is not work-efficient. In this section, we give a work-efficient implementation of their algorithm using our bucketing interface, and we compare the performance of the codes in Section 5.

Algorithm 3 Approximate Set Cover

```
1:  El = {∞, ..., ∞}                                   ▷ initialized to all ∞
2:  Fl = {0, ..., 0}                                   ▷ initialized to all 0
3:  D = {deg(v_0), ..., deg(v_{n-1})}                  ▷ initialized to initial out-degrees
4:  b                                                  ▷ the current bucket number
5:  procedure BucketNum(s) return ⌊log_{1+ε} D[s]⌋
6:  procedure ElmUncovered(e) return Fl[e] == 0
7:  procedure UpdateD(s, d) D[s] = d
8:  procedure AboveThreshold(s, d) return d >= ⌈(1 + ε)^{max(b,0)}⌉
9:  procedure WonElm(s, e) return s == El[e]
10: procedure InCover(s) return D[s] == ∞
11: procedure VisitElms(s, e) writeMin(&El[e], s)
12: procedure WonEnough(s, elmsWon)
13:     threshold = ⌈(1 + ε)^{max(b-1,0)}⌉
14:     if (elmsWon > threshold) then
15:         D[s] = ∞                                   ▷ puts s in the set cover
16: procedure ResetElms(s, e)
17:     if (El[e] == s) then
18:         if (InCover(s)) then
19:             Fl[e] = 1                              ▷ e is covered by s
20:         else
21:             El[e] = ∞                              ▷ reset e
22: procedure SetCover(G = (S ∪ E, A))
23:     B = makeBuckets(|S|, BucketNum, decreasing)
24:     while ((b, Sets) = B.nextBucket() and b ≠ nullbkt) do
25:         SetsD = edgeMapFilter(G, Sets, ElmUncovered, Pack)
26:         vertexMap(SetsD, UpdateD)
27:         Active = vertexFilter(SetsD, AboveThreshold)
28:         edgeMap(G, Active, VisitElms, ElmUncovered)
29:         ActiveCts = edgeMapFilter(G, Active, WonElm)
30:         vertexMap(ActiveCts, WonEnough)
31:         edgeMap(G, Active, ResetElms)
32:         Rebucket = {(s, B.get_bucket(b, BucketNum(s)) |
                s ∈ Sets and not InCover(s)}
33:         B.updateBuckets(Rebucket, |Rebucket|)
34:     return {i | InCover(i) == true}
```

The Blelloch et al. algorithm works by first bucketing all sets based on their cost. In the weighted case, the algorithm first ensures that the ratio between the costliest set and cheapest set is polynomially bounded, so that the total number of buckets is kept logarithmic (see Lemma 4.2 of [10]). It does this by discarding sets that are costlier than a threshold, and including sets cheaper than another threshold in the cover. The remaining sets are bucketed based on their normalized cost (the cost per element). In order to guarantee polylogarithmic depth, only $O(\log m)$ buckets are maintained, with a set having cost C going into bucket $\left\lfloor \log_{1+\epsilon} C \right\rfloor$. The main loop of the algorithm iterates over the buckets from the least to most costly bucket. Each step invokes a subroutine to compute

a maximal nearly-independent set (MaNIS) of sets in the current bucket. MaNIS computes a subset of the sets in the current bucket that are almost non-overlapping in the sense that each set chosen by MaNIS covers many elements that are not covered by any other chosen set. For sets not chosen by MaNIS, the number of uncovered elements they cover is shrunk by a constant factor w.h.p. We refer the reader to the original paper for proofs on both MaNIS and the set cover algorithm. We now describe our algorithm for unweighted set cover, and note that it can be easily modified for the weighted case as well.

The pseudocode for our implementation of the Blelloch et al. algorithm is shown in Algorithm 3. We assume that the set cover instance is represented as an undirected bipartite graph with sets and elements on opposite sides. The array *El* contains the set each element is assigned to. The array *Fl* specifies whether elements are covered ($Fl[e] = 0$ if and only if e is uncovered). Initially all elements are not covered (lines 1–2). The array D contains the number of remaining elements covered by each set (line 3). As sets are represented by vertices, each entry of D is initially just the degree of that vertex. b stores the current bucket id (line 4), which is updated on line 24 when we extract the next bucket. The bucket structure is created by specifying $n = |S|$, BucketNum, and the keyword decreasing (line 23), as we process sets in decreasing order based on the number of uncovered elements they cover.

Each round starts by extracting the next non-empty bucket (line 24). The degrees of sets are updated lazily, so the first phase of the algorithm packs out edges to covered elements and computes the sets that still cover enough elements to be active in this round. On line 25, we call edgeMapFilter with the function ElmUncovered and the option Pack, which packs out any covered elements in the sets' adjacency lists and updates their degrees. The return value of edgeMapFilter is a vertexSubset$_{\textbf{int}}$ (*SetsD*), where the associated value with each set is its new degree. On line 26 we apply vertexMap over *SetsD* with the function UpdateD, which updates D with the new degrees. Finally, we call vertexFilter with the function AboveThreshold to compute the vertexSubset, *Active*, which is the subset of *SetsD* that still have sufficient degree.

The next phase of the algorithm implements one step of MaNIS. Note that instead of implementing MaNIS as a separate subroutine, we implicitly compute it by fusing the loop that computes a MaNIS with the loop that iterates over the buckets. On line 28, active sets reserve uncovered elements using an edgeMap, breaking ties based on their IDs using writeMin. edgeMap checks whether a neighboring element is uncovered using ElmUncovered (line 6), and if so calls VisitElms (line 11), which uses a writeMin to atomically update the parent of e. Next, we compute a vertexSubset, *ActiveCts*, by calling edgeMapFilter with the function WonElm (line 9). The value associated with each set in *ActiveCts* is the number of elements successfully reserved by it. We then apply vertexMap over *ActiveCts* (line 30) with the function WonEnough (lines 12–15), which checks whether the number of elements reserved is above a threshold (line 13), and if so updates the set to be in the cover.

The last phase of the algorithm marks elements that are newly covered, resets elements whose sets did not make it into the cover, and finally reinserts sets that did not make it into the cover back into the bucket structure. On line 31, we call edgeMap with the supplied function ResetElms (lines 16–21) which first checks that s is the set which reserved e (line 17). If s joined the cover, then we mark e as covered (line 19). Otherwise, we reset $El[e] = \infty$ (line 21) so that e can be correctly visited on future rounds. Finally, we compute *Rebucket*, a vertexSubset containing the sets that did not join the cover in this round, where the value associated with each set is its bucket_dest. The bucket structure is updated with the sets in *Rebucket* on line 33. Finally, after all rounds are over, we return the subset of sets whose ids are in the cover (line 34).

Input Graph	Num. Vertices	Num. Edges	ρ
com-Orkut	3,072,627	234,370,166	5,667
Twitter	41,652,231	1,468,365,182	–
Twitter-Sym	41,652,231	2,405,026,092	14,963
Friendster	124,836,180	3,612,134,270	10,034
Hyperlink2012-Host	101,717,775	2,043,203,933	–
Hyperlink2012-Host-Sym	101,717,775	3,880,015,728	19,063
Hyperlink2012	3,563,602,789	128,736,914,167	–
Hyperlink2012-Sym	3,563,602,789	225,840,663,232	58,710
Hyperlink2014	1,724,573,718	64,422,807,961	–
Hyperlink2014-Sym	1,724,573,718	124,141,874,032	130,728

Table 2: Graph inputs, including both vertices and edges.

5 EXPERIMENTS

All of our experiments are run on the same machine configuration as in Section 3.4. The input graph sizes and peeling-complexity (for undirected graphs) that we use are shown in Table 2. ***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 [31]. ***Friendster*** is an undirected social-network graph. ***Hyperlink2012*** and ***Hyperlink2014*** are directed hyperlink graphs obtained from the WebDataCommons dataset where nodes represent web pages [39]. ***Hyperlink2012-Host*** is a directed hyperlink graph also from the WebDataCommons dataset where nodes represent a collection of web pages belonging to the same hostname. Unless mentioned otherwise, the input graph is assumed to be directed, with the symmetrized version of the graph denoted with the suffix ***Sym***.

We create weighted graphs for evaluating wBFS by selecting edge weights between $[1, \log n)$ uniformly at random. These graphs are not suitable for testing Δ-stepping, as we found that $\Delta = 1$ was always faster than a larger value of Δ. To understand the performance of our Δ-stepping implementation, we generate another family of weighted graphs with edge weights picked uniformly between $[1, 10^5)$. We successfully added edge-weights between $[1, \log n)$ to the Hyperlink2014 graph. However, due to space limitations on our machine, we were unable to store the Hyperlink2012 graph with edge-weights between $[1, \log n)$ and both the Hyperlink2012 and Hyperlink2014 graphs with edge-weights between $[1, 10^5)$. We use 'in parallel' to refer to running times using 144 hyper-threads.

***k*-core (coreness).** Table 3 shows the running time of the work-efficient implementation of k-core from Julienne and the work-inefficient implementation of k-core from Ligra. Figure 2 shows the running time of both implementations as a function of thread count. We see that our work-efficient implementation achieves between 4-41x parallel speedup over the implementation running on a single thread. Our speedups are smaller on graphs where ρ is large while n and m are relatively small, such as com-Orkut and Twitter-Sym.

We also ran the Batagelj and Zaversnik (BZ) algorithm described in Section 4.1 and found that our single-thread times are always about 1.3x faster than that of the BZ algorithm. This is because on each round we move a vertex to a new bucket just once, even if many edges are deleted from it whereas the BZ algorithm will move that vertex many times. As our algorithm on a single thread is always faster than the BZ algorithm, we report self-relative speedup, which is a lower bound on speedup over the BZ algorithm.

Unfortunately, we were unable to obtain the code for the ParK algorithm [16], which is to the best of our knowledge the fastest existing parallel implementation of k-core. Instead, we used a similar work-inefficient implementation of k-core available in Ligra. In parallel, our work-efficient implementation is between 2.6–9.2x faster than the work-inefficient implementation from Ligra. On Hyperlink2012-Sym and Hyperlink2014-Sym, the work-inefficient implementation did not terminate in a reasonable amount of time, and so we only report times for our implementation in Julienne. A recent paper also reported experimental results for a different parallel algorithm for k-core that is not work-efficient [47]. On a similar configuration to their machine our implementation is about 10x faster on com-Orkut, the largest graph they test on.

wBFS and Δ-stepping. Table 3 shows the running time of the Δ-stepping and wBFS implementations from Julienne and the GAP benchmark suite, the priority-based Bellman-Ford implementation from Galois, the Bellman-Ford implementation from Ligra and the sequential solver from the DIMACS shortest path challenge [6, 42]. Figures 3 and 4 show the running time of the four parallel implementations as a function of thread count. To the best of our knowledge, we are not aware of any existing parallel implementations of wBFS, so we test wBFS against the same implementations as Δ-stepping, setting $\Delta = 1$. We see that our work-efficient implementation achieves between 22–43x parallel speedup over the implementation running on a single thread for wBFS and between 18–32.4x parallel speedup over our implementation running on a single thread for Δ-stepping. For Δ-stepping, we found that setting $\Delta = 32768$ performed best in our experiments.

Like our implementation, the SSSP implementation in GAP does not perform the light/heavy optimization described in the original Δ-stepping paper [40]. Instead of having shared buckets, it uses thread-local bins to represent buckets. The Galois algorithm is a version of Bellman-Ford that schedules nodes based on their distance from the source (closer vertices have higher priority). Because the Galois algorithm avoids synchronizing after each annulus, it achieves good speedup on graphs with large diameter, but where paths with few hops are also likely to be the shortest paths in the graph (such as road networks). On such graphs our algorithm performs poorly due to a large amount of synchronization.

All implementations achieve good speedup with an increased number of threads. On a single thread our implementation is usually faster than the single-thread times for other implementations. This is likely because of an optimization we implemented in our EDGEMAP routine, which allows traversals to only write to an amount of memory proportional to the size of the output frontier. In parallel, while the GAP implementation usually outperforms us by a small amount, we remain very competitive, being between 1.07-1.1x slower for wBFS, and between 1.1–1.7x faster for Δ-stepping. We are between 1.6–3.4x faster than the Galois implementation on wBFS and between 1.2–2.9x faster on Δ-stepping. Our implementation is between 1.2–3.9x faster for wBFS and 1.8–5.2x faster for Δ-stepping compared to the Bellman-Ford implementation in Ligra [51]. We note that there is recent work on another parallel algorithm for SSSP [35] and based on their speedups over the Δ-stepping implementation in Galois, our Julienne implementation seems competitive. We leave a detailed comparison for future work.

Approximate Set Cover. We generated bipartite graphs to use as set cover instances by having vertices represent both the sets and the elements. Table 3 shows the running time of the work-efficient implementation of approximate set cover from Julienne and the work-inefficient implementation of approximate set cover from the PBBS benchmark suite [53]. Figure 5 shows the running time of both implementations as a function of thread count. We set ϵ to be 0.01 for both implementations. We see that our work-efficient implementation achieves between 4–35x parallel speedup over the implementation running on a single thread. Both implementations achieve poor speedup on com-Orkut, due to the relatively large number of rounds compared to the graph size. Our implementation achieves between 17–35x parallel speedup on our other test graphs.

The PBBS implementation is from Blelloch et al. [10] and implements the same algorithm as us [9]. Both implementations compute the same covers. We note that the PBBS implementation is not work-efficient. Instead of rebucketing the sets that are not chosen in a given step by using a bucket structure, it carries them over to the next step. In parallel, our times are between 1.2x slower to 2x faster compared to the PBBS implementation. On graphs like Twitter-Sym, the PBBS implementation carries a large number of unchosen sets for many rounds. In these cases, our implementation achieves good speedup over the PBBS implementation because it rebuckets these sets instead of inspecting them on each round.

6 CONCLUSION

We have presented the Julienne framework which allows for simple and theoretically efficient implementations of bucketing-based graph algorithms. Using our framework, we obtain the first work-efficient k-core algorithm with non-trivial parallelism. Our implementations either outperform or are competitive with hand-optimized codes for the same applications, and can process graphs with hundreds of billions of edges in the order of minutes on a single machine.

ACKNOWLEDGMENTS

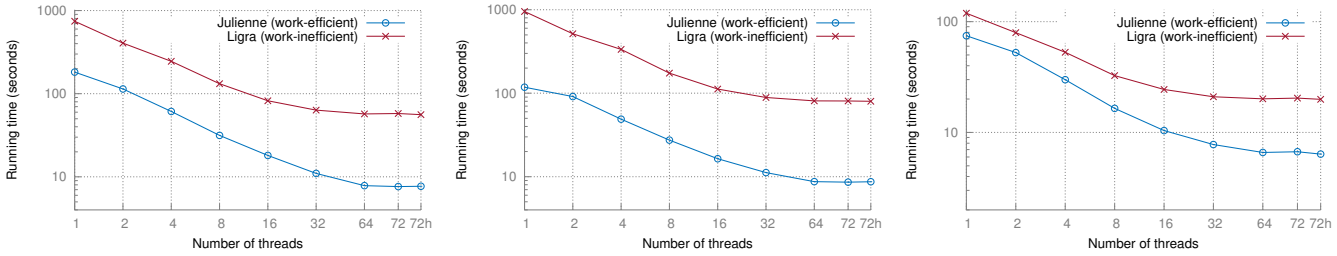
This research was supported in part by NSF grants CCF-1314590 and CCF-1533858, the Intel Science and Technology Center for Cloud Computing, and the Miller Institute for Basic Research in Science at UC Berkeley.

REFERENCES

[1] D. Achlioptas and M. Molloy. The solution space geometry of random linear equations. *Random Structures & Algorithms*, 46(2), 2015.
[2] J. I. Alvarez-Hamelin, L. Dall'asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *NIPS*. 2005.
[3] R. Anderson and E. W. Mayr. A P-complete problem and approximations to it. Technical report, 1984.

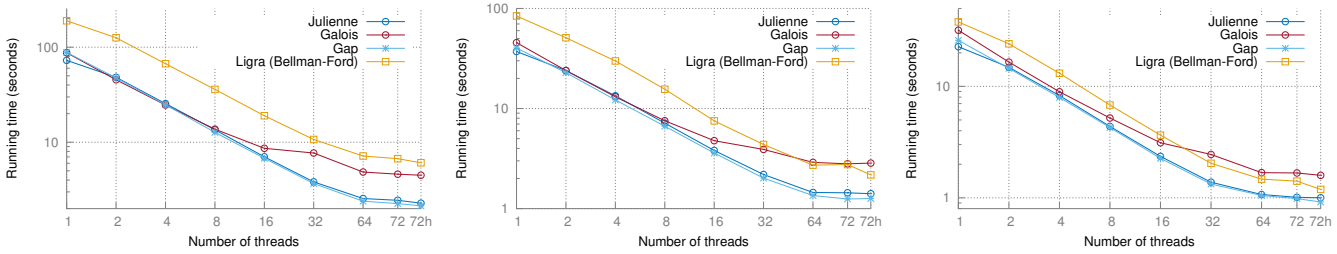
Application	com-Orkut			Twitter			Friendster			Hyperlink2012-Host			Hyperlink2012			Hyperlink2014		
	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)	(1)	(72h)	(SU)
k-core (Julienne)	**5.43**	**1.3**	4.17	**74.6**	**6.37**	11.7	**182**	**7.7**	23.6	**118**	**8.7**	13.5	**8515**	**206**	41.3	**2820**	**97.2**	29.0
k-core (Ligra)	11.6	3.35	3.46	119	19.9	5.97	745	56	13.3	953	80.1	11.9	-	-	-	-	-	-
wBFS (Julienne)*	**2.01**	0.093	21.6	**22.8**	0.987	23.1	**73.9**	2.29	32.2	37.9	1.39	27.2	-	-	-	**392**	**9.02**	43.4
Bellman-Ford (Ligra)*	4.02	0.175	22.9	37.9	1.19	31.8	190	6.08	31.2	84.2	2.17	38.8	-	-	-	2610	35.5	73.5
wBFS (GAP)*	2.35	**0.083**	28.3	25.9	**0.919**	28.1	88.1	**2.14**	41.1	40.4	**1.26**	32.0	-	-	-	-	-	-
wBFS (Galois)*	3.46	0.319	10.8	31.9	1.59	20.06	87.6	4.49	19.5	45.5	2.85	15.9	-	-	-	-	-	-
wBFS (DIMACS)*	3.488	-	-	26.54	-	-	78.19	-	-	**35.38**	-	-	-	-	-	-	-	-
Δ-stepping (Julienne)†	**3.18**	**.167**	19.0	36.3	**2.01**	18.0	112	**3.45**	32.4	49.0	**2.09**	23.4	-	-	-	-	-	-
Bellman-Ford (Ligra)†	10.2	0.423	24.1	111	3.64	30.4	613	18.2	33.6	295	7.84	37.6	-	-	-	-	-	-
Δ-stepping (GAP)†	4.33	.294	14.7	67.6	2.39	28.2	175	4.23	41.3	57.9	2.33	24.8	-	-	-	-	-	-
Δ-stepping (Galois)†	5.1	.487	10.4	64.1	2.58	24.8	122	5.56	21.9	53.8	3.17	16.9	-	-	-	-	-	-
Δ-stepping (DIMACS)†	4.44	-	-	**35.7**	-	-	**105**	-	-	55.5	-	-	-	-	-	-	-	-
Set Cover (Julienne)	**3.66**	0.844	4.33	55.4	**3.23**	17.1	165	**6.6**	25.0	93.5	**4.83**	19.3	**3720**	**104**	35.7	**1070**	**45.1**	23.7
Set Cover (PBBS)	4.47	**0.665**	6.72	**48.4**	6.71	7.21	**137**	6.86	19.9	**71.6**	8.58	8.34	-	-	-	-	-	-

Table 3: Running times (in seconds) of our algorithms over various 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 speedup of the application (single-thread time divided by 72-core time). Applications marked with $*$ and † use graphs with weights uniformly distributed in $[1, \log n)$ and $[1, 10^5)$ respectively. We display the fastest sequential and parallel time for each problem in each column in bold.



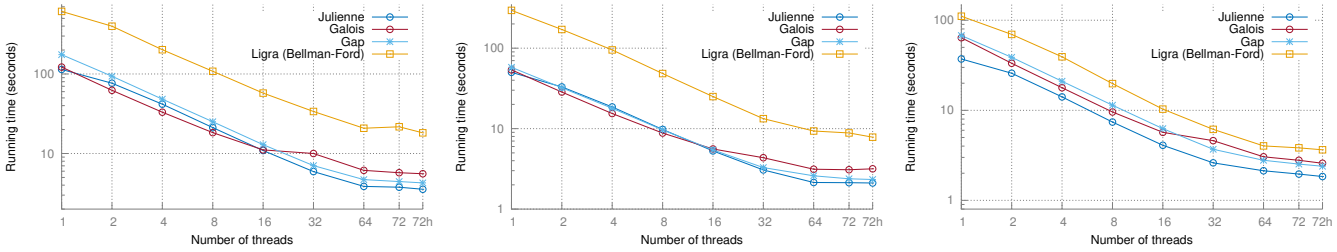
(a) Friendster **(b) Hyperlink2012-Host-Sym** **(c) Twitter-Sym**

Figure 2: Running time of k-core in seconds on a 72-core machine (with hyper-threading). "72h" refers to 144 hyper-threads.



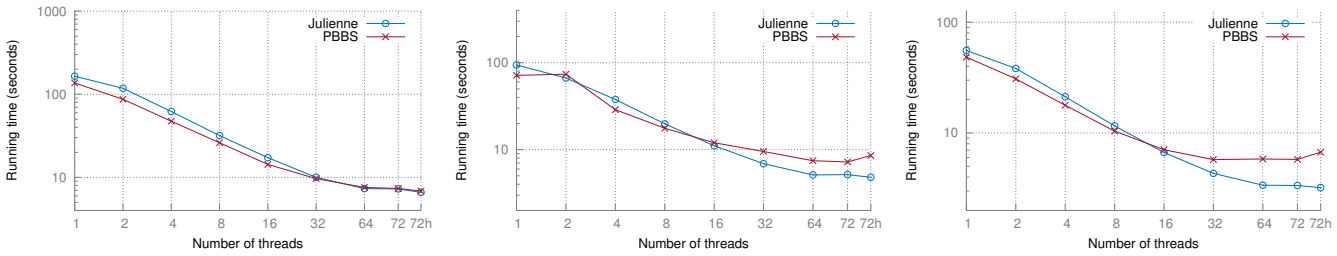
(a) Friendster **(b) Hyperlink2012-Host-Sym** **(c) Twitter-Sym**

Figure 3: Running time of wBFS in seconds on a 72-core machine (with hyper-threading). The graphs have edge weights that are uniformly distributed in $[1, \log n)$. "72h" refers to 144 hyper-threads.



(a) Friendster **(b) Hyperlink2012-Host-Sym** **(c) Twitter-Sym**

Figure 4: Running time of Δ-stepping in seconds on a 72-core machine (with hyper-threading). The graphs have edge weights that are uniformly distributed in $[1, 10^5)$. "72h" refers to 144 hyper-threads.



(a) Friendster (b) Hyperlink2012-Host-Sym (c) Twitter-Sym

Figure 5: Running time of set cover in seconds on a 72-core machine (with hyper-threading). "72h" refers to 144 hyper-threads.

[4] V. Batagelj and M. Zaversnik. An $O(m)$ algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[5] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *SC*, 2012.

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

[7] B. Berger, J. Rompel, and P. W. Shor. Efficient NC algorithms for set cover with applications to learning and geometry. *J. Comput. Syst. Sci.*, 49(3), Dec. 1994.

[8] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan. Parallel shortest paths using radius stepping. In *SPAA*, 2016.

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

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

[11] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *J. Parallel Distrib. Comput.*, 49(1), Feb. 1998.

[12] F. Chierichetti, R. Kumar, and A. Tomkins. Max-cover in map-reduce. In *WWW*, 2010.

[13] E. Cohen. Using selective path-doubling for parallel shortest-path computations. *J. Algorithms*, 22(1), Jan. 1997.

[14] 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.

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

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

[17] A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *IPDPS*, 2014.

[18] R. B. Dial. Algorithm 360: Shortest-path forest with topological ordering [H]. *Commun. ACM*, 12(11), Nov. 1969.

[19] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1), Dec. 1959.

[20] B. Elser and A. Montresor. An evaluation study of bigdata frameworks for graph processing. In *Big Data*, 2013.

[21] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3), July 1987.

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

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

[24] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *PPoPP*, 2011.

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

[26] J. Jiang, M. Mitzenmacher, and J. Thaler. Parallel peeling algorithms. *ACM Trans. Parallel Comput.*, 3(1), Jan. 2017.

[27] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. Syst. Sci.*, 9(3), 1974.

[28] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. k-core decomposition of large networks on a single PC. *Proc. VLDB Endow.*, 9(1), Sept. 2015.

[29] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2), Nov. 1997.

[30] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani. Fast greedy algorithms in mapreduce and streaming. *ACM Trans. Parallel Comput.*, 2(3), Sept. 2015.

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

[32] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8), Apr. 2012.

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

[34] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*, 2007.

[35] 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.

[36] 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.

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

[38] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *HotOS*, 2015.

[39] 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.

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

[41] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. *TPDS*, 24(2), 2013.

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

[43] R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. In *ICPP*, 1985.

[44] K. Pechlivanidou, D. Katsaros, and L. Tassiulas. MapReduce-based distributed k-shell decomposition for online social networks. In *SERVICES*, 2014.

[45] S. Rajagopalan and V. V. Vazirani. Primal-dual RNC approximation algorithms for set cover and covering integer programs. *SIAM J. Comput.*, 28(2), Feb. 1999.

[46] A. E. Sariyüce and A. Pinar. Fast hierarchy construction for dense subgraphs. *Proc. VLDB Endow.*, 10(3), Nov. 2016.

[47] A. E. Sariyuce, C. Seshadhri, and A. Pinar. Parallel local algorithms for core, truss, and nucleus decompositions. *arXiv preprint arXiv:1704.00386*, 2017.

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

[49] H. Shi and T. H. Spencer. Time-work tradeoffs of the single-source shortest paths problem. *J. Algorithms*, 30(1), Jan. 1999.

[50] K. Shin, T. Eliassi-Rad, and C. Faloutsos. CoreScope: Graph mining using k-core analysis–patterns, anomalies and algorithms. In *ICDM*, 2016.

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

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

[53] 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.

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

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

[56] T. H. Spencer. Time-work tradeoffs for parallel algorithms. *J. ACM*, 44(5), Sept. 1997.

[57] S. Stergiou and K. Tsioutsiouliklis. Set cover at web scale. In *SIGKDD*, 2015.

[58] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[59] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: a high-performance graph processing library on the GPU. In *PPoPP*, 2016.

[60] S. Wuchty and E. Almaas. Peeling the yeast protein network. *Proteomics*, 5(2), 2005.