Parallel Batch-Dynamic Algorithms for k-Core Decomposition and Related Graph Problems

Quanquan C. Liu[†]
Northwestern University
USA
quanquan@northwestern.edu

Jessica Shi
MIT CSAIL
USA
jeshi@mit.edu

Shangdi Yu
MIT CSAIL
USA
shangdiy@mit.edu

Laxman Dhulipala[†]
University of Maryland
USA
laxman@umd.edu

Julian Shun
MIT CSAIL
USA
jshun@mit.edu

ABSTRACT

Maintaining a k-core decomposition quickly in a dynamic graph has important applications in network analysis. The main challenge for designing efficient *exact* algorithms is that a single update to the graph can cause significant global changes. Our paper focuses on *approximation* algorithms with small approximation factors that are much more efficient than what exact algorithms can obtain.

We present the first parallel, batch-dynamic algorithm for approximate k-core decomposition that is efficient in both theory and practice. Our algorithm is based on our novel *parallel level data structure*, inspired by the sequential level data structures of Bhattacharya et al. [STOC '15] and Henzinger et al. [2020]. Given a graph with n vertices and a batch of updates $\mathcal{B}$, our algorithm provably maintains a $(2+\varepsilon)$-approximation of the coreness values of all vertices (for any constant $\varepsilon > 0$) in $O(|\mathcal{B}| \log^2 n)$ amortized work and $O(\log^2 n \log\log n)$ depth (parallel time) with high probability.

As a by-product, our k-core decomposition algorithm also gives a batch-dynamic algorithm for maintaining an $O(\alpha)$ out-degree orientation, where α is the *current* arboricity of the graph. We demonstrate the usefulness of our low out-degree orientation algorithm by presenting a new framework to formally study batch-dynamic algorithms in bounded-arboricity graphs. Our framework obtains new provably-efficient parallel batch-dynamic algorithms for maximal matching, clique counting, and vertex coloring.

We implemented and experimentally evaluated our k-core decomposition algorithm on a 30-core machine with two-way hyper-threading on 11 graphs of varying densities and sizes. Compared to the state-of-the-art algorithms, our algorithm achieves up to a 114.52× speedup against the best parallel implementation, up to a 544.22× speedup against the best approximate sequential algorithm, and up to a 723.72× speedup against the best exact sequential algorithm. We also obtain results for our algorithms on graphs that are orders-of-magnitude larger than those used in previous studies.

1 INTRODUCTION

Discovering the structure of large-scale networks is a fundamental problem for many areas of computing. One of the key challenges is to detect communities in which individuals (or vertices) have close ties with one another, and to understand how well-connected a particular individual is to the community. The well-connectedness of a vertex or a group of vertices is naturally captured by the concept of a k-core or, more generally, the k-core decomposition; hence, this particular problem and its variants have been widely studied in the machine learning [3, 33, 40], database [18, 23, 32, 62, 73], social network analysis and graph analytics [27, 28, 52, 54], computational biology [24, 55, 66, 71], and other communities [39, 54, 69, 79].

Given an undirected graph G, with n vertices and m edges, the k-core of the graph is the maximal subgraph $H \subseteq G$ such that the induced degree of every vertex in H is at least k. The k-core decomposition of the graph is defined as a partition of the graph into layers such that a vertex v is in layer k if it belongs to a k-core but not a $(k+1)$-core; this value is known as the *coreness* of the vertex, and the coreness values induce a natural hierarchical clustering. Classic algorithms for k-core decomposition are inherently sequential. A well-known algorithm for finding the decomposition is to iteratively select and remove all vertices v with smallest degree from the graph until the graph is empty [72]. Unfortunately, the length of the sequential dependencies, or the *depth*, of such a process can be $\Omega(n)$ given a graph with n vertices. As k-core decomposition is a P-complete problem [5], it is unlikely to have a parallel algorithm with polylogarithmic depth. To obtain parallel methods with $\text{poly}(\log n)$ depth, we relax the condition of obtaining an *exact* decomposition to one of obtaining a close *approximate* decomposition.

Previous works studied approximate k-core decompositions as a way for obtaining faster and more scalable algorithms in larger graphs than in exact settings [21, 23, 33, 40, 83]. Approximate coreness values are useful for applications where existing methods are already approximate, such as diffusion protocols in epidemiological studies [24, 55, 66, 71], community detection and network centrality measures [30, 34, 46, 74, 86, 90], network visualization and modeling [3, 20, 89, 91], protein interactions [4, 7], and clustering [41, 60]. Furthermore, due to the rapidly changing nature of today's large networks, many recent studies have focused on the *dynamic* setting, where edges and vertices can be inserted and deleted, and the k-core decomposition is computed in real time. There has been significant interest in obtaining fast and practical dynamic, approximate and exact k-core algorithms. Dynamic algorithms have been developed for both the sequential [63, 63, 65, 78, 83, 88, 94] and parallel [6, 48, 51] settings. There has also been interest in the closely-related dynamic k-truss problem [1, 49, 70, 93]. However, to the best of our knowledge, *there are no existing parallel batch-dynamic k-core algorithms with provable polylogarithmic depth*, which our algorithm achieves.

[†]This work was done while the authors were at MIT CSAIL.

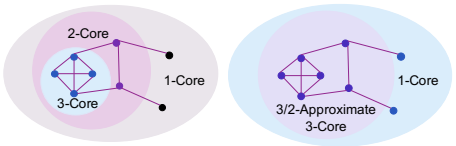


Figure 1: Exact k-core decomposition (left) and $(3/2)$-approximate k-core decomposition (right).

Our paper focuses on the *batch-dynamic* setting where updates are performed over a batch of *multiple* edge updates applied simultaneously. Such a setting is conducive to parallelization, which we leverage to obtain scalable algorithms. We provide a work-efficient batch-dynamic approximate k-core decomposition algorithm based on a parallel level data structure that we design. We implement our algorithm and show experimentally that it performs favorably compared to the state-of-the-art. Furthermore, we show that our parallel level data structure can be used to obtain work-efficient parallel batch-dynamic algorithms for several other problems, specifically, low out-degree orientation, maximal matching, k-clique counting, and vertex coloring.

We introduce the necessary definitions in Section 2 before giving a technical overview of our results in Section 3. Section 5 presents our parallel level data structure and k-core decomposition algorithm in more detail. Section 6 presents experimental results. Section 7 gives our parallel, static, approximate algorithm for k-core decomposition. Finally, Section 8 gives our low out-degree framework for our maximal matching (Section 9), k-clique counting (Section 10), and coloring (Section 11) results.

2 PRELIMINARIES

This paper studies undirected, unweighted graphs, and we use n to denote the number of vertices and m to denote the number of edges in a graph. Definition 2.3 defines approximate k-core decomposition. The definition requires the definition of a k-core, which we define first.

Definition 2.1 (k-Core). *For a graph G and positive integer k, the k-**core** of G is the maximal subgraph of G with minimum induced degree k.*

Definition 2.2 (k-Core Decomposition). *A k-**core decomposition** is a partition of vertices into layers such that a vertex v is in layer k if it belongs to a k-core but not to a $(k+1)$-core. $k(v)$ denotes the layer that vertex v is in, and is called the **coreness** of v.*

Definition 2.2 defines an *exact* k-core decomposition. A c-*approximate* k-core decomposition is defined as follows.

Definition 2.3 (c-Approximate k-Core Decomposition). *A c-**approximate** k-**core decomposition** is a partition of vertices into layers such that a vertex v is in layer k' only if $\frac{k(v)}{c} \leq k' \leq ck(v)$, where $k(v)$ is the coreness of v.*

We let $\hat{k}(v)$ denote the *estimate* of v's coreness. Fig. 1 shows an example of a k-core decomposition and a $(3/2)$-approximate k-core decomposition.

Model Definitions. We analyze the theoretical efficiency of our parallel algorithms in the *work-depth model*. The model is defined in terms of two complexity measures, ***work*** and ***depth*** [25, 50]. The ***work*** is the total number of operations executed by the algorithm. The ***depth*** is the longest chain of sequential dependencies.

Symbol	Meaning
$G = (V, E)$	undirected/unweighted graph
n, m	number of vertices, edges, resp.
α	current arboricity of graph
Δ	current maximum degree of graph
K	number of levels in PLDS
$N(v)$ (resp. $N(S)$)	set of neighbors of vertex v (resp. vertices S)
$\mathrm{dl}(v)$	*desire-level* of vertex v
$\ell, \ell(v)$	a level (starting with level $\ell = 0$), current level of vertex v, resp.
V_ℓ, Z_ℓ	set of vertices in level ℓ, set of vertices in levels $\geq \ell$, resp.
g_i	set of levels in group i (starting with g_0)
$g(v), gn(\ell)$	*group number* of vertex v, index i where level $\ell \in g_i$, resp.
$k(v), \hat{k}(v)$	coreness of v, estimate of the coreness of v, resp.
$\mathrm{up}(v)$,$\mathrm{up}^*(v)$	*up-degree* of v, *up*-degree* of v, resp.
$\varepsilon, \lambda, \delta$	constants where $\varepsilon, \lambda, \delta > 0$

Table 1: Table of notations used in this paper.

We assume that concurrent reads and writes are supported in $O(1)$ work/depth. A ***work-efficient*** parallel algorithm is one with work that asymptotically matches the best-known sequential time complexity for the problem. We say that a bound holds ***with high probability (w.h.p.)*** if it holds with probability at least $1 - 1/n^c$ for any $c \geq 1$.

We use parallel primitives in our algorithms, which take as input a sequence A of length n, including: parallel ***reduce-add***, which returns the sum of the entries in A, and parallel ***filter***, which also takes as input a predicate function f, and returns the sequence B containing each element $a \in A$ where $f(a)$ is true, while preserving the same relative order as the order of elements in A. These primitives take $O(n)$ work and $O(\log n)$ depth [50]. We also use ***parallel hash tables*** that support insertions, deletions, and membership queries; they can perform n insertions or deletions in $O(n)$ work and $O(\log^* n)$ depth w.h.p., and n membership queries in $O(n)$ work and $O(1)$ depth w.h.p. [42]. Provided an input sequence A, a parallel ***prefix-sum*** takes as input an identity x and an associative binary operator $\oplus$, and returns the sequence B of length n where $B[i] = \bigoplus_{j<i} A[j] \oplus x$. This primitive takes $O(n)$ work and $O(\log n)$ depth [50].

Our parallel algorithms operate in the batch-dynamic setting. A ***batch-dynamic*** algorithm processes updates (vertex or edge insertions/deletions) in batches $\mathcal{B}$ of size $|\mathcal{B}|$. For simplicity, since we can reprocess the graph using an efficient parallel static algorithm when $|\mathcal{B}| \geq m$, we consider $1 \leq |\mathcal{B}| < m$ for our bounds.

Given a graph $G = (V, E)$ and a sequence of batches of edge insertions and deletions, $\mathcal{B}_1, \ldots, \mathcal{B}_N$, where $\mathcal{B}_i = (E^i_{delete}, E^i_{insert})$, the goal is to efficiently maintain a $(2+\varepsilon)$-approximate k-core decomposition (for any constant $\varepsilon > 0$) after applying each batch $\mathcal{B}_i$ (in order) on G. In other words, let $G_i = (V, E_i)$ be the graph after applying batches $\mathcal{B}_1, \ldots, \mathcal{B}_i$ and suppose that we have a $(2+\varepsilon)$-approximate k-core decomposition on G_i; then, for $\mathcal{B}_{i+1}$, our goal is to efficiently find a $(2+\varepsilon)$-approximate k-core decomposition of $G_{i+1} = (V, (E_i \cup E^{i+1}_{insert}) \setminus E^{i+1}_{delete})$.

All notations used are summarized in Table 1. Our data structure also maintains a *low out-degree orientation*, which may be parameterized by a graph property known as the *arboricity*.

Definition 2.4 (Arboricity). *The arboricity (α) of a graph is the minimum number of spanning forests needed to cover the graph.*

Definition 2.5 (c-Approximate Low Out-Degree Orientation)**.** *Given an undirected graph $G = (V, E)$, a c-approximate low out-degree orientation is an acyclic orientation of all edges in G such that the maximum out-degree of any vertex, d^+_{max}, is within a c-factor of the minimum possible maximum out-degree, d^+_{opt} of any acyclic orientation:*[1] $d^+_{opt}/c \leq d^+_{max} \leq c \cdot d^+_{opt}$.

We define an $O(\alpha)$ ***out-degree orientation*** to be an acyclic orientation where all out-degrees are $O(\alpha)$. For an oriented graph, we call neighbors of vertex v connected by outgoing edges the ***out-neighbors*** of v and neighbors of v connected by incoming edges the ***in-neighbors*** of v. Definitions of the other problems we consider are given at the top of their respective sections (Sections 9 to 11).

3 TECHNICAL OVERVIEW

In this paper, we provide a number of parallel work-efficient algorithms for various problems. This section gives an overview of our algorithms and how they compare with prior work. Table 2 summarizes our algorithmic results.

We first discuss k-core decomposition. A number of previous works [64, 68, 78, 92, 93] provided methods for maintaining the *exact* k-core decomposition under single edge updates in the sequential setting. Unfortunately, none of these works provide algorithms with provable polylogarithmic update time. The main bottleneck for obtaining *provably-efficient* methods is that a single edge update can cause *all* coreness values to change: consider a cycle with one edge removed as a simple example. Removing and adding the edge into this cycle, repeatedly in succession, causes the coreness of all vertices to change by one with each update. In the parallel setting, a number of previous works [6, 38, 48, 51, 87] investigated batch-dynamic algorithms for exact k-core decomposition. Unfortunately, none of these works have poly$(\log n)$ depth and some even have $\Omega(n)$ depth.

This paper shows that we can surprisingly obtain a parallel batch-dynamic k-core decomposition algorithm with amortized time bounds that are independent of the number of vertices that *changed coreness* for *approximate* coreness. Such provable time bounds can be obtained by cleverly avoiding updating coreness values until enough error has accumulated; once such error has accumulated, we can charge the amount of time required to update the coreness to the number of updates that occurred. Doing so carefully allows a provable $O(\log^2 n)$ amortized work per update that is independent of the number of changed coreness values. A recent paper by Sun et al. [83] provides a *sequential* dynamic approximate k-core decomposition algorithm that takes $O(\log^2 n)$ amortized time per update. Their algorithm is a threshold peeling/elimination procedure that gives a $(2+\varepsilon)$-approximation bound. They also provide another sequential algorithm, which they call *round-indexing*, that performs faster in practice.[2] However, they do not provide formal runtime proofs for this algorithm. Their threshold peeling algorithm is inherently sequential since a vertex that changes thresholds can cause another to change their threshold (and coreness estimate), resulting in a long chain of sequential dependencies; such a situation results in polylogarithmic *amortized* depth, whereas efficient parallel algorithms require polylogarithmic depth w.h.p. in the *worst case*, which we obtain.

[1] d^+_{opt}, is equal to the *degeneracy*, d, of G, and is closely related to α: $d/2 \leq \alpha \leq d$.

[2] Our experiments compare against the round-indexing algorithm since it is faster than their thresholding peeling algorithm in practice.

Table 2: Work and depth bounds of algorithms in this paper.[3]

Problem	Approx	Work	Depth	Adversary
k-core	$(2+\varepsilon)$	$O(\lvert\mathcal{B}\rvert \log^2 n)$	$\widetilde{O}(\log^2 n)$[4]	Adaptive
k-core	$(2+\varepsilon)$	$O(m+n)$	$\widetilde{O}(\log^3 n)$	Static
Orientation	$(4+\varepsilon)$	$O(\lvert\mathcal{B}\rvert \log^2 n)$	$\widetilde{O}(\log^2 n)$	Adaptive
Matching	Maximal	$O(\lvert\mathcal{B}\rvert(\alpha + \log^2 n))$	$\widetilde{O}(\log \Delta \log^2 n)$[6]	Adaptive
k-clique	Exact	$O(\lvert\mathcal{B}\rvert \alpha^{k-2} \log^2 n)$	$\widetilde{O}(\log^2 n)$	Adaptive
Coloring	$O(\alpha \log n)$[5]	$O(\lvert\mathcal{B}\rvert \log^2 n)$	$\widetilde{O}(\log^2 n)$	Oblivious
Coloring	$O(2^\alpha)$	$O(\lvert\mathcal{B}\rvert \log^3 n)$	$\widetilde{O}(\log^2 n)$	Adaptive

To design our k-core decomposition algorithm, we formulate a *parallel level data structure (PLDS)* inspired by the sequential level data structures (LDS) of Bhattacharya et al. [13] and Henzinger et al. [47] to maintain a partition of the vertices satisfying specific degree properties in certain induced subgraphs. In the LDS, vertices are updated one at a time. One of our main technical insights is that we can update many vertices *simultaneously*, leading to high parallelism. Our k-core decomposition algorithm is work-efficient, and matches the approximation factor of the best-known sequential dynamic approximate k-core decomposition algorithm of Sun et al. [83], while achieving polylogarithmic depth w.h.p.

Dynamic problems related to k-core decompositions have been recently studied in the theory community, such as densest subgraph [13, 80] and low out-degree orientations [10, 19, 45, 47, 53, 56, 57, 82]; some of these works use the LDS. However, none of these previous works proved guarantees regarding the k-core decomposition that can be maintained via a LDS. Notably, we show via a new, intuitive proof that one can use the level of a vertex to estimate its coreness in the LDS of [47]. Unlike the proof in [83] for their dynamic algorithm, our proof does not require densest subgraphs nor any additional information besides the two invariants maintained by the structure.

Our main theoretical and practical technical contributions for k-core decomposition are three-fold: (1) we present a simple modification and a new $(2+\varepsilon)$-approximate coreness proof for the sequential level data structure of [13, 47] (which were not previously used for coreness values) using only the levels of the vertices—no such modification was known prior to this work since [83] requires an additional elimination/peeling/round-indexing procedure; (2) we provide the first parallel work-efficient batch-dynamic level data structure that takes $O(\log^2 n \log\log n)$ depth w.h.p., which we use to obtain a $(2+\epsilon)$-approximate batch-dynamic k-core decomposition algorithm; and (3) we provide multicore implementations of our new algorithm and demonstrate its practicality through extensive experimentation with state-of-the-art parallel and sequential algorithms.

The following theorems give our theoretical bounds.

Theorem 3.1 (Batch-Dynamic k-Core Decomposition)**.** *Given $G = (V, E)$ where $n = |V|$ and batch of updates $\mathcal{B}$, our algorithm maintains $(2+\varepsilon)$-approximations of core values for all vertices (for any constant $\varepsilon > 0$) in $O(|\mathcal{B}| \log^2 n)$ amortized work and $O(\log^2 n \log\log n)$ depth w.h.p., using $O(n \log^2 n + m)$ space.*

[3] All bounds are w.h.p., except for the work of static k-core and $O(\alpha \log n)$-coloring.

[4] $\widetilde{O}$ hides a factor of $O(\log\log n)$.

[5] We denote by α the *current* arboricity of the graph *after processing all updates including the most recent ones.*

Using the same parallel level data structure, we also obtain the following result for maintaining a low out-degree orientation.

Theorem 3.2 (Batch-Dynamic Low Out-Degree Orientation)**.** *Our algorithm maintains an $(4+\varepsilon)$-approximation of a minimum acyclic out-degree orientation, with the same bounds as Theorem 3.1, where the amortized number of edge flips is $O(|\mathcal{B}|\log^2 n)$.*

A consequence of Theorem 3.2 is the following corollary.

Corollary 3.3 ($O(\alpha)$ Out-Degree Orientation)**.** *Our algorithm maintains an $O(\alpha)$ out-degree orientation, where α is the current arboricity (Definition 2.4), with the same bounds as Theorem 3.2.*

Using Theorem 3.2, we design a framework for parallel batch-dynamic algorithms on bounded-arboricity graphs for batch of updates $\mathcal{B}$, which in addition to problem-specific techniques allows us to obtain a set of batch-dynamic algorithms for a variety of other fundamental graph problems including maximal matching, clique counting, and vertex coloring. The coloring algorithms are based heavily on the sequential algorithms of Henzinger et al. [47], but we present them as an application of our framework. For the problems we consider in this paper, Fig. 3 summarizes the update times of the previous best-known sequential results for their respective settings.

Theorem 3.4 (Batch-Dynamic Maximal Matching)**.** *We maintain a maximal matching in $O(|\mathcal{B}|(\alpha+\log^2 n))$ amortized work and $O(\log^2 n\,(\log\Delta+\log\log n))$ depth w.h.p.,[6] in $O(n\log^2 n+m)$ space.*

Theorem 3.5 (Batch-Dynamic Implicit $O(2^\alpha)$-Vertex Coloring)**.** *We maintain an implicit $O(2^\alpha)$-vertex coloring[7] in $O(|\mathcal{B}|\log^3 n)$ amortized work and $O(\log^2 n)$ depth w.h.p. for updates, and $O(Q\alpha\log n)$ work and $O(\log n)$ depth w.h.p., for Q queries, using $O(n\log^2 n+m)$ space.*

Theorem 3.6 (Batch-Dynamic k-Clique Counting)**.** *We maintain the count of k-cliques in $O(|\mathcal{B}|\alpha^{k-2}\log^2 n)$ amortized work and $O(\log^2 n\log\log n)$ depth w.h.p., in $O(m\alpha^{k-2}+n\log^2 n)$ space.*

All of the above results are robust against *adaptive* adversaries which have access to the algorithm's previous outputs. The following algorithm is robust against *oblivious* adversaries which do not have access to previous outputs.

Theorem 3.7. *We maintain an $O(\alpha\log n)$-vertex coloring in $O(|\mathcal{B}|\log^2 n)$ amortized expected work and $O(\log^2 n\log\log n)$ depth w.h.p., in $O(m+n\log^2 n+\alpha\log n)$ space.*

Our k-core, low out-degree orientation, and vertex coloring algorithms are work-efficient when compared to the best-known sequential, dynamic algorithms for the respective problems [13, 47, 83]. For maximal matching, our algorithm is work-efficient when $\alpha=\Omega(\log^2 n)$ when compared to the best-known sequential algorithm that is robust against adaptive adversaries [45, 75]; the extra work when $\alpha=o(\log^2 n)$ comes from the fact that our bounds are with respect to the *current* arboricity, compared to [45, 75] whose bounds are with respect to the *maximum* arboricity over the sequence of updates.

The best-known batch-dynamic algorithm for k-clique counting, by Dhulipala et al. [29], takes $O(|\mathcal{B}|m\alpha^{k-4})$ expected work and $O(\log^{k-2} n)$ depth w.h.p., using $O(m+|\mathcal{B}|)$ space. Compared with their algorithm, our algorithm uses less work when $m=\omega(\alpha^2\log^2 n)$. In many real-world networks, $\alpha<<\sqrt{m}$ (see e.g., Table 3, for maximum k-core values, which upper bound α); thus, our result is more efficient in many cases at an additional multiplicative space cost of $O(\alpha^{k-2})$. We also obtain smaller depth for all $k>4$. We provide further comparisons with the best-known sequential clique counting algorithm [31], and we describe more specific batch-dynamic challenges we face in designing the above algorithms in their respective sections. The components of the PLDS used in each of the above results are summarized in Fig. 2.

[6] Δ denotes the maximum *current* degree of the graph *after* processing all updates.
[7] An *implicit* vertex coloring algorithm returns valid colorings for queried vertices.

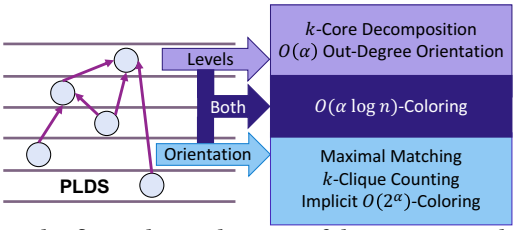


Figure 2: This figure shows what parts of the PLDS are used in each result. The level of each vertex is used to determine the k-core decomposition (Theorem 3.1) and low out-degree orientation (Theorem 3.2 and Corollary 3.3). The orientation of the edges is used for maximal matching (Theorem 3.4), implicit $O(2^\alpha)$-coloring (Theorem 3.5), and k-clique counting (Theorem 3.6). Finally, both are used for $O(\alpha\log n)$-coloring (Theorem 3.7).

Finally, using ideas from our batch-dynamic k-core decomposition algorithm, we provide a new parallel static $(2+\varepsilon)$-approximate k-core decomposition algorithm. We compare this algorithm with the best-known parallel static exact algorithm of [27] which uses $O(m+n)$ expected work and $O(\rho\log m)$ depth w.h.p., where ρ is the *number of steps necessary to peel all vertices* (ρ could potentially be $\Omega(n)$). Hence, [27] does not guarantee poly$(\log n)$ depth.

Theorem 3.8. *Given $G=(V,E)$ with $n=|V|$ vertices and $m=|E|$ edges, for any constant $\varepsilon>0$, our algorithm finds an $(2+\varepsilon)$-approximate k-core decomposition in $O(n+m)$ expected work and $O(\log^3 n)$ depth w.h.p., using $O(n+m)$ space.*

Experimental Contributions. In addition to our theoretical contributions, we also provide optimized multicore implementations of our k-core decomposition algorithms. We compare the performance of our algorithms with state-of-the-art algorithms on a variety of real-world graphs using a 30-core machine with two-way hyper-threading. Our parallel static approximate k-core algorithm achieves a 2.8–3.9x speedup over the fastest parallel exact k-core algorithm [27] and achieves a 14.76–36.07x self-relative speedup.

We show that our parallel batch-dynamic k-core algorithm achieves up to 544.22× speedups over the state-of-the-art sequential dynamic approximate k-core algorithm of Sun et al. [83], while achieving comparable accuracy. We also achieve up to 114.52× speedups over the state-of-the-art parallel batch-dynamic exact k-core algorithm of Hua et al. [48], and up to 723.72× speedups against the state-of-the-art sequential exact k-core algorithm of Zhang and Yu [93]. Our batch-dynamic algorithm outperforms the best multicore static k-core algorithms by up to 121.76× on batch sizes that are less than 1/3 of the number of edges in the entire graph.

Our algorithm exhibits improvements in runtime while maintaining the same or smaller error, even when using only four threads (available on a standard laptop), and remains competitive at one

thread. We demonstrate that existing exact dynamic implementations are not efficient or scalable enough to handle graphs with billions of edges, whereas our algorithm is able to. Furthermore, our demonstrated speedups of up to two orders of magnitude indicates that our implementation not only fills the gap for processing graphs that are orders of magnitude larger than can be handled by existing implementations, but also that it is the best option for many smaller networks. Our code is publicly available at https://github.com/qqliu/batch-dynamic-kcore-decomposition.

4 COMPARISONS WITH OTHER RELATED WORK

Parallel Exact Batch-Dynamic Algorithms. The most recent, state-of-the-art parallel batch-dynamic algorithm by Hua et al. [48] improves upon the previous parallel algorithms of Aridhi et al. [6], Wang et al. [87], and Jin et al. [51]. Their algorithm relies on the concept of a *joint edge set*, whose insertion and removal determines the core numbers of the vertices. However, their algorithm could take $\Omega(n)$ depth as they use a standard depth-first search to traverse vertices in the joint edge set as well as vertices outside of the joint edge set. In comparison, our algorithm provably has $O(\log^2 n \log\log n)$ depth w.h.p. Our theoretical improvements also translate to practical gains since we demonstrate greater scalability in our experiments.

Another recent work by Gabert et al. [38] provides a scalable exact k-core maintenance algorithm. Both their asymptotic work and depth is super-polylogarithmic (in fact, in the worst case it could be as bad as computing from scratch). Unfortunately, the code for their experiments is proprietary and hence not available for comparison. However, their reported experimental results overall appear slower than our results, described in more detail in Section 6.4.

Low Out-Degree Orientations. Many previous works give dynamic algorithms for low out-degree orientations with respect to bounds on the *maximum* arboricity that ever exists in the graph, α_{max} [10, 19, 45, 53, 56, 57, 82]. Noticeably, these sequential, dynamic works save a $O(\log n)$ factor in the running time compared to sequential dynamic algorithms that compute the orientation with respect to the *current* arboricity [13, 47]. In practice, the arboricity of real-world graphs may vary as batches of updates are applied, and in particular, the k-core numbers of each vertex can change drastically (e.g., many follows and unfollows can occur in a very short period of time following a viral post). Our work matches the update time of [13, 47] for maintaining a low out-degree orientation for the current α. This explains why our work bounds for maximal matching requires an additional $O(\log n)$ factor compared to previous works [45, 75] that were in terms of α_{max}.

Other Graph Problems. Using low out-degree orientations, a number of works in the past have studied the other dynamic graph problems we study in this paper, including maximal matching, vertex coloring, and clique counting [8, 11, 12, 22, 29, 31, 37, 44, 45, 47, 58, 67, 75, 76, 82]. The best update time for these problems in the sequential settings are summarized in Fig. 3.

In the sequential setting, the best-known algorithm for k-clique counting uses $O(\log^{k^2} n)$ update time in bounded *expansion* graphs for any k-vertex subgraph [31]. Bounded expansion is a more restricted class of graphs than bounded arboricity.[8] Their algorithm crucially requires the *fraternal augmentation* graph, G', which is created from an input directed graph, $G = (V, E)$, by adding an edge (u, v) (direction chosen arbitrarily) if and only if (w, u) and (w, v) exist. Provided an out-degree orientation of size σ, their algorithm runs in $O(\sigma^{k^2} \log^{k^2} n)$ time; so for bounded arboricity graphs, their algorithm can find any subgraph of size k with $O(\alpha^{k^2} \log^{k^2} n)$ update time [31]. Our algorithm also gives a better update time in the sequential setting than [31] for counting cliques (for $|\mathcal{B}| = 1$).

Figure 3: Previous best-known sequential algorithm results.

	Summary of Best-Known Sequential Results		
Problem	Approx	Update Time	Adversary
k-core	$(2+\varepsilon)$	$O(\log^2 n)$ [47, 83], Lemma 5.12	Adaptive
Orientation	$(4+\varepsilon)$	$O(\log^2 n)$ [47]	Adaptive
Matching	Maximal	$O(\alpha_{max} + \log n/\log\log n)$ [45, 75]	Adaptive
k-clique	Exact	$O(\alpha_{max}^{k^2} \log^{k^2} n)$ [31]	Adaptive
Coloring	$O(\alpha \log n)$	$O(\log^2 n)$ [47]	Oblivious
Coloring	$O(2^{\alpha})$	$O(\log^3 n)$ [47]	Adaptive

5 BATCH-DYNAMIC k-CORE DECOMPOSITION

In this section, we describe our parallel, batch-dynamic algorithm for maintaining an $(2+\varepsilon)$-approximate k-core decomposition (for any constant $\varepsilon > 0$) and prove its theoretical efficiency.

5.1 Algorithm Overview

We present a *parallel level data structure (PLDS)* that maintains a $(2+\varepsilon)$-approximate k-core decomposition that is inspired by the class of sequential level data structures (LDS) of [13, 47]. Our algorithm achieves $O(\log^2 n)$ amortized work per update and $O(\log^2 n \log\log n)$ depth w.h.p. We also present a deterministic version of our algorithm that achieves the same work bound with $O(\log^3 n)$ depth. Our data structure can also handle batches of vertex insertions/deletions. Our data structure requires $O(\log^2 n)$ amortized work, which matches the $O(\log^2 n)$ amortized update time of [13, 47]. We also present a *deterministic* version of our algorithm that achieves the same work bound with $O(\log^3 n)$ depth in Section 5.8.

In addition to edge updates, our data structure also handles batches of vertex insertions/deletions, discussed in Section 5.9. As in [47], our data structure can handle *changing arboricity* that is not known a priori. Such adaptivity is necessary to successfully maintain accurate approximations of coreness values.

The LDS and our PLDS consists of a partition of the vertices into $K = O(\log^2 n)$ ***levels***.[9] We provide a very high level overview of PLDS in this section. The levels are partitioned into equal-sized ***groups*** of consecutive levels. Updates are partitioned into insertions and deletions. Vertices move up and down levels depending on the type of edge update incident to the vertex. Rules governing the induced degrees of vertices to neighbors in different levels determine whether a vertex moves. Using information about the level of a vertex, we obtain a $(2+\varepsilon)$-approximation on the coreness of the vertex.

[8] Graphs with bounded expansion have bounded arboricity, but not vice versa.
[9] When $m = o(n)$, we can also show that $O(\log^2 m)$ levels suffice.

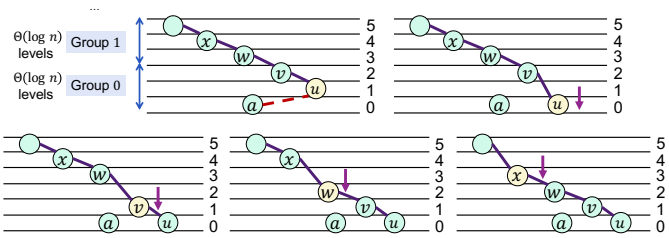


Figure 4: Example of a cascade of vertex movements caused by an edge deletion on u (shown by the dashed red line).

After every edge update, vertices update their levels depending on whether they satisfy two invariants. One invariant upper bounds the induced degree of each vertex v in the subgraph consisting of all vertices in the same or higher level. Vertices whose degree exceeds this bound move up one or more levels. We process the levels from smallest to largest level and move all vertices from the same level in parallel. The second invariant lower bounds the induced degree of each vertex v in the subgraph consisting of all vertices in the level below v, the level of v and all levels higher than the level of v. Vertices that violate this invariant calculate a ***desire-level*** or the closest level they can move to that satisfies this invariant. Then, vertices with the same desire-level are moved in parallel to that level. Finally, the coreness estimates of the vertices are computed based on the current level of each vertex. We obtain the low out-degree orientation by orienting edges from lower to higher levels (breaking ties by vertex index). Fig. 5 shows the invariants maintained by our algorithm; Figs. 6 and 7 show how our algorithm processes insertion and deletion updates. Together, they demonstrate an example run of our algorithm.

5.2 Sequential Level Data Structure (LDS)

The sequential level data structures (LDS) of [13, 47] maintains a low out-degree orientation under dynamic updates. Within their LDS, a vertex moves up or down levels one by one, where a vertex v (incident to an edge update) first checks whether an invariant is violated, and then may move up or down one level. Then, the vertex checks the invariants and repeats. Such movements may cause other vertices to move up or down levels. The LDS combined with our Section 5.6 directly gives an $O(\log^2 n)$ update time sequential, dynamic algorithm that outputs $(2+\varepsilon)$-approximate coreness values.

Unfortunately, such a procedure can be slow in practice. Specifically, a vertex that moves one level could cause a cascade of vertices to move one level. Then, if the vertex moves again, the same cascade of movements may occur. An example is shown in Fig. 4. Furthermore, any trivial parallelization of the LDS to support a batch of updates will run into race conditions and other issues, requiring the use of locks which blows up the runtime in practice.

Thus, our PLDS solves several challenges posed by the sequential LDS. Given a batch $\mathcal{B}$ of edge updates: **(1)** our algorithm processes the levels in a careful order that yields provably low depth for batches of updates; **(2)** our insertion algorithm processes vertices on each level at most once, which is key to the depth bounds—after vertices move up from level ℓ, no future step in the algorithm moves a vertex up from level ℓ; and **(3)** our deletion algorithm moves vertices to their final level in one step. In other words, a vertex moves at most once in a deletion batch.

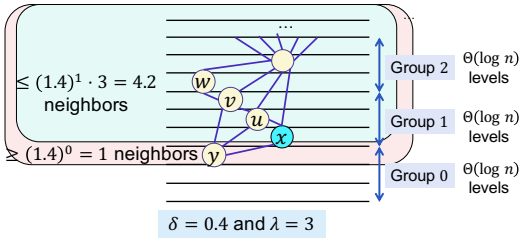


Figure 5: Example of invariants maintained by the PLDS for $\delta = 0.4$ and $\lambda = 3$. There are $\Theta(\log n)$ groups, each with $\Theta(\log n)$. Each vertex is in exactly one level of the structure and moves up and down by some movement rules. For example, vertex x (blue) is on level 3 and in group 1.

Algorithm 1 Update($\mathcal{B}$)

Input: A batch of edge updates $\mathcal{B}$.

1: Let $\mathcal{B}_{ins}$ = all edge insertions in $\mathcal{B}$, and $\mathcal{B}_{del}$ = all edge deletions in $\mathcal{B}$.
2: Call RebalanceInsertions($\mathcal{B}_{ins}$). [Algorithm 2]
3: Call RebalanceDeletions($\mathcal{B}_{del}$). [Algorithm 3]

5.3 Detailed PLDS Algorithm

As mentioned previously, the vertices of the input graph $G = (V, E)$ in our PLDS are partitioned across K ***levels***. For each level $\ell = 0, \ldots, K-1$, let V_ℓ be the set of vertices that are currently assigned to level ℓ. Let Z_ℓ be the set of vertices in levels $\geq \ell$. Provided a constant $\delta > 0$, the levels are partitioned into ***groups*** $g_0, \ldots, g_{\lceil \log_{(1+\delta)} n \rceil}$, where each group contains $4\lceil \log_{(1+\delta)} n \rceil$ consecutive levels. Each $\ell \in \left[i\lceil \log_{(1+\delta)} n \rceil, \ldots, (i+1)\lceil \log_{(1+\delta)} n \rceil - 1 \right]$ is a level in group i. Our data structure consists of $K = O(\log^2 n)$ total levels. The PLDS satisfies the following invariants as introduced in [13, 47], which also govern how the data structure is maintained. The invariants assume a given constant $\delta > 0$ and a constant $\lambda > 0$.

Invariant 1 (Degree Upper Bound). *If vertex $v \in V_\ell$, level $\ell < K$ and $\ell \in g_i$, then v has at most $(2 + 3/\lambda)(1+\delta)^i$ neighbors in Z_ℓ.*

Invariant 2 (Degree Lower Bound). *If vertex $v \in V_\ell$, level $\ell > 0$, and $\ell - 1 \in g_i$, then v has at least $(1+\delta)^i$ neighbors in $Z_{\ell-1}$.*

Vertices with no neighbors are placed in level 0. An example partitioning of vertices and maintained invariants is shown in Fig. 5. Let $\ell(v)$ be the level that v is currently on. We define the ***group number***, $g(v)$, of a vertex v to be the index i of the group g_i where $\ell(v) \in g_i$. Similarly, we define $gn(\ell) = i$ to be the group number for level ℓ where $\ell \in g_i$. We define the ***up-degree***, $\mathrm{up}(v)$, of a vertex v to be the number of its neighbors in $Z_{\ell(v)}$ (***up-neighbors***), and ***up*-degree***, $\mathrm{up}^*(v)$, to be the number of its neighbors in $Z_{\ell(v)-1}$ (up^*-***neighbors***). These two notions of induced degree correspond to the requirements of the two invariants of our data structure. We define neighbors w of v at levels $\ell(w) < \ell(v)$ to be the ***down-neighbors*** of v. Lastly, the ***desire-level*** $\mathrm{dl}(v)$ of a vertex v is the *closest level to the current level of the vertex* that satisfies both Invariant 1 and Invariant 2.

Definition 5.1 (Desire-level). *The* desire-level, $\mathrm{dl}(v)$, *of vertex v is the level ℓ' that minimizes $|\ell(v) - \ell'|$, and where $\mathrm{up}^*(v) \geq (1+\delta)^{i'}$ and $\mathrm{up}(v) \leq (2 + 3/\lambda)(1+\delta)^i$ where $\ell' - 1 \in g_{i'}$, $\ell' \in g_i$, and $i' \leq i$. In other words, the desire-level of v is the closest level ℓ' to the current level of v, $\ell(v)$, where both Invariant 1 and Invariant 2 are satisfied.*

Algorithm 2 RebalanceInsertions(B_{ins})

Input: A batch of edge insertions B_{ins}.

1: Let U contain all up-neighbors of each vertex, keyed by vertex. So $U[v]$ contains all up-neighbors of v.
2: Let L_v contain all neighbors of v in levels $[0, \ldots, \ell(v) - 1]$, keyed by level number.
3: **parfor** each edge insertion $e = (u, v) \in \mathcal{B}_{ins}$ **do**
4: Insert e into the graph.
5: **for** each level $l \in [0, \ldots, K - 1]$ starting with $l = 0$ **do**
6: **parfor** each vertex v incident to B_{ins} or is marked, where $\ell(v) = l \cap \text{up}(v) > (2 + 3/\lambda)(1 + \delta)^{gn(l)}$ **do**
7: Mark and move v to level $l + 1$ and create $L_v[l]$ to store v's neighbors at level l.
8: **parfor** each $w \in N(v)$ of a vertex v that moved to level $l + 1$ and w stayed in level l **do**
9: $U[v] \leftarrow U[v] \setminus \{w\}, L_v[l] \leftarrow L_v[l] \cup \{w\}$.
10: **parfor** each $u \in N(v)$ of a vertex v that moved to level $l + 1$ and u is in level $l + 1$ **do**
11: Mark u if $\text{up}(u) > (2 + 3/\lambda)(1 + \delta)^{gn(l+1)}$.
12: $U[u] \leftarrow U[u] \cup \{v\}, L_u[l] \leftarrow L_u[l] \setminus \{v\}$.
13: **parfor** each $x \in N(v)$ of a vertex v that moved to level $l + 1$ and x is in level $\ell(x) \geq l + 2$ **do**
14: $L_x[l] \leftarrow L_x[l] \setminus \{v\}, L_x[l + 1] \leftarrow L_x[l + 1] \cup \{v\}$.
15: Unmark v if $\text{up}(v) \leq (2 + 3/\lambda)(1 + \delta)^{gn(l+1)}$. Otherwise, leave v marked.

We show that the invariants are always maintained except for a period of time when processing a new batch of insertions/deletions. During this period, the data structure undergoes a *rebalance procedure*, where the invariants may be violated. The main update procedure in Algorithm 1 separates the updates into insertions and deletions (Line 1), and then calls RebalanceInsertions (Line 2) and RebalanceDeletions (Line 3). We make two *crucial* observations: when processing a batch of insertions, Invariant 2 is never violated; and, similarly, when processing a batch of deletions, Invariant 1 is never violated. Thus, no vertex needs to move *down* when processing an insertion batch and no vertex needs to move *up* when processing a deletion batch. The two procedures are asymmetric, and so we first describe RebalanceInsertions (Algorithm 2), and then describe RebalanceDeletions (Algorithm 3).

Data Structures. Each vertex v keeps track of its set of neighbors in two structures. U keeps track of the neighbors at v's level and above. We denote this set of v's neighbors by $U[v]$. L_v keeps track of neighbors of v for every level below $\ell(v)$—in particular, $L_v[j]$ contains the neighbors of v at level $j < \ell(v)$.

RebalanceInsertions(B_{ins}). Algorithm 2 shows the pseudocode. Provided a batch of insertions B_{ins}, we iterate through the K levels from the lowest level $\ell = 0$ to the highest level $\ell = K - 1$ (Line 5). For each level, in parallel we check the vertices incident to edge insertions in B_{ins} or is marked to see if they violate Invariant 1 (Line 6). If a vertex v in the current level l violates Invariant 1, we move v to level $l + 1$ (Line 7). After moving v, we update structures $U[v], L_v$, and the structures of $w \in N(v)$ where $\ell(w) \in [l, l + 1]$. First, we create $L_v[l]$ to store the neighbors of v in level l (Line 7). If v moved to level $l + 1$ and w stayed in level l, then we delete w from $U[v]$ and instead insert w into $L_v[l]$ (Lines 8–9). We do not need to make any data structure modifications for w since v stays in $U[w]$. Similarly, no data structure modifications to v and w are necessary when both v and w move to level $l + 1$. For each neighbor of v on level $l + 1$, we need to check whether it now violates Invariant 1 (Line 10). If it does, then we mark the vertex (Line 11). We process any such marked vertices when we process level $l + 1$. We also update the U and L arrays of every neighbor of v on level $l + 1$ (Line 12). Specifically, let u be one such neighbor, we add v to $U[u]$ and remove v from $L_u[l]$. We conclude by making appropriate modifications to L for each neighbor on levels $\geq l + 2$ (Lines 13–14). Specifically, let x be one such neighbor, we remove v from $L_x[l]$ and add v to $L_x[l + 1]$. All neighbors of vertices that moved can be checked and processed in parallel. Finally, v becomes unmarked if it satisfies all invariants; otherwise, it remains marked and must move again in a future step (Line 15).

Fig. 6 shows an example of our entire insertion procedure described in Algorithm 2 for $\delta = 0.4$ and $\lambda = 3$. The red lines in the example represent the batch of edge insertions. Thus, in (a), the newly inserted edges are the edges (u, v), (u, x), and (x, w). We iterate from the bottommost level (level 0) to the topmost level (level $K - 1$).

The first level where we encounter vertices that are marked or are adjacent to an edge insertion is level 2. Since level 2 is part of group 0, the cutoff for Invariant 1 is $(2 + 3/\lambda)(1 + \delta)^0 = 3$ provided $\lambda = 3$ and $\delta = 0.4$. In level 2, only w violates Invariant 1 since the number of its neighbors on levels ≥ 2 is 4 (x, y, z, and a), so $\text{up}(w) = 4 > 3$ (shown in (b)). Then, in (c), we move w up to level 3. We need to update the data structures for neighbors of w at level 3 and above (as well as w's own data structures); the vertices with data structure updates are x, w, y, and z. After the move, x becomes marked because it now violates Invariant 1 (the cutoff for level 3 is $(2 + 3/3)(1 + 0.4) = 4.2$ since level 3 is in group 1); w becomes unmarked because it no longer violates Invariant 1. In (d), we move on to process level 3. The only vertex that is marked or violates Invariant 1 is x. Therefore, we move x up one level (shown in (e)) and update relevant data structures (of x, v, y, z, and b).

RebalanceDeletions(B_{del}). Unlike in LDS, deletions in PLDS are handled by moving each vertex at most once, directly to its final level (the vertex *does not move* again during this procedure). We show in the analysis that this guarantee is *crucial to obtaining low depth*. The pseudocode is shown in Algorithm 3. For each vertex v incident to an edge deletion, we check whether it violates Invariant 2 (Line 4). On Line 4, $gn(\ell(v) - 1)$ gives the group number i where $\ell(v) - 1 \in g_i$. If v violates Invariant 2, we calculate its desire-level, $\text{dl}(v)$, using CalculateDesireLevel (Line 5), described next. We iterate through the levels from $l = 0$ to $l = K - 1$ (Line 6). Then, in parallel for each vertex v whose desire-level is l, we move v to level l (Lines 7–8). We update the data structures of each v that moved and $w \in N(v)$ where $\ell(w) \geq l$ (Lines 9–21). Specifically, we need to update $U[v], U[w], L_v$, and L_w if v was originally an up-neighbor of w and becomes a down-neighbor or vice versa. Finally, we update the desire-level of neighbors of v that no longer satisfy Invariant 2 (Lines 22–23). We process all vertices that move and their neighbors in parallel.

Fig. 7 shows an example of Algorithm 3 for $\delta = 1$ and $\lambda = 3$. In (a), the newly deleted edges are (x, z) and (y, w). For each vertex adjacent to an edge deletion, we calculate its desire-level, or the closest level to its current level that satisfies Invariant 2. In (b),

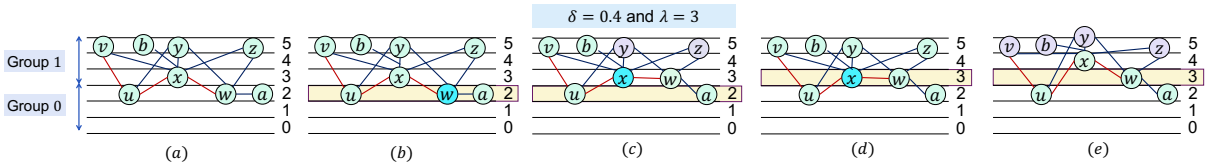


Figure 6: Example of RebalanceInsertions described in the text for $\delta = 0.4$ and $\lambda = 3$. The red lines represent the batch of edge insertions.

Algorithm 3 RebalanceDeletions($\mathcal{B}_{del}$)

Input: A batch of edge deletions $\mathcal{B}_{del}$.

1: Let U contain all up-neighbors of each vertex, keyed by vertex. So $U[v]$ contains all up-neighbors of v. Let L_v contain all neighbors of v in levels $[0, \ldots, \ell(v) - 1]$, keyed by level number.
2: **parfor** each edge deletion $e = (u, v) \in \mathcal{B}_{del}$ **do**
3: Remove e from the graph.
4: **parfor** each vertex v where $\text{up}^*(v) < (1+\delta)^{gn(\ell(v)-1)}$ **do**
5: Calculate $\text{dl}(v)$ using CalculateDesireLevel(v).
6: **for** each level $l \in [0, \ldots, K-1]$ starting with level $l = 0$ **do**
7: **parfor** each vertex v where $\text{dl}(v) = l$ **do**
8: Move v to level l.
9: **parfor** each vertex v where $\text{dl}(v) = l$ **do**
10: **parfor** each neighbor w of v where $\ell(w) \geq l$ **do**
11: Let p_v and p_w be the previous levels of v and w, respectively, before the move.
12: **if** $\ell(w) = l$ **then**
13: $L_w[p_v] \leftarrow L_w[p_v] \setminus \{v\}, L_v[p_w] \leftarrow L_v[p_w] \setminus \{w\}$.
14: $U[w] \leftarrow U[w] \cup \{v\}, U[v] \leftarrow U[v] \cup \{w\}$.
15: **else**
16: **if** $p_v > \ell(w)$ **then**
17: $U[w] \leftarrow U[w] \setminus \{v\}, L_v[\ell(w)] \leftarrow L_v[\ell(w)] \setminus \{w\}$.
18: **else if** $p_v = \ell(w)$ **then**
19: $U[w] \leftarrow U[w] \setminus \{v\}$.
20: **else** $L_w[p_v] \leftarrow L_w[p_v] \setminus \{v\}$.
21: $L_w[l] \leftarrow L_w[l] \cup \{v\}, U[v] \leftarrow U[v] \cup \{w\}$.
22: **if** $\text{up}^*(w) < (1+\delta)^{gn(\ell(w)-1)}$ **then**
23: Recalculate $\text{dl}(w)$ using Algorithm 4.

only x and z violate Invariant 2. The lower bound on the number of neighbors that must be at or above level 3 for x and level 4 for z is $(1+\delta)^1 = 2$ since $\delta = 1$ and levels 3 and 4 are in group 1. (Recall that the lower bound is calculated with respect to the level *below* x and z.) We calculate that the desire-levels of x and z are both 3. The desire-levels of y and w are their current levels because they do not violate the invariant. Then, we iterate from the bottommost level (starting with level 0) to the topmost level (level $K-1$). Level 3 is the first level where vertices want to move. Then, we move x and z to level 3 (shown in (c)). We only need to update the data structures of neighbors at or above x and z so we only update the structures of x, y, and z. Invariant 2 is no longer violated for x and z. In fact, our algorithm guarantees that each vertex *moves at most once*. We check whether any of x or z's up-neighbors violate Invariant 2. Indeed, y now violates the invariant. In (d), we recompute the desire-level of y and its desire-level is now 4. Then, we move y to level 4 in (e).

CalculateDesireLevel(v). Algorithm 4 shows the procedure for calculating the desire-level, $\text{dl}(v)$, of vertex v, which is used in Algorithm 3. Let $gn(\ell)$ be the index i where level $\ell \in g_i$. We use a doubling procedure followed by a binary search to calculate the desire-level. We initialize a variable d to $\text{up}^*(v)$ (number of neighbors at or above level $\ell(v)-1$). Starting with level $\ell(v)-2$, we add the number of neighbors in level $\ell(v)-2$ to d (Algorithm 4, Line 3).

Algorithm 4 CalculateDesireLevel(v)

Input: A vertex v that needs to move to a level $j < \ell(v)$.
Output: The desire-level $\text{dl}(v)$ of vertex v.

1: $d \leftarrow \text{up}^*(v), p \leftarrow 1, i \leftarrow 2$
2: **while** $d < (1+\delta)^{gn(\ell(v)-p)}$ and $\ell(v) - p > 0$ **do**
3: $d \leftarrow d + \sum_{j=p}^{i-1} \left|L_v[\ell(v) - j - 1]\right|$
4: **if** $d \geq (1+\delta)^{gn(\ell(v)-i)}$ **then**
5: Binary search within levels $[\ell(v) - i + 1, \ldots, \ell(v) - p]$ to find the closest level to $\ell(v)$ that satisfies Invariants 1 and 2; **return** this level.
6: $p \leftarrow i, i \leftarrow \min(2 \cdot i, \ell(v))$.
7: **return** 0.

This procedure checks whether moving v to $\ell(v)-1$ satisfies Invariant 2 (Line 4). If it passes the check, then we are done and we move v to $\ell(v)-1$. Otherwise, we iteratively double the number of levels from which we count neighbors until we find a level where Invariant 2 is satisfied (Line 6). On each iteration, we sum the number of neighbors (Line 3) in the range of levels using a parallel reduce. We continue until we find a level where Invariant 2 is satisfied. Let this level be ℓ' and the previous cutoff be ℓ_{prev}. Finally, we perform a binary search within the range $[\ell', \ldots, \ell_{prev}]$ to find the *closest* level to $\ell(v)$ that satisfies Invariant 2 (Line 5).

5.4 Efficiency Analysis

We now analyze the work and depth of our PLDS. First, it is easy to show that there exists a level where both invariants are satisfied. This allows our PLDS to assign each vertex to a single level.

Lemma 5.2. *If a vertex v violates Invariant 1, then there exists a level $l > \ell(v)$ where v satisfies both Invariant 1 and Invariant 2. If a vertex w violates Invariant 2, then there exists a level $l < \ell(w)$ where w satisfies both invariants or $l = 0$ (it is in the bottommost level).*

Proof. First note that no vertex can simultaneously violate both Invariant 1 and Invariant 2. Thus, suppose first that v violates Invariant 1. Then, this means that the number of neighbors of v on levels $\geq \ell(v)$ is more than $(2+3/\lambda)(1+\delta)^{g(v)}$ where $g(v)$ is the group number of v. If v still violates Invariant 1 on level $\ell(v)+1$, then we keep moving v to the next level.

Otherwise, v does not violate Invariant 1 on level $\ell(v)+1$. Since we know that v violated Invariant 1 on level $\ell(v)$, then after we move v to $\ell(v)+1$, v's up*-degree is greater than $(1+\delta)^{gn(\ell(v))}$; hence, v also does not violate Invariant 2. The very last level of the K levels has up-degree bound $(2+3/\lambda)(1+\delta)^{\lceil \log_{1+\delta}(n) \rceil} > 2n$ when a vertex can be adjacent to at most $n-1$ vertices. Hence, there must exist a level at or below the last level where both invariants are satisfied. A similar argument holds for Invariant 2. □

Then, we make the following two observations that a batch of insertions never violates Invariant 2 and a batch of deletions never violates Invariant 1. This is true because deletions can never increase

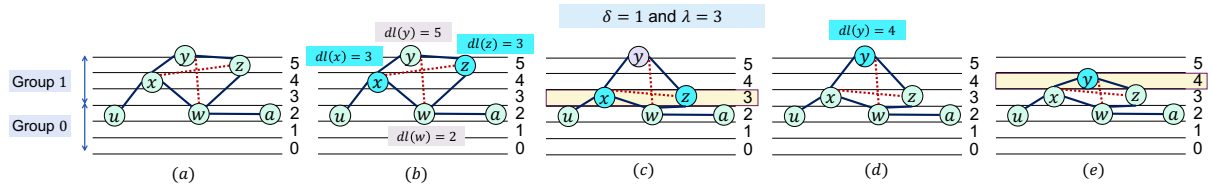


Figure 7: Example of RebalanceDeletions described in the text for $\delta = 1$ and $\lambda = 3$. The red dotted lines represent the batch of edge deletions.

the up-degree of any vertex and insertions can never decrease the up*-degree of any vertex.

Observation 5.3 (Batch Insertions). *Given a batch of insertions, $\mathcal{B}_{ins}$, Invariant 2 is never violated while $\mathcal{B}_{ins}$ is applied.*

Proof. The first part of the algorithm inserts the edges into the data structure. Since no edges are removed from the data structures, the degrees of all the vertices after the insertion of edges cannot decrease. Invariant 2 was satisfied before the insertion of the edges, and hence, it remains satisfied after the insertion of edges because no vertices lose neighbors. We prove that the lemma holds for the remaining part of Algorithm 2 via induction on the level i processed by the procedure. In the base case, when $i = 0$, all vertices v in the level which violate Invariant 1 are moved up to a level $\mathsf{dl}(v) > 0$. By definition of desire-level, v is moved to a level where Invariant 2 is still satisfied, by Lemma 5.2. Vertices from level 0 which move to levels $k \geq 1$ cannot decrease the up*-degree for neighbors in all levels j where $j > 1$. Thus, Invariant 2 cannot be violated for these vertices. Vertices not adjacent to v are not affected by the move.

We assume that Invariant 2 was not violated up to level i and prove it is not violated while processing vertices on level $i + 1$. By our induction hypothesis, no vertices violate Invariant 2 before we process level i+1. Then, when we process level i+1, no vertices move down to a lower level than $i + 1$ by construction of our algorithm because Invariant 2 is not violated for any vertex on level $i + 1$ and if Invariant 1 is violated for any vertex w, w must move up to a higher level. Any vertex w which moves up to a higher level cannot decrease the up*-degree of neighbors of w. Hence, no vertex on levels $\geq i + 1$ can violate Invariant 2. The up*-degree of vertices on levels $< i + 1$ are not affected by the move. Hence, no vertices on levels $< i + 1$ violate Invariant 2. Finally, if a vertex on level $i + 1$ violates Invariant 1, it will move to a level $j > i + 1$ where both invariants are satisfied by Lemma 5.2. □

Observation 5.4 (Batch Deletions). *Given a batch of deletions, $\mathcal{B}_{del}$, Invariant 1 is never violated while $\mathcal{B}_{del}$ is applied.*

Proof. Algorithm 3 first applies all the edge deletions in the batch. Edge deletions cannot make the up-degree of any vertex greater; thus, no vertex violates Invariant 1 after applying the edge deletions. We prove that the rest of the algorithm does not violate Invariant 1 via induction over the levels i. Specifically, we use as our induction hypothesis that after processing the i'th level, no vertices violate Invariant 1. In the base case, when $i = 0$, no vertices violate Invariant 1 at the beginning, and vertices from levels $i > 0$ move to level 0. This means that during the processing of level $i = 0$, vertices only move to level 0 from a higher level. Thus, all such vertices that move will move to a lower level. Since vertices which move to lower levels do not increase the up-degree of any other vertices, no vertex can violate Invariant 1 at the end of processing level 0. We now prove the case for processing level $i + 1$. In this case, we assume by our induction hypothesis that no vertices violate Invariant 1 after we finish processing level i. Thus, all vertices that want to move to level $i + 1$ and violate Invariant 2 are at levels $j > i + 1$. Such vertices move down and thus cannot increase the up-degree of any vertex. This means that after moving all vertices that want to move to level $i + 1$, no vertices violate Invariant 1. □

Batch Insertion Depth Bound. Using our observations, the depth of our batch insertion algorithm (Algorithm 2) depends on the following lemma which states that once we have *processed* a level (after finishing the corresponding iteration of Line 5), no vertex will want to move from any level lower than that level. This means that each level is processed exactly once, resulting in at most $O(\log^2 n)$ levels to be processed sequentially.

Lemma 5.5. *After processing level i in Algorithm 2, no vertex v in levels $\ell(v) \leq i$ will violate Invariant 1. Furthermore, no vertex w on levels $\ell(w) > i$ will have $\mathsf{dl}(w) \leq i$.*

Proof. We prove this via induction. For the base case $i = 0$, all vertices on level 0 are part of each other's up-degree; then, no vertices which move up from $i = 0$ can cause the up-degree of any vertices remaining in level 0 to increase. We now assume the induction hypothesis for $i - 1$ and prove the case for i. Vertices on level $j \leq i$ already contain vertices on levels $\geq i$ in its up-degree. Such vertices on levels $\geq i$ when moved to a higher level are still part of the up-degree of vertices on levels $j \leq i$. Hence, no vertices on levels $j \leq i$ will violate Invariant 1 due to vertices in levels $\geq i$ moving up to a level $l > i$. Then, in order for a vertex w with $\ell(w) > i$ to have $\mathsf{dl}(w) \leq i$, some neighbors of w must have moved to a level $\leq i$. By Observation 5.3, no vertices move down during Algorithm 2, so this is not possible. □

Batch Deletion Depth Bound. For the batch deletion algorithm (Algorithm 3), we prove that, starting from the lowest level, after all vertices with $\mathsf{dl}(w) = i$ are moved to the i'th level, no vertex v will have $\mathsf{dl}(v) \leq i$. This means that each level is processed exactly once, resulting in at most $O(\log^2 n)$ levels to be processed sequentially.

Lemma 5.6. *After processing all vertices that move to level i in Algorithm 3, no vertex v needs to be moved to any level $j \leq i$ in a future iteration of Line 6; i.e., no vertex v has $\mathsf{dl}(v) \leq i$ after processing i.*

Proof. We prove this via induction. In the base case when $i = 0$, all vertices with $\mathsf{dl}(v) = 0$ are moved to level 0. All vertices which have $\mathsf{dl}(v) = 0$ are vertices which have degree 0. Thus, all vertices that do not have $\mathsf{dl}(v) = 0$ have degree ≥ 1 and have $\mathsf{dl}(w) \geq 1$. Hence, after moving all vertices with $\mathsf{dl}(v) = 0$ to level 0, no additional vertices need to be moved to level 0. Assuming our induction hypothesis, we now show our lemma holds for level $i + 1$. All vertices that move to level $i + 1$ violated Invariant 2 and hence

have up*-degree $< (1+\delta)^{gn(j-1)}$ at level $j > i+1$ and up*-degree $\geq (1+\delta)^{gn(i)}$ at level $i+1$. After moving all vertices with $\mathsf{dl}(v) = i+1$ to level $i+1$, no vertices on levels $k \leq i+1$ have their up*-degree decreased by the move. We conclude the proof with vertices at levels $l > i+1$. Suppose for the sake of contradiction that there exists some vertex w on level $l > i+1$ which has $\mathsf{dl}(w) \leq i+1$ after the move. In order for $\mathsf{dl}(w) \leq i+1$, some neighbor(s) of w must move below level i, a contradiction. Finally, due to Observation 5.4, no vertices below level $i+1$ will move up. □

We describe the depth of our parallel data structures next. We provide a set of linear-space data structures in Section 5.8 at the cost of increased depth.

Lemma 5.7. *Algorithm 1 returns a randomized parallel level data structure that maintains Invariant 1 and Invariant 2 and has $O(\log^2 n \log\log n)$ depth, w.h.p., and $O(n\log^2 n + m)$ space.*

Proof. By Lemma 5.5 and Lemma 5.6, Algorithm 2 (Line 5) and Algorithm 3 (Line 6) iterates through $O(\log^2 n)$ levels sequentially. Thus, the depth of algorithms is determined by the depth of the procedures that are run in each level the algorithm iterates through.

We maintain the list of neighbors using separate parallel hash tables for each vertex v. One hash table contains v's neighbors at the same or higher levels. Vertex v's neighbors in levels below $\ell(v)$ are placed in a separate hash table for each level. Parallel lookups into the hash tables require $O(1)$ depth w.h.p., and inserting and deleting elements within the tables require $O(\log^* n)$ depth w.h.p. Simultaneously changing the values within the hash table require $O(\log^* n)$ depth w.h.p. Then, the depth per level of the structure is dominated by Algorithm 4.

The only additional depth we need to consider is the depth incurred from Algorithm 4. Both the doubling search and the binary search require $O(\log K) = O(\log\log n)$ depth. All other contributions come from concurrently modifying and accessing dynamic arrays and hash tables and can be done in $O(\log^* n)$ depth w.h.p.

Using the above, we successfully prove that the depth of Algorithm 1 is $O(\log^2 n \log\log n)$ w.h.p. The extra space in addition to storing the graph is $O(n\log^2 n)$ because we must have $O(\log^2 n)$ size dynamic arrays for each vertex to track their neighbors at lower levels (i.e., the neighbors in L_v). Thus, the total depth of our randomized algorithm is $O(\log^2 n \log\log n)$ w.h.p., and the space used is $O(n\log^2 n + m)$. □

5.5 Potential Argument for Work Bound

Our work bound uses the potential functions presented in Section 4 of [13]. We show that we can analyze our algorithm using these potential functions and our parallel algorithm serializes to a set of sequential steps that obey the potential function. We obtain the following lemma by the potential argument provided in this section.

Lemma 5.8. *For a batch of $|\mathcal{B}| < m$ updates, Algorithm 1 returns a PLDS that maintains Invariant 1 and Invariant 2 in $O(|\mathcal{B}|\log^2 n)$ amortized work and $O(\log^2 n \log\log n)$ depth w.h.p., using $O(n\log^2 n + m)$ space.*

5.5.1 Proof of Work Bound. Unlike the algorithm presented in [13, 47], in each round, to handle deletions, we recompute the $\mathsf{dl}(v)$ of any vertex v that we want to move to a lower level. Specifically, we compute and move v to the closest level that satisfies both Invariant 1 and Invariant 2. This is a different algorithm from the algorithm presented in [13, 47], and so we present for completeness a work argument for our modified algorithm. The work bound we present accounts for the work of any one vertex's movement up or down levels using the potential function argument of [13]. Note that this potential function also gives us the amortized work per edge update of our algorithm since there exists a corresponding set of sequential updates that cannot do less work than the set of parallel updates. Although the algorithm is different, the below potential work bound argument follows closely the work bound proof presented in Bhattacharya et al. [13]. However, we repeat the proof again here (with some modifications) for completeness.

Charging the Cost of Moving Levels. The strategy behind our potential function is to use the *increase* in our potential function due to edge updates to pay for the *decrease* in potential due to vertices moving up or down levels, which is enough to account for the work of moving the vertices. We can then charge our costs to the increase in potential due to edge updates. Below, we bound the increase in potential due to edge updates and the decrease in potential due to vertex movements.

We use the following potential function to calculate our potential. First, recall some notation. Let Z_i be the set of vertices in levels i to K. In other words, $Z_i = \bigcup_{j=i}^{K} V_j$. Let $N(u, Z_i)$ be the set of neighbors of u in the induced subgraph given by Z_i. Let $\ell(u)$ be the current level that u is on. Finally, let $gn(\ell)$ be the group number of level ℓ; in other words, $\ell \in g_{gn(\ell)}$. Let $f : [n] \times [n] \to \{0, 1\}$ be a function where $f(u, v) = 1$ when $\ell(u) = \ell(v)$ and $f(u, v) = 0$ when $\ell(u) \neq \ell(v)$. Using the potential functions defined in [13], for some constant $\lambda > 0$:

$$\Pi = \sum_{v \in V} \Phi(v) + \sum_{e \in E} \Psi(e) \tag{1}$$

$$\Phi(v) = \lambda \sum_{i=0}^{\ell(v)-1} \max(0, (2 + 3/\lambda)(1+\delta)^{gn(i)} - |N(v, Z_i)|) \tag{2}$$

$$\Psi(u, v) = 2(K - \min(\ell(u), \ell(v))) + f(u, v) \tag{3}$$

We first calculate the potential changes for insertions and deletions of edges.

Insertion. The insertion of an edge (u, v) creates a new edge with potential $\Psi(u, v)$. The new potential has value at most $2K + 1$. With an edge insertion $\Phi(u)$ and $\Phi(v)$ cannot increase. Thus, the potential increases by at most $2K + 1$.

Deletion. The deletion of edge (u, v) increases potentials $\Phi(u)$ and $\Phi(v)$ by at most $(2\lambda + 3)K$ and $2K$, respectively. It does not increase any other potential since the potential of edge (u, v) is eliminated.

First it is easy to see that the potential Π is always non-negative. Thus, we can use the positive gain in potential over edge insertions and deletions to pay for the decrease in potential caused by moving vertices to different levels.

Now we discuss the change in potential given a movement of a vertex to a higher or lower level. Moving such a vertex decreases the potential and we show that this decrease in potential is enough to pay for the cost of moving the vertex to a higher or lower level.

A vertex v moves from level i to level $\mathsf{dl}(v) < i$ due to Algorithm 3. Since vertex v moved down at least one level, this means that prior to the move, its up*-degree is $\mathsf{up}^*(v) < (1+\delta)^{gn(\ell(v)-1)}$. It is moved to a level $\mathsf{dl}(v)$ where its up*-degree is at least $(1+\delta)^{gn(\mathsf{dl}(v)-1)}$ and its up-degree is at most $(2+3/\lambda)\,(1+\delta)^{gn(\mathsf{dl}(v))}$ (or it is moved to level 0).

The potential before the move is at least

$$\lambda \sum_{i=0}^{\mathsf{dl}(v)-1} \max\left(0, (2+3/\lambda)\,(1+\delta)^{gn(i)} - |N(v, Z_i)|\right) + \sum_{i=\mathsf{dl}(v)}^{\ell(v)-1} (\lambda+3)(1+\delta)^{gn(i)}$$

since we only move a vertex to a lower level if $\mathsf{up}^*(v) < (1+\delta)^{gn(\ell(v)-1)}$ and we move it to the closest level $\mathsf{dl}(v)$ where Invariant 2 is no longer violated. To derive the second term, since we moved vertex v to level $\mathsf{dl}(v)$, we know that its degree $|N(v, Z_{\mathsf{dl}(v)})| < (1+\delta)^{gn(\mathsf{dl}(v))}$ (otherwise, we could have moved v to level $\mathsf{dl}(v)+1$). Then, substituting $(1+\delta)^{gn(i)}$ for all levels $i \geq \mathsf{dl}(v)$ into $\Phi(v)$ allows us to obtain $\sum_{i=\mathsf{dl}(v)}^{\ell(v)-1}(\lambda+3)(1+\delta)^{gn(i)}$. Then, when it reaches its final level, we know that it is at the highest level it can move to or at level 0. In both cases,

$$\Phi(v) = \lambda \sum_{i=0}^{\mathsf{dl}(v)-1} \max\left(0, (2+3/\lambda)\,(1+\delta)^{gn(i)} - |N(v, Z_i)|\right)$$

after the move. In this case, $\Phi(v)$ decreases by at least $\sum_{i=\mathsf{dl}(v)}^{\ell(v)-1}(\lambda+3)(1+\delta)^{gn(i)}$.

We need to account for two potential increases: the increase in Ψ and the increase in Φ from neighbors of v. There are less than $(1+\delta)^{gn(\mathsf{dl}(v))}$ such neighbors that we need to consider. Namely, there are less than $(1+\delta)^{gn(\mathsf{dl}(v))}$ neighbors in levels $\geq \mathsf{dl}(v)$ that we need to consider for the potential increase. This is due to the fact that we moved v to the *highest* level that satisfies the invariants. If v has $\geq (1+\delta)^{gn(\mathsf{dl}(v))}$ neighbors in $Z_{\mathsf{dl}(v)}$, then the desire-level of v would be $\mathsf{dl}(v)+1$ since v satisfies Invariant 2 at level $\mathsf{dl}(v)+1$ and we can increase its $\mathsf{dl}(v)$. Furthermore, we only need to consider neighbors in levels $\geq \mathsf{dl}(v)$ since only these neighbors will contribute to the potential increase by Eq. (2) and Eq. (3).

We first consider the increase in Ψ. The total potential increase in $\Psi(u,v)$ (Eq. (3)) summed over the increase for every edge (u,v) where $\ell(u) \geq \mathsf{dl}(v)$ is at most $2(\ell(v)-\mathsf{dl}(v))(1+\delta)^{gn(\mathsf{dl}(v))}$. This is due to the fact that for each edge (u,v), the potential gain from Ψ is upper bounded by 2 for every level in $[\mathsf{dl}(v), \ell(v)-1]$. Thus, in total over $< (1+\delta)^{gn(\mathsf{dl}(v))}$ such neighbors results in a total potential increase of less than $2(\ell(v)-\mathsf{dl}(v))(1+\delta)^{gn(\mathsf{dl}(v))}$.

Now we consider the potential increase in Φ. For this potential increase, we need to account for the increase in potential of every neighbor whose edge is flipped by the move. Decreasing the degree of each neighbor by one for each of $|N(v, \mathsf{dl}(v))| < (1+\delta)^{gn(\mathsf{dl}(v))}$ neighbors results in the total increase in Φ. In other words, for each flipped edge (v,w), $N(w, Z_i)$ decreases by 1 for each level $i \in [\mathsf{dl}(v)+1, \ell(v)]$. The total increase in Φ is then at most $\lambda(\ell(v) - \mathsf{dl}(v))(1+\delta)^{gn(\mathsf{dl}(v))}$ by Eq. (2) over all flipped edges since there are less than $(1+\delta)^{gn(\mathsf{dl}(v))}$ such neighbors in levels $\geq \mathsf{dl}(v)+1$ and so the total number of flipped edges is less than $(1+\delta)^{gn(\mathsf{dl}(v))}$.

Then, in total, the potential decrease is at least

$$\left(\sum_{i=\mathsf{dl}(v)}^{\ell(v)-1} (\lambda+3)(1+\delta)^{gn(i)}\right) - 2(\ell(v)-\mathsf{dl}(v))(1+\delta)^{gn(\mathsf{dl}(v))} - \lambda(\ell(v)-\mathsf{dl}(v))(1+\delta)^{gn(\mathsf{dl}(v))} \geq (\ell(v)-\mathsf{dl}(v))(1+\delta)^{gn(\mathsf{dl}(v))}$$

which is enough to pay for the at most $(1+\delta)^{gn(\mathsf{dl}(v))}$ edge flips as well as the $O(\ell(v)-\mathsf{dl}(v))$ work for computing the desire-level. The total number of edge flips is upper bounded by $|N(v, \mathsf{dl}(v))|$. Since we moved v to $\mathsf{dl}(v)$ and not $\mathsf{dl}(v)+1$, we know that v satisfies Invariant 2 at $\mathsf{dl}(v)$ and not at $\mathsf{dl}(v)+1$. Then, this means that $|N(v, \mathsf{dl}(v))| < (1+\delta)^{gn(\mathsf{dl}(v))}$. Hence, our number of edge flips is also bounded by $(1+\delta)^{gn(\mathsf{dl}(v))}$.

A vertex v moves from level i to level $i+1$ due to Algorithm 2. In order for Algorithm 2 to move a vertex from level i to $i+1$, it must have violated Invariant 1 and that $\mathsf{up}(v) > (2+3/\lambda)(1+\delta)^{gn(i)}$ before the move. Before and after the move, $\Phi(v) = 0$, since in these cases $\mathsf{up}^*(v) > (2+3/\lambda)\,(1+\delta)^{gn(i-1)}$ and $\mathsf{up}^*(v) > (2+3/\lambda)\,(1+\delta)^{gn(i)}$, respectively. Thus, $\Phi(v)$ does not change in value. Furthermore, the $\Phi(w)$ of its neighbors w cannot increase. Then, this leaves us with the potential change in $\Psi(v,w)$.

Z_i is the set of neighbors that v has to iterate through within its data structures if v goes up a level. The potential decrease for every neighbor of v on $i = \ell(v)$ is 1. The potential decrease for every neighbor on level $i+1$ is 1. Finally, the potential decrease for every neighbor in levels $> \mathsf{dl}(v)$ is 2. Then, the potential decrease for every neighbor in Z_i is at least 1 and is enough to pay for the $O(|Z_i|)$ cost of iterating and moving the neighbors of v in its data structures.

Parallel Amortized Work. The last part of the proof that needs to be shown is that any set of parallel level data structure operations that is undertaken by Algorithm 2 or Algorithm 3 has a sequential set of operations of the form detailed above (i.e., moving v to $\mathsf{dl}(v)$ or moving v from level i to $i+1$) that consists of the same or strictly larger set of operations.

Lemma 5.9. *For any set of operations performed in parallel by Algorithm 2 or Algorithm 3, there exists an identical set of sequential operations to the set of parallel operations.*

Proof. In Algorithm 2, the parallel set of operations consists of moving all vertices that violate Invariant 1 in the same level i up to level $i+1$. Again, suppose we choose an arbitrary order to move the vertices in level i to level $i+1$. Given two neighbors in the order v and w, if v moves to level $i+1$, the up-degree of w still includes v; since the up-degree of any vertex w is not affected by the previous vertices that moved to level $i+1$, w moves to $i+1$ on its turn. This order provides a sequential set of operations that is equivalent to the parallel set of operations.

In Algorithm 3, the parallel set of operations consists of moving a set of vertices down from arbitrary levels to the same level i. We show that there exists an identical set of sequential operations to the parallel operations. First, any vertex whose $\mathsf{dl}(v) = i$ considered all vertices in levels $\geq i-1$ in its calculation of $\mathsf{dl}(v)$. Thus, any other vertex w moving from a level $j > i$ to level i is included in calculating the desire-level of vertex v. Suppose we pick an arbitrary order to move the vertices that have $\mathsf{dl}(v) = i$ to level i. Then, the desire-level of any vertex w whose $\mathsf{dl}(w) = i$ does not change after v is moved to level i. Hence, when it is w's turn in the order, w moves to level i. This arbitrary order is a sequential set of operations that is identical to the parallel set of operations. □

Lemma 5.10. *For a batch of $|\mathcal{B}| < m$ updates, Algorithm 1 requires $O(|\mathcal{B}| \log^2 n)$ amortized work with high probability. The required space is $O(n \log^2 n + m)$ using the randomized data structures.*

Proof. Our potential argument handles the cost of moving neighbors of a vertex v between different levels. Namely, our potential argument shows that such costs of updating neighbor lists of nodes require $O(\log^2 n)$ amortized work per edge update to the structure since we showed that the $O(\log^2 n)$ potential increase from each edge insertion or deletion is enough to pay for the cost of moving vertices to different levels.

Then, it remains to calculate the amount of work of Algorithm 4. We can obtain the size of each neighbor list in $O(1)$ work and depth. If we show that the work of running Algorithm 4 is asymptotically bounded by the work of calculating the set of neighbor vertices that need to be moved between neighbor lists for a vertex, then we can also charge this work to the potential. To compute the first lower bound on $\mathsf{dl}(v)$, we maintain a cumulative sum of the total number of neighbors for each vertex at or below the current level $\ell(v)$. Then, we sequentially double the number of elements we use to compute the next level. We use $O(\ell - \mathsf{dl}(v))$ work to compute $\mathsf{dl}(v)$.

Finally, we also bound the work of the final binary search. Let R be the size of the range of values in which we perform our binary search. The size of the number of possible levels becomes smaller as we decrease our range of values to search. Whenever we go right in the binary search, we perform $R/2$ work. Whenever we go left in the binary search, we also perform at most $R/2$ work. Thus, the total amount of work we perform while doing the binary search is $O(R)$. And by the argument above, the amount of work is $O\left(|Z_{\mathsf{dl}(v)} \setminus Z_{\ell(v)}|\right)$.

The total work of Algorithm 4 is $O(|Z_{\mathsf{dl}(v)} \setminus Z_{\ell(v)}| + (\ell - \mathsf{dl}(v)))$ which we can successfully charge to the potential. We conclude that the amount of work per update is $O(\log^2 n)$. □

5.6 Estimating the Coreness and Orientation

$(2+\varepsilon)$-Approximation of Coreness. The *coreness estimate*, $\hat{k}(v)$, is an estimate of the coreness of a vertex v. We compute a coreness estimate using *only* v's level and the number of levels per group (which is fixed). We show how to use such information to obtain a $(2+\varepsilon)$-approximation to the actual coreness of v for any constant $\varepsilon > 0$. (We can find an approximation for any fixed ε by appropriately setting δ and λ.) To calculate $\hat{k}(v)$, we find the largest index i of a group g_i, where $\ell(v)$ is at least as high as the highest level in g_i.

Definition 5.11 (Coreness Estimate)**.** *The* coreness estimate $\hat{k}(v)$ *of vertex v is $(1+\delta)^{\max(\lfloor(\ell(v)+1)/4\lceil\log_{1+\delta} n\rceil\rfloor-1,0)}$, where each group has $4\lceil\log_{(1+\delta)} n\rceil$ levels.*

To see an example, consider vertex y in Fig. 7 (e). We estimate $\hat{k}(y) = 1$ since the highest level that is the last level of a group and is equal to or below level $\ell(y) = 4$ is level 2. Level 2 is part of group 0, and so our coreness estimate for y is $(1+\delta)^0 = 1$. This is a 2-approximation of its actual coreness of 2. Using Definition 5.11, we prove that our PLDS maintains a $(2+3/\lambda)(1+\delta)$-approximation of the coreness value of each vertex, for any constants $\lambda > 0$ and $\delta > 0$. Therefore, we obtain the following lemma giving the desired $(2+\varepsilon)$-approximation. Our experimental analysis shows that our theoretical bounds limit the maximum error of our experiments, although our average errors are much smaller. To get a maximum error bound of $(2+\varepsilon)$ for any $\varepsilon > 0$, we can set $\delta = \varepsilon/3$ and $\lambda = \frac{9}{\varepsilon} + 3$.

By Lemma 5.13, it suffices to return $\hat{k}(v)$ as the estimate of the coreness of v; this proves the approximation factor in Theorem 3.1.

Lemma 5.12. *Let $\hat{k}(v)$ be the coreness estimate and $k(v)$ be the coreness of v, respectively. If $k(v) > (2+3/\lambda)(1+\delta)^{g'}$, then $\hat{k}(v) \geq (1+\delta)^{g'}$. Otherwise, if $k(v) < \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$, then $\hat{k}(v) < (1+\delta)^{g'}$.*

Proof. For simplicity, we assume the number of levels per group is $4\lceil\log_{(1+\delta)} m\rceil + 1$ (a tighter analysis can accommodate the case when the number of levels per group is $\lceil\log_{(1+\delta)} m\rceil$). Let $T(g')$ be the topmost level of group g'. In the first case, we show that if $k(v) > (2+3/\lambda)(1+\delta)^{g'}$, then v would be in a level higher than $T(g')$ in our level data structure. This would also imply that $\hat{k}(v) \geq (1+\delta)^{g'}$. Suppose for the sake of contradiction that v is located at some level $\ell(v)$ where $\ell(v) \leq T(g')$. This means that $\mathsf{up}(v) \leq (2+3/\lambda)(1+\delta)^{g'}$ at level $\ell(v)$. Furthermore, by the invariants of our level data structure, each vertex w at the same or lower level has $\mathsf{up}(w) \leq (2+3/\lambda)(1+\delta)^{g'}$. This means that when we remove all vertices starting at level 0 sequentially up to and including $\ell(v)$, all vertices removed have degree $\leq (2+3/\lambda)(1+\delta)^{g'}$ when removed. Thus, when we reach $\ell(v)$, v also has degree $\leq (2+3/\lambda)(1+\delta)^{g'}$. This is a contradiction with $k(v) > (2+3/\lambda)(1+\delta)^{g'}$. It must then be the case that v is at a level higher than $T(g')$ and $\hat{k}(v) \geq (1+\delta)^{g'}$.

Now we prove that if $k(v) < \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$, then $\hat{k}(v) < (1+\delta)^{g'}$. We assume for sake of contradiction that $k(v) < \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$ and $\hat{k}(v) \geq (1+\delta)^{g'}$. To prove this case, we consider the following process, which we call the *pruning* process. Pruning is done on a subgraph $S \subseteq G$. We use the notation $d_S(v)$ to denote the degree of v in the subgraph induced by S. For a given subgraph S, we *prune* S by repeatedly removing all vertices v in S whose $d_S(v) < \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$. Note that in this argument, we need only consider levels from the same group g' before we reach a contradiction, so we assume that all levels are in the group g'. Let j represent the number of levels below level $T(g')$. (Recall that because $\hat{k}(v) \geq (1+\delta)^{g'}$, $\ell(v) \geq T(g')$, if we consider a level $\ell(v) > T(g')$, then the up*-degree cannot decrease due to Invariant 2 becoming stricter. This only makes our proof easier, and so for simplicity, we consider $\ell(v) = T(g')$.) We prove

via induction that the number of vertices pruned from the subgraph induced by $Z_{T(g')-j}$ *must* be at least

$$\left(\frac{(2+3/\lambda)(1+\delta)}{2}\right)^{j-1}\left((1+\delta)^{g'}-\frac{(1+\delta)^{g'}}{(2+3/\lambda)\,(1+\delta)}\right)$$

or otherwise, $k(v) \geq \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$. We first prove the base case when $j=1$. In the base case, we know that $d_{Z_{T(g')-1}}(v) \geq (1+\delta)^{g'}$ by Invariant 2. Then, if fewer than $(1+\delta)^{g'}-\frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$ neighbors of v are pruned from the graph, then v is part of a $\geq \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$-core and $k(v) \geq \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$, a contradiction.

Thus, at least $(1+\delta)^{g'}-\frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$ vertices must be pruned in this case. We now assume the induction hypothesis for j and prove that this is true for step $j+1$. By Invariant 2, each vertex on level $T(g')-j$ and above has degree *at least* $(1+\delta)^{g'}$ in graph $Z_{T(g')-j-1}$. Then, in order to prune all X vertices from the previous induction step, we must prune at least $\frac{(1+\delta)^{g'}X}{2}$ edges, since each edge decreases the degree of two vertices by 1; all adjacent edges of a pruned vertex are also pruned/removed. Each vertex that is pruned can remove *at most* $\frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$ edges when it is pruned, by definition of our pruning procedure since we prune vertices with degree $< \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$. Thus, the *minimum* number of vertices we must prune in order to prune the $X=\left(\frac{(2+3/\lambda)(1+\delta)}{2}\right)^{j-1}\left((1+\delta)^{g'}-\frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}\right)$ vertices from the previous step is

$$\frac{\#\text{ edges that need to be pruned}}{\max\#\text{ edges pruned per pruned vertex}}=\frac{(1+\delta)^{g'}X}{2\frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}}$$

$$=\frac{(2+3/\lambda)(1+\delta)}{2}X$$

$$=\left(\frac{(2+3/\lambda)(1+\delta)}{2}\right)^{j}\left((1+\delta)^{g'}-\frac{(1+\delta)^{g'}}{(2+3/\lambda)\,(1+\delta)}\right).$$

Thus, we have proven our argument for the $(j+1)$-st induction step. Note that for $j=\lceil\log_{(2+3/\lambda)(1+\delta)/2}(4m+1)\rceil$, we have $j \leq 4\lceil\log_{(1+\delta)}(m)\rceil+1$. This is because, since we pick λ to be a constant greater than 0, $2+3/\lambda > 2$ and for large enough m, $\log_{(2+3/\lambda)(1+\delta)/2}(4m+1) \leq 4\lceil\log_{(1+\delta)}(m)\rceil+1$. Then, by our induction, if we substitute $4\lceil\log_{(1+\delta)}(m)\rceil+1$ for j,

$$\left(\frac{(2+3/\lambda)(1+\delta)}{2}\right)^{4\lceil\log_{(1+\delta)}(m)\rceil}\left((1+\delta)^{g'}-\frac{(1+\delta)^{g'}}{(2+3/\lambda)\,(1+\delta)}\right)$$
$$> 4m\cdot(1/2)=2m.$$

This means we must prune at least $2m+1$ vertices at this step, which we cannot because there are at most $2m$ vertices in a level that is not level 0. This last step holds because all vertices with degree 0 must be on the first level. Hence, all vertices not on level 0 must be adjacent to at least one edge, and $n \leq 2m$ where n is the number of vertices on the level that is not level 0. Thus, our assumption is incorrect and we have proven our desired property. □

We show that Lemma 5.12 implies Lemma 5.13.

Lemma 5.13. *The coreness estimate $\hat{k}(v)$ of a vertex v satisfies $\frac{k(v)}{(2+\varepsilon)} \leq \hat{k}(v) \leq (2+\varepsilon)k(v)$ for any constant $\varepsilon > 0$.*

Proof. Suppose $\hat{k}(v)=(1+\delta)^{g}$. Then, by Lemma 5.12, we have $\frac{(1+\delta)^{g}}{(2+3/\lambda)(1+\delta)} \leq k(v) \leq (2+3/\lambda)\,(1+\delta)^{g+1}$. Then, substituting $\hat{k}(v)=(1+\delta)^{g}$ and solving the above bounds, $\frac{k(v)}{(2+3/\lambda)(1+\delta)} \leq \hat{k}(v) \leq (2+3/\lambda)\,(1+\delta)k(v)$. For any constant $\varepsilon > 0$, there exists constants $\lambda, \delta > 0$ where $\frac{k(v)}{2(1+\varepsilon)} \leq \hat{k}(v) \leq 2(1+\varepsilon)k(v)$. □

For arbitrary batch sizes, getting better than a 2-approximation for coreness values is P-complete [5], and so there is unlikely to exist a polylogarithmic-depth algorithm with such guarantees.

Proof of Theorem 3.2. The approximation factor for our algorithm is given by Lemma 5.13. The work and depth bounds of our algorithm is given by Lemma 5.10 and Lemma 5.7. Altogether, we prove our main theorem. □

5.7 $O(\alpha)$ Out-Degree Orientation

We orient all edges from vertices in lower levels to higher levels, breaking ties for vertices on the same level by using their indices. Such an orientation can be maintained dynamically in the same work and depth as our PLDS via a parallel hash table keyed by the edges and where the values give the orientation. Specifically, we require the following data structures for maintaining a low out-degree orientation. First, we maintain a parallel hash table, H, containing the edges of the graph. The edge (u,v) is the key in the hash table where $u \prec v$ (i.e. the index of u is less than the index of v). The value for key (u,v) is 0 if the edge is oriented from u to v and 1 if the edge is oriented from v to u. The pseudocode is shown in Algorithm 5. Additionally, we make a slight modification to our update algorithm that keeps track of the edges that were searched when a vertex moves to a higher or lower level. The pseudocode for our algorithm is given in Algorithm 5.

Proof of Corollary 3.3. Let the degeneracy of the graph be d. As is well-known, the degeneracy of the graph is equal to k_{max} where k_{max} is the maximum k-core of the graph. Furthermore, it is well-known that $\frac{d}{2} \leq \alpha \leq d$. By Lemma 5.13, the vertices in the largest k-core in the graph are in a level with group number at most $\log_{(1+\delta)}((2+3/\lambda)(1+\delta)d)+1$. This means that the up-degree of each vertex in that group is at most $(2+3/\lambda)(1+\delta)^{\log_{(1+\delta)}((2+3/\lambda)(1+\delta)d)}=(4+\varepsilon)d$ for any constant $\varepsilon > 0$ for appropriate settings of $\lambda, \delta > 0$. We then also obtain an $(8+\varepsilon)\alpha$ out-degree orientation where α is the arboricity of the graph. □

5.8 Deterministic and Space-Efficient Data Structures

In addition to the randomized data structures presented in Section 3.4, we present two additional sets of data structures that we can use to obtain a *deterministic* and a *space-efficient* $(2+\varepsilon)$-approximate k-core algorithms.

Algorithm 5 LowOutdegreeOrient($\mathcal{B}$)

Input: A batch $\mathcal{B}$ of updates.

Output: A set of edges F that were flipped after processing the batch of updates. An edge $(u, v) \in F$ represents the orientation of the edge *before* the flip. Also returns oriented updates $(u, v) \in \mathcal{B}$ where for edge deletions (u, v) is the orientation of the edge *before* the deletion and for edge insertions (u, v) is the orientation of the edge *after* the insertion.

$F \leftarrow \emptyset$.
parfor each searched edge (u, v) for a vertex that moved levels **do**
  if $H[(u, v)] = 0$ and $((\ell(u) > \ell(v)$ or $(\ell(u) = \ell(v)$ and $v < u))$ **then**
    $F \leftarrow F \cup (u, v)$.
  else if $H[(u, v)] = 1$ and $(\ell(v) > \ell(u)$ or $(\ell(u) = \ell(v)$ and $u < v))$ **then**
    $F \leftarrow F \cup (u, v)$.
$J \leftarrow \emptyset$.
parfor each edge update $\{u, v\} \in \mathcal{B}$ **do**
  if $\{u, v\}$ is an insertion **then**
    Add to J the orientation of edge *after* processing $\mathcal{B}$.
  else
    Add to J the orientation of edge *before* processing $\mathcal{B}$.
return F, J.

The work of all of our randomized, deterministic, and space-efficient algorithms are the same; however, using randomization allows us to obtain a better depth with slightly less complicated data structures.

Deterministic Data Structures. We initialize an array U, of size n. Each vertex is assigned a unique index in U. The entry for the i'th vertex, $U[i]$, contains a pointer to a dynamic array that stores the neighbors of vertex v_i at levels $\geq \ell(v_i)$. Each vertex v_i also stores another dynamic array, L_{v_i}, that contains pointers to a set of dynamic arrays storing the neighbors of v_i partitioned by their levels j where $j < \ell(v_i)$. Specifically, we maintain a separate dynamic array for each level from level 0 to level $\ell(v_i) - 1$ storing the neighbors of v_i at each respective level. We also maintain the current level of each vertex in an array.

To perform a batch of insertions into a dynamic array, we insert the elements at the end of the array. The array is resized and doubles in size if too many elements are inserted into the array (and it exceeds its current size). For a batch of deletions, the deletions are initially marked with a "deleted" marker indicating that the element in the slot has been deleted. A counter is used to maintain how many slots contain "deleted." Then, once a constant fraction of elements (e.g. 1/2) has "deleted" marked in their slots, the array is cleaned up by reassigning vertices to new slots and resizing the array.

Lemma 5.14. *Algorithm 1 returns a deterministic level data structure that maintains Invariant 1 and Invariant 2 and has $O(\log^3 n)$ worst-case depth and $O(n \log^2 n + m)$ space.*

Proof. All edge updates can be partitioned into $\mathcal{B}_{ins}$ and $\mathcal{B}_{del}$ in parallel in $O(\log n)$ depth. Then, it remains to bound the depth of Algorithm 2 and Algorithm 3.

Algorithm 2 iterates through all $K = O(\log^2 n)$ levels sequentially. By Lemma 5.5, no vertices on levels $\leq i$ will violate Invariant 1 after processing level i. Thus, by the end of the procedure no vertices violate Invariant 1. By Observation 5.3, Invariant 2 was never violated during Algorithm 2. Thus, both invariants are maintained at the end of the algorithm. Since we iterate through $O(\log^2 n)$ levels and, in each level, we require checking the neighbors at one additional level which can be done in parallel in $O(1)$ depth, the total depth of this procedure is $O(\log^2 n)$. For each level, an additional depth of $O(\log n)$ might be necessary to compute the element offsets and then resize the arrays. Then, Algorithm 2 requires $O(\log^3 n)$ worst-case depth.

Algorithm 3 iterates through all $K = O(\log^2 n)$ levels sequentially. By Observation 5.4 and Lemma 5.6, after processing level i, no vertices on a level higher than $i+1$ will have $\mathsf{dl}(v) \leq i+1$ and no vertices on levels $\leq i$ will violate Invariant 1. Thus, by the end of the procedure all vertices satisfy Invariant 2. Furthermore, Invariant 1 was never violated due to Observation 5.4. There are $O(\log^2 n)$ levels and for each level we require running Algorithm 4 to obtain the $\mathsf{dl}(v)$ of each affected vertex v that should be moved to each level.

Running Algorithm 4 requires $O(\log \log n)$ depth to obtain the first level that satisfies invariants for each affected vertex v and $O(\log \log n)$ depth for the final binary search that determines the closest level to $\ell(v)$ that satisfies the invariants. In conclusion, Algorithm 3 requires $O(\log^3 n)$ worst-case depth.

Altogether, Algorithm 1 requires $O(\log^3 n)$ worst-case depth. □

$O(m)$ Total Space Data Structures. Here we describe how to reduce the total space usage of our data structures to $O(m)$. All of our previous data structures use $O(n \log^2 n + m)$ space, which means that when $m = O(n)$, we use space that is superlinear in the size of the graph. To reduce the total space to $O(m)$, we maintain two structures for L_{v_i}. We can use either the deterministic or randomized structures for the other structures. Each L_{v_i} is maintained as a linked list. The j'th node in the linked list maintains the number of neighbors of v_i at the j'th non-empty level (a non-empty level is one where v_i has neighbors at that level) that is less than $\ell(v_i)$. The node representing a level is removed from the linked list when the level becomes empty. Each node in L_{v_i} contains pointers to vertices at the level represented by the node. Each vertex then contains pointers to every edge it is adjacent to and every edge contains pointers to the two nodes in the two linked lists representing the levels on which the endpoints of the edge reside. Using either dynamic arrays or hash tables for the lists of neighbors allow us to maintain these data structures in $O(m)$ space. Since we only maintain a node in our linked list for every non-empty level, our linked list contains $O(m)$ nodes.

Using the data structures above, we can prove equivalent results to Theorem 3.1.

Lemma 5.15. *Algorithm 1 returns a deterministic level data structure that maintains Invariant 1 and Invariant 2 and has $O(\log^4 m)$ depth, while using $O(m)$ space.*

Proof. The proof is the same as the proof of Lemma 5.14 except that we replace Algorithm 4 with a linear search in the linked list, which has size at most the number of levels, which is $O(\log^2 n)$. The specific data structure we use for each vertex v is a linked list with each node of the linked list representing a level $\leq \ell(v) - 1$ which contains one or more neighbors of v. Then, each node in the linked list contains a pointer to a dynamic array containing the neighbors in that level. Thus, the total depth is $O(\log^4 n)$. □

5.8.1 Overall Work and Depth Bounds. Our deterministic and space-efficient structures also give the following corollary using our above depth bound arguments.

Using Lemma 5.14 and Lemma 5.15, we obtain the following two corollaries.

Corollary 5.16. *For a batch of $|\mathcal{B}| < m$ updates, Algorithm 1 returns a PLDS that maintains Invariant 1 and Invariant 2 in $O(|\mathcal{B}| \log^2 n)$ amortized work and $O(\log^3 n)$ depth, using $O(n \log^2 n + m)$ space.*

Corollary 5.17. *For a batch of $|\mathcal{B}| < m$ updates, Algorithm 1 returns a PLDS that maintains Invariant 1 and Invariant 2 in $O(|\mathcal{B}| \log^2 n)$ amortized work and $O(\log^4 m)$ depth, using $O(m)$ space.*

5.9 Handling Vertex Insertions and Deletions

We can handle vertex insertions and deletions by inserting vertices that have zero degree and considering deletions of vertices to be a batch of edge deletions of all edges adjacent to the deleted vertex. When we insert a vertex with zero degree, it automatically gets added to level 0 and remains in level 0 until edges incident to the vertex are inserted. For a vertex deletion, we add all edges incident to the deleted vertex to a batch of edge deletions. Note, first, that all vertices which have 0 degree will remain in level 0. Thus, there are at most $O(m)$ vertices which have non-zero degree.

In this setting, we may need to rebuild the PLDS from scratch. Instead of maintaining $\lceil 4 \log^2 n \rceil$ levels, we maintain $\lceil 8 \log^2 n \rceil$ levels in this setting. Doubling the number of levels is a very loose bound to ensure that we can handle two times the number of vertices in the graph before we perform a rebuild of our entire structure. To maintain $O(\log^2 n)$ levels in our data structure, we rebuild the data structure once we have made $n/2$ vertex updates. Rebuilding the data structure requires $O(n \log^2 n)$ total work which we can amortize to the $n/2$ vertex updates to obtain $O(\log^2 n)$ amortized work w.h.p. Running Algorithm 2 and Algorithm 3 on the entire set of $O(n + m)$ vertices and edges requires $O(\text{poly} \log n)$ depth w.h.p. depending on the specific set of data structures we use.

Lastly, in order to obtain a set of vertices which are re-numbered consecutively (in order to maintain our space bounds), we perform parallel integer sort or hashing.

6 EXPERIMENTAL EVALUATION

In this section, we compare the performance of our dynamic PLDS with existing approaches on a set of large real-world graphs. Our results show that our algorithms consistently achieve speedups, by up to two orders of magnitude, compared with all of the previous state-of-the-art dynamic k-core decomposition algorithms.

Evaluated Algorithms. We evaluate two versions of our algorithm: ***PLDS***: an exact implementation of our theoretical algorithm and ***PLDSOpt***: a version with $\lceil \log_{1+\delta} n/50 \rceil$ levels per group. ***PLDS*** maintains the approximation guarantees given by Lemma 5.13, while ***PLDSOpt*** achieves better performance while maintaining slightly worse approximation bounds.

We compare our algorithms with the following *dynamic* implementations: ***Sun***: the sequential, approximate algorithm of Sun et al. [83], specifically their faster, round-indexing algorithm, which is publicly available [84]; ***Hua***: the parallel, exact algorithm of Hua et al. [48], kindly provided by the authors; ***Zhang***: the sequential, exact algorithm of Zhang and Yu [93], kindly provided by the authors; and ***LDS***: our implementation of the sequential, approximate algorithm of Henzinger et al. [47], but using our coreness approximation procedure in Section 5.6. All are state-of-the-art algorithms, outperforming previous algorithms in their respective categories.

We also implemented ***ApproxKCore***, our new static parallel approximate k-core decomposition algorithm (Theorem 3.8). We compared it with ***ExactKCore***, the state-of-the-art parallel, static, exact k-core algorithm of Dhulipala et al. [27].

Table 3: Graph sizes and largest values of k for k-core decomposition.

Graph Dataset	Num. Vertices	Num. Edges	Largest value of k
dblp	317,080	1,049,866	101
brain	784,262	267,844,669	1200
wiki	1,094,018	2,787,967	124
youtube	1,138,499	2,990,443	51
stackoverflow	2,584,164	28,183,518	163
livejournal	4,846,609	42,851,237	329
orkut	3,072,441	117,185,083	253
ctr	14,081,816	16,933,413	2
usa	23,947,347	28,854,312	3
twitter	41,652,230	1,202,513,046	2484
friendster	65,608,366	1,806,067,135	304

Setup. We use `c2-standard-60` Google Cloud instances (3.1 GHz Intel Xeon Cascade Lake CPUs with a total of 30 cores with two-way hyper-threading, and 236 GiB RAM) and `m1-megamem-96` Google Cloud instances (2.0 GHz Intel Xeon Skylake CPUs with a total of 48 cores with two-way hyper-threading, and 1433.6 GB RAM). We use hyper-threading in our parallel experiments by default. Our programs are written in C++, use a work-stealing scheduler [15], and are compiled using `g++` (version 7.5.0) with the `-O3` flag. We terminate experiments that take over 3 hours. PLDS and PLDSOpt finished within 3 hours for all experiments.

Datasets. We test our algorithms on 11 real-world undirected graphs from SNAP [61], the DIMACS Shortest Paths challenge road networks [26], and the Network Repository [77], namely ***dblp***, ***brain***, ***wiki***, ***orkut***, ***friendster***, ***stackoverflow***, ***usa***, ***ctr***, ***youtube***, and ***livejournal***. We also used ***twitter***, a symmetrized version of the Twitter network [59]. We remove duplicate edges, zero-degree vertices, and self-loops. Table 3 reflects the graph sizes *after* this removal, and gives the largest k-core values. Both *stackoverflow* and *wiki* are temporal networks; for these, we maintain the edge insertions and deletions in the temporal order from SNAP. *usa* and *ctr* are two high-diameter road networks and *brain* is a highly dense human brain network from NeuroData (https://neurodata.io/). All experiments are run on the `c2-standard-60` instances, except for *twitter* and *friendster*, which are run on the `m1-megamem-96` instances as they require more memory.

Ins/Del/Mix Experiments. Our experiments are run for *three different types of batched updates*, referred to by: (1) **Ins**: starting with an empty graph, *all* edges are inserted in multiple size $|\mathcal{B}|$ batches of insertion updates, (2) **Del**: starting with the original graph, *all* edges are deleted in multiple size $|\mathcal{B}|$ batches of deletion updates, and (3) **Mix**: starting with the initial graph minus a random set I of $|\mathcal{B}|/2$ edges, a set D of $|\mathcal{B}|/2$ random edges is chosen among the edges in the graph; then, a single size $|\mathcal{B}|$ mixed batch of updates with insertions I and deletions D is applied. For the temporal graphs, *stackoverflow* and *wiki*, the order of updates in the batches follows the order in SNAP [61]. For the rest, updates are generated by

taking two random permutations of the edge list, one for **Ins** and one for **Del**. Batches are generated by taking regular intervals of the permuted lists. For **Mix**, I and D are chosen uniformly at random.

Some past works only ran experiments in the **Mix** setting [48, 93], while others [83] also consider **Ins** and **Del**. In this paper, we run experiments in all three settings. For **Ins** and **Del**, we consider the average running time across all batches as a good indicator of how well the algorithm performs. For **Mix**, we test each algorithm and dataset 3 rounds each and take the average.

We use the original timing functions provided by Hua, Sun, Zhang, and ExactKCore. We use the original code of Hua and Zhang for **Mix** and modify their code to perform **Ins** and **Del**. We note that Hua's timing function does not include the time to process the graph and maintain their data structures; we include all such times in our code. All other benchmarks also include this time. If we include this time in their implementation, their running times increase by up to 8× for some experiments. This explains some of Hua's experimental performance improvements over the other benchmarks.

The static algorithms, ExactKCore and ApproxKCore, are rerun on the entire graph after each batch of updates in **Ins** and **Del**. For the **Mix** batch, we order all insertions in the batch before all deletions. Then, we generate two static graphs per batch, one following all insertions, and the other following all deletions. We rerun the static algorithms on each static graph and take the average of the times to obtain comparable per-batch running times. We do this because some of the deletion updates may cancel the insertion updates in the batch.

6.1 PLDS Implementation Details

We implemented our algorithms using the primitives from the Graph Based Benchmark Suite [28]. We implemented the PLDS with work, depth, and space bounds given in Theorem 3.1. One can choose to instead implement our space-efficient version of our data structure in exchange for additional poly$(\log n)$ factors in the theoretical depth.

Our data structure uses concurrent hash tables with linear probing [81], which support x concurrent insertions, deletions, or finds in $O(x)$ amortized work and $O(\log^* x)$ depth w.h.p. [42]. For deletions, we used the folklore *tombstone* method: when an element is deleted, we mark the slot in the table as a tombstone, which can be reused, or cleared during a table resize. We also use dynamic arrays, which support adding or deleting x elements from the end in $O(x)$ amortized work and $O(1)$ depth.

We first assign each vertex a unique ID in $[n]$. Then, we maintain an array U of size n keyed by vertex ID that returns a parallel hash table containing neighbors of v on levels $\geq \ell(v)$. For each vertex v, we maintain a dynamic array L_v keyed by indices $i \in [0, \ell(v) - 1]$. The i'th entry of the array contains a pointer to a parallel hash table containing the neighbors of v in level i. Appropriate pointers exist that allow $O(1)$ work to access elements in structures. Furthermore, we maintain a hash table which contains pointers to vertices v where $\mathrm{dl}(v) \neq \ell(v)$, partitioned by their levels. This allows us to quickly determine which vertices to move up (in Algorithm 2) or move down (in Algorithm 3).

We make one modification in our parallel implementation of our insertion procedure from our theoretical algorithm which is instead of moving vertices up level-by-level, we perform a parallel filter and sort that calculates the desire-level of vertices we move up. This results in more work theoretically, but we find that, practically, it results in faster runtimes. Also, notably, in practice, we optimized the performance of our PLDS by considering $\lceil \frac{\log_{(1+\delta)} m}{50} \rceil$ levels per group instead of $\lceil \log_{(1+\delta)} m \rceil$. We also implemented a version of our structure that *exactly follows* our theoretical algorithm and compared the performance of both structures. We see that even such a simple optimization resulted in significant gains in performance, up to 23.89×.

6.2 Accuracy vs. Running Time

We start by evaluating the empirical error ratio of the per-vertex core estimates given by our implementations (PLDSOpt, PLDS, LDS) and Sun on *dblp* and *livejournal*, using batches of size 10^5 and 10^6, respectively. Fig. 8 shows the average batch time (in seconds) against the average and maximum *per-vertex* core estimate error ratio. This error ratio is computed as $\max\left(\frac{\hat{k}(v)}{k(v)}, \frac{k(v)}{\hat{k}(v)}\right)$ for each vertex v (where $\hat{k}(v)$ is the core estimate and $k(v)$ is the exact core value). The average is the error ratio averaged across all vertices and the maximum is the maximum error. If the exact core number is 0, we ignore the vertex in our error ratio since our algorithm guarantees an estimate of 0; for vertices of non-zero degree, the lowest estimated core number is 1 for all implementations.

The parameters we use for PLDSOpt, PLDS, and LDS are all combinations of $\delta = \{0.2, 0.4, 0.8, 1.6, 3.2, 6.4\}$ and $\lambda = \{3, 6, 12, 24, 48, 96\}$. We call these *theoretically-efficient parameters*, since they maintain the work-efficiency of our algorithms. For Sun, we use all combinations of their parameters $\varepsilon_{sun} = \lambda_{sun} = \{0.2, 0.4, 0.8, 1.6, 3.2\}$, and $\alpha_{sun} = \{2(1 + 3\varepsilon_{sun})\}$. We also tested $\alpha_{sun} = \{1.1, 2, 3.2\}$, as done in Sun et al.'s work [83]. When $\alpha = 1.1$, the theoretical efficiency bounds by Sun et al. [83] no longer hold, but they yield better estimates empirically. We compare this heuristic setting to a similar one in our algorithms, where we replace $(2 + 3/\lambda)$ with 1.1 in our code (where our efficiency bounds no longer hold) for $\delta = \{0.4, 0.8, 1.6, 3.2\}$. We refer to these as the *heuristic parameters*.

Fig. 8 shows that, using theoretically-efficient parameters, our PLDSOpt, PLDS, and LDS implementations are faster than Sun, Zhang, and Hua, for parameters that give similar average and maximum per-vertex core estimate error ratios. Furthermore, besides PLDS, PLDSOpt *outperforms all other algorithms*, regardless of approximation factor and error. This set of experiments demonstrates the flexibility of our algorithm; one can achieve smaller error at the cost of slightly increased runtime. However, as the experiments demonstrate, PLDSOpt still outperforms all other algorithms even when the parameters are tuned to give small error; this performance gain is maintained for **Ins**, **Del**, and **Mix**. Greater speedups are achieved on *livejournal* compared to *dblp*. Such a result is expected since larger batches allow for greater parallelism.

Concretely, compared with Zhang, PLDSOpt achieves 7.19–147.59×, 19.70–58.41×, and 9.75–142.79× speedups on **Ins**, **Del**, and **Mix** batches, respectively. Compared with Hua, PLDSOpt achieves 2.49–33.95×, 6.81–24.51×, and 2.94–21.77× speedups.

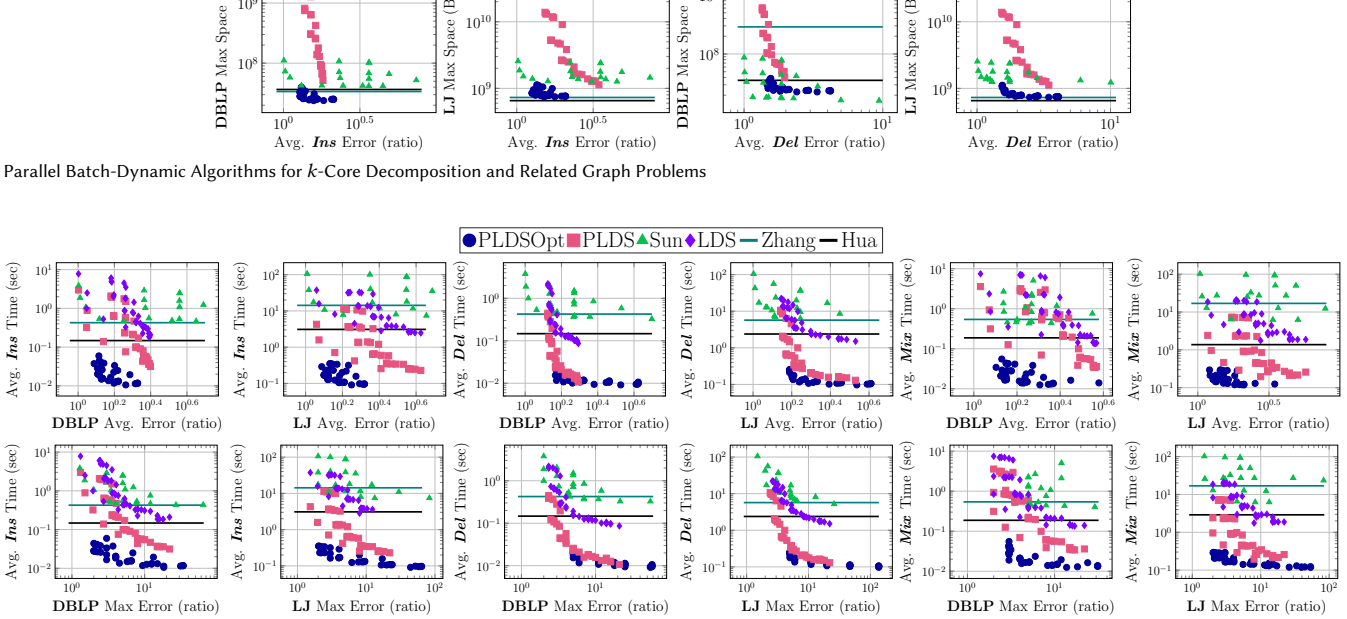


Figure 8: Comparison of the average per-batch time versus the average (top row) and maximum (bottom row) per-vertex core estimate error ratio of PLDSOpt, PLDS, Sun, and LDS, using varying parameters, on the *dblp* and *livejournal* graphs, with batch sizes 10^5 and 10^6, respectively. Experiments were run for Ins, Del, and Mix. The data uses theoretically-efficient parameters as well as the heuristic parameters where $(2+3/\lambda) = \alpha_{sun} = 1.1$. Runtimes for Hua and Zhang are shown as horizontal lines.

Against PLDS, PLDSOpt obtains 2.98–47.8×, 1.03–25.58×, and 1.5–76.94× speedups for **Ins**, **Del**, and **Mix**, respectively, on parameters that give similar approximations. Compared with Sun, on parameters that give similar theoretical guarantees and smaller empirical average error, PLDSOpt achieves 21.34–544.22×, 25.49–128.65×, and 19.04–248.36× speedups for **Ins**, **Del**, and **Mix**, respectively. Neither Zhang nor Hua guarantee polylogarithmic work. The peeling-based algorithm of Sun can have large depth and they do not provide a concrete bound on their amortized work for their faster, round-indexing implementation. Thus, the speedups we obtain over the benchmarks are due to the greater theoretical efficiency and because our algorithms are parallel.

Finally, PLDSOpt achieves average error in the ranges 1.26–2.13, 1.47–4.20, and 1.28–2.33 for **Ins**, **Del**, and **Mix**, respectively. PLDS gives comparable average errors in the ranges 1.27–4.22, 1.33–3.39, and 1.63–5.73, for **Ins**, **Del**, and **Mix**, respectively, while running slower than PLDSOpt for all parameters, despite the guarantee that the maximum error of PLDS is bounded by $(1+\delta)(2+3/\lambda)$ (Lemma 5.13). Thus, our optimized version allows us to obtain good error bounds empirically while drastically improving performance.

For *all of the remaining experiments*, set $\delta = 0.4$ and $\lambda = 3$.

6.3 Batch Size vs. Running Time

Fig. 9 shows the average per-batch running times for **Ins**, **Del**, and **Mix** on varying batch sizes for PLDSOpt, PLDS, Hua, LDS, and Zhang on *dblp* and *livejournal*. We do not run this experiment on Sun since their implementation does not have batching. Our experiments show that PLDSOpt is faster for all batch sizes except for the smallest **Del** and **Mix** batches.

Against PLDS, PLDSOpt achieves a speedup over all batches from 10.85–21.25×, 2.81–5.65×, and 10.42–29.28× for **Ins**, **Del**, and **Mix**, respectively, on *dblp* and 8.47–16.9×, 1.99–7.18×, and 1.9–15.26× for **Ins**, **Del**, and **Mix**, respectively, on *livejournal* for all but the batch of size 100 for **Del**. On the batch size of 100 , PLDS performs better than PLDSOpt by a 1.79× factor. Compared with Hua, PLDSOpt achieves speedups over all batches from 5.17–16.43×, 3.39–44.58×, and 2.53–13.05× for **Ins**, **Del**, and **Mix**, respectively, on *dblp* and 15.97–114.52×, 1.71–45.01×, and 9.10–19.82× for **Ins**, **Del**, and **Mix**, respectively, on *livejournal*. Compared with Zhang, PLDSOpt achieves speedups of 2.49–22.74×, 2.00–29.92×, and 2.95–21.57× for **Ins**, **Del**, and **Mix**, respectively, on *dblp*, and 31.53–95.33×, 1.25–73.19× and 4.26–87.05× for **Ins**, **Del**, and **Mix**, respectively, on *livejournal* on all but the smallest batches for **Del** and **Mix**. For **Del** with a batch size of 100, Zhang is the fastest with speedups of 1.46× and 6.86× over PLDSOpt on *dblp* and *livejournal*, respectively. For **Mix** with batch size 100, LDS is the fastest with speedups of 3.19× over PLDSOpt on *livejournal*. For small batch sizes, sequential algorithms perform better than parallel algorithms since the runtimes of parallel algorithms are dominated by parallel overheads.

6.4 Thread Count vs. Running Time

Fig. 10 shows the scalability of PLDSOpt, PLDS, and Hua with respect to their single-thread running times on *dblp* and *livejournal* using a batch size of 10^6. LDS, Sun, and Zhang are represented as horizontal lines since they are sequential. For **Ins**, **Del**, and **Mix** batches, PLDSOpt and PLDS achieve up to 30.28×, 32.02×, and 33.02×, and 26.46×, 25.33×, and 21.15×, self-relative speedup, respectively. Hua achieves up to a 3.6× self-relative speedup. We see that our PLDS algorithms achieve greater self-relative speedups than Hua. Also, with just 4 threads (available on a standard laptop), PLDSOpt already outperforms all other algorithms. Hua's algorithm performs DFS/BFS, which could lead to linear depth, potentially explaining the bottleneck to their scalability with more cores.

Gabert et al. [38] present a parallel batch-dynamic k-core decomposition algorithm but their code is proprietary. However, their algorithm appears slower and less scalable based on their paper's stated results. For example, their algorithm on 10^5 edges using 32 threads for the *livejournal* graph requires 4 seconds, while our algorithm on a batch of 10^6 edges using 30 threads (more edges and fewer threads) requires a *maximum* of 0.35 seconds. Also, they appear to exhibit a maximum of 8× self-relative speedup on *livejournal* while we exhibit 21.2× self-relative speedup on *livejournal*.

6.5 Results on Large Graphs

Fig. 11 shows the runtimes of PLDSOpt, PLDS, Hua, Sun, and Zhang compared with the static algorithms ExactKCore and ApproxKCore on additional graphs, using **Ins**, **Del**, and **Mix** batches, all of size 10^6.

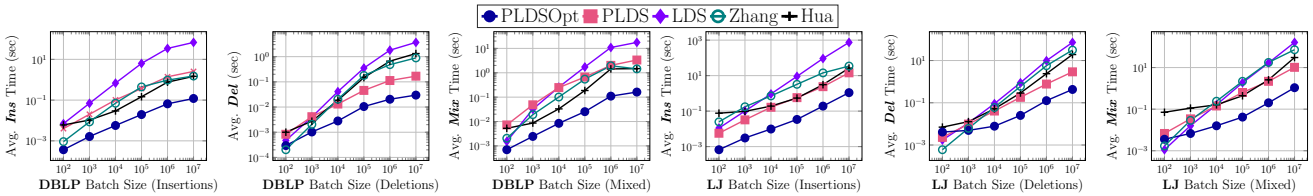


Figure 9: Average Ins, Del, and Mix per-batch running times on varying batch sizes for PLDSOpt, PLDS, LDS, Zhang, and Hua on *dblp* and *livejournal*.

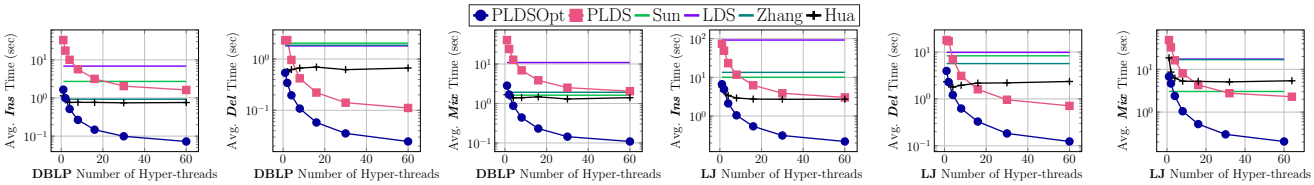


Figure 10: Parallel speedup of PLDSOpt, PLDS, and Hua, with respect to their single-threaded running times on *dblp* and *livejournal* on Ins, Del, and Mix batches of size 10^6 for all algorithms. The "60" on the x-axis indicates 30 cores with hyper-threading. LDS, Sun, and Zhang are shown as horizontal lines since they are sequential.

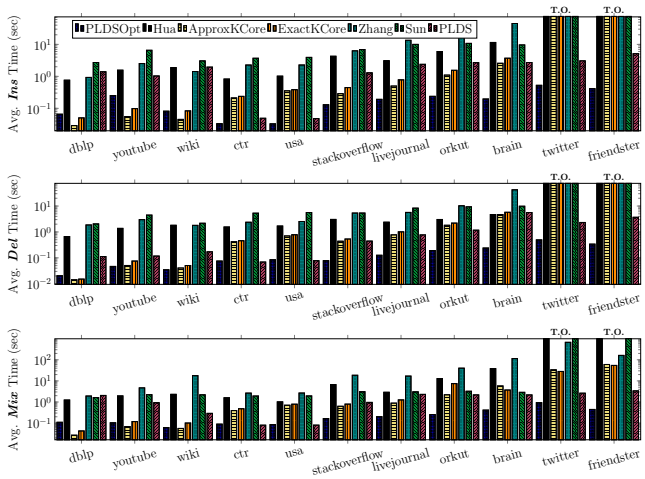


Figure 11: Average per-batch running times for PLDSOpt, Hua, PLDS, Sun, Zhang, ApproxKCore, and ExactKCore, on *dblp, youtube, wiki, ctr, usa, stackoverflow, livejournal, orkut, brain, twitter*, and *friendster* with batches of size 10^6 (and approximation settings $\delta = 0.4$ and $\lambda = 3$ for PLDSOpt and PLDS). All benchmarks (except PLDSOpt and PLDS) timed out (T.O.) at 3 hours for *twitter* and *friendster* for Ins and Del. Hua and Sun timed out on *twitter* and *friendster* for Mix. The top graph shows insertion-only, middle graph shows deletion-only, and bottom graph shows mixed batch runtimes.

ExactKCore and ApproxKCore are run from scratch over the entire graph after every batch since they do not handle batch updates. PLDSOpt and PLDS finished for all graphs and experiments while all other algorithms timed out on **Ins** and **Del** batches for *twitter* and *friendster*. Zhang was able to finish on **Mix** because their indexing algorithm (used to create their data structures provided the initial graph without the mixed batch) was able to finish; since only one mixed batch is used to update the graph, the sum of the time needed for indexing plus the update time of one batch fell under the timeout. The same is true for ExactKCore and ApproxKCore. However, these algorithms were not able to finish for **Ins** and **Del** because the sum of the update times across all batches is too high.

PLDSOpt is faster than all other dynamic algorithms on all types of batches, except for PLDS on *ctr* and *usa*. We report concrete speedups for experiments which finished within the timeout. For **Ins**, it gets 10.01–229.71× speedups over Zhang, 6.20–58.66× speedups over Hua, 26.02–119.77× speedups over Sun, and 1.45–23.89× speedups over PLDS. For **Del**, it gets 30–176.48× speedups over Zhang, 15.79–52.36× speedups over Hua, 41.02–100.34× speedups over Sun, and 2.51–23.45× speedups over PLDS (except on *ctr* and *usa*). For **Mix**, it gets 17.54–723.72× speedups over Zhang, 11.34–91.95× over Hua, 6.95–35.59× speedups over Sun, and 2.81–18.68× speedups over PLDS (except on *ctr* and *usa*). These massive speedups over previous work demonstrate the utility of PLDSOpt not only on large graphs but also on smaller graphs. Notably, our PLDSOpt and PLDS algorithms perform not only well on dense networks but also on very sparse road networks. For *ctr* and *usa*, PLDS performs better than PLDSOpt, achieving up to a 1.09× speedup on **Del** and 1.12× speedup on **Mix**.

Compared to the static algorithms, PLDSOpt achieves speedups for all but the smallest graphs, *dblp*, *wiki*, and *youtube*. For these graphs, the batch of size 10^6 accounts for more than 1/3 of the edges, and so even if the static algorithm reprocesses the entire graph per batch, it does not process many more edges past the batch size. Thus, it is expected that the parallel static algorithms perform better on small graphs and large batches. For all but the smallest graphs, PLDSOpt obtains 2.22–13.09×, 5.56–19.64×, and 4.4–121.76× speedups over the *fastest* static algorithm for each graph for **Ins**, **Del**, and **Mix**, respectively. ExactKCore and ApproxKCore both timeout for **Ins** and **Del** on *twitter* and *friendster*; otherwise, we expect to see the large improvements that we see for **Mix** on these experiments.

6.6 Accuracy of Approximation Algorithms

We also computed the average and maximum errors of all of our approximation algorithms for our experiments shown in Fig. 11. According to our theoretical proofs, the maximum error (for PLDS) should be $(2+3/3)(1+0.4) = 4.2$. We confirm that the maximum empirical error for PLDS falls under this constraint. PLDSOpt achieves an average error of 1.24–2.37 compared to errors of 1.26–3.48 for

PLDS, 1.01–4.17 for ApproxKCore, and 1.03–3.23 for Sun. PLDSOpt gets a maximum error of 3–6 compared to 2–4.19 for PLDS, 3–5 for ApproxKCore, and 3–5.99 for Sun. We conclude that our error bounds match those of the current best-known algorithms and are sufficiently small to be of use for many applications.

6.7 Sensitivity of PLDS and PLDSOpt to δ and λ

In Fig. 12, we provide a sensitivity analysis for the parameters δ and λ on the *maximum* error of our PLDS and PLDSOpt algorithms since our theoretical guarantees are for the *maximum error* and as we showed in Section 6.2, the average error does not vary significantly for our chosen set of parameters. The first three graphs of Fig. 12 shows the effect of fixing δ while varying λ and the last three show the opposite.

We see that for both PLDSOpt and PLDS, different λ values do not affect either the error by much (each line is essentially a cluster of points). This matches what we expect theoretically. Recall our bound on error, $(1+\delta)(2+3/\lambda)$; suppose we set $\delta = 0.4$ and $\lambda = 3$ as in our experiments. This leads to an upper bound of 4.2. If we increase λ to 6, this only decreases the error to 3.5. On the other hand, if δ is increased to 0.8, then the error increases to 5.4, resulting in greater sensitivity to δ.

However, *increasing* δ leads to a drastic decrease in running time (each line is a decreasing curve) at the expense of a large increase in error. Again, this matches what we expect theoretically, since δ affects the number of levels in PLDS and PLDSOpt (recall that in our algorithm, the number of levels per group is $\lceil \log_{(1+\delta)}(m) \rceil$). A larger number of levels leads to larger running time and we see this in our results. We do not see as large an increase for PLDSOpt since we divide the number of levels by 50. This means that for *livejournal* the number of levels per group is $\lceil \log_{(1+\delta)}(42851237)/50 \rceil = 1$ for all $\delta < 0.42$. We see this in our experiments as the curves for PLDSOpt are flat for $\delta \in [0.8, 6.4]$.

For the rest of the experiments, we fix $\delta = 0.4$ and $\lambda = 3$ based on our sensitivity analysis; these parameters offer a reasonable tradeoff between approximation error and speed, as shown in Fig. 8 and Fig. 12. For Sun, we choose the parameters $\varepsilon = \lambda = 2$ and $\alpha = 2$ since we observe these parameters give similar approximation errors to the parameters that we chose for our algorithms.

6.8 Space Usage

For each program, we implemented functions that measured the space usage of the data structures used in the algorithms (specifically, the private and public variables maintained in their data structure classes); for all of the algorithms, we do not count ephemeral space usage needed by auxiliary structures that are not maintained as either private or public variables of their data structure class. For this set of experiments, we only test on **Ins** and **Del** since maximum space is used when the entire graph is present in memory.

Fig. 13 shows the results of our space-bound experiments. Although PLDS uses more memory than most other implementations, our PLDSOpt uses less memory than Hua and Zhang in most settings (up to 1.34× factor less memory than the minimum space used by either) for *dblp* and up to 1.08x additional space in a few cases; for *livejournal*, it uses up to 1.72x additional space compared to the minimum space used by Hua and Zhang. Sun uses more space than PLDSOpt for most cases; although for a few parameters for deletions in *dblp*, it uses up to 1.9× less space. Since we have a $O(\log^2 n)$ factor in our space usage bound, we expect a slight increase in our space usage compared to algorithms with linear space; however, as we demonstrated, empirically our space usage is not much greater, and we believe that this small extra space usage is a small price to pay for the large improvement in performance obtained by our algorithms. We provide theoretical space-efficient implementations of our PLDS which may also prove to be more space-efficient in practice.

Algorithm 6 Static Approximate k-core Decomposition

Input: An undirected graph $G(V, E)$.
Output: An array of $(2+\varepsilon')$-approximate coreness values for any constant $\varepsilon' > 0$.

$\forall v \in V$, let $C[v] = |N(v)|$.
finished $\leftarrow 0$, $t \leftarrow 0$, $\varepsilon \leftarrow \frac{\sqrt{4\varepsilon'+9}-3}{2}$, $\delta \leftarrow \frac{2}{\varepsilon}$.
Let M be a bucketing structure formed by initially assigning each $v \in V$ to the $\lceil \log_{1+\varepsilon} C[v] \rceil$'th bucket.
while (*finished* $< |V|$) **do**
  $(I, bkt) \leftarrow$ Vertex IDs and bucket ID of next (peeled) bucket in M.
  $t \leftarrow bkt$.
  for iteration $j \in [\lceil \log_{1+\delta}(n) \rceil]$ **do**
    $R \leftarrow \{(v, r_v) \mid v \in N(I), r_v = |\{(u, v) \in E \mid u \in I\}|\}$.
    $U \leftarrow$ Array of length $|R|$.
    parfor $R[i] = (v, r_v)$, $i \in [0, |R|)$ **do**
      $inducedDeg = C[v] - r_v$
      $C[v] = \max(inducedDeg, \lceil (1+\varepsilon)^{t-1} \rceil)$
      $newbkt = \max(\lceil \log_{1+\varepsilon} C[v] \rceil, t)$
      $U[i] = (v, newbkt)$
    Update M for each $(u, newbkt)$ in U.
    $next\text{-}bkt \leftarrow$ bucket ID of the next smallest bucket in M.
    if $(1+\varepsilon)^{next\text{-}bkt} \leq (2+\varepsilon)(1+\varepsilon)^t$ **then**
      $(I, next\text{-}bkt) \leftarrow$ Vertex IDs of the next (peeled) bucket in M.
    else
      break
return C.

7 STATIC $(2+\varepsilon)$-APPROXIMATE k-CORE

Due to the P-completeness of k-core decomposition for $k \geq 3$ [5], all known static exact k-core algorithms do not achieve polylogarithmic depth. We introduce a linear work and polylogarithmic depth $(2+\varepsilon')$-approximate k-core decomposition algorithm (with only one-sided error) based on the parallel bucketing-based peeling algorithm for static *exact* k-core decomposition of Dhulipala et al. [27]. The algorithm maintains a mapping M from $v \in V$ to a set of *buckets*, with the bucket for a vertex $M(v)$ changing over the course of the algorithm. The algorithm starts at $k = 0$, peels all vertices with degree at most $(2+\varepsilon)(1+\varepsilon)^k$ where ε is set to $\frac{\sqrt{4\varepsilon'+9}-3}{2}$, increments k, and repeats until the graph becomes empty. The approximate core value of v is $(1+\varepsilon)^{k-1}$ where we use the value of k when v is peeled. We observe that the dynamic algorithm in this paper can be combined with a peeling algorithm like the above to yield a linear-work approximate k-core algorithm with polylogarithmic depth.

Algorithm 6 shows pseudocode for our approximate k-core algorithm, which computes an approximate coreness value for each vertex. The algorithm sets the initial coreness estimates, $C[v]$, of each

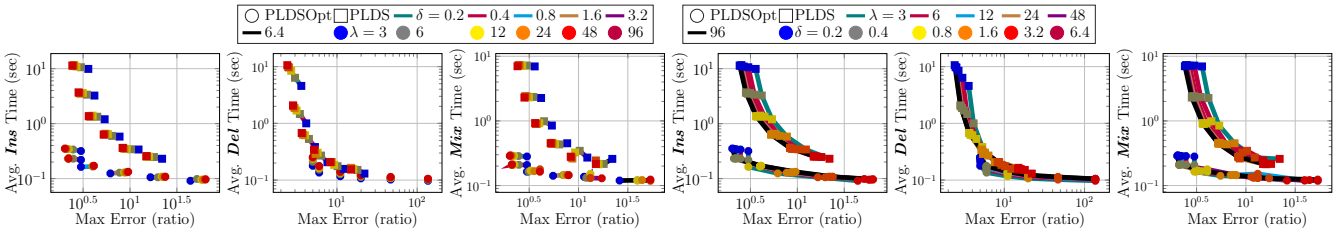


Figure 12: Sensitivity analysis of PLDSOpt and PLDS on *livejournal*. The first three plots fix δ; each line is a fixed δ value and each point is a different λ value. The last three plots fix λ and vary δ.

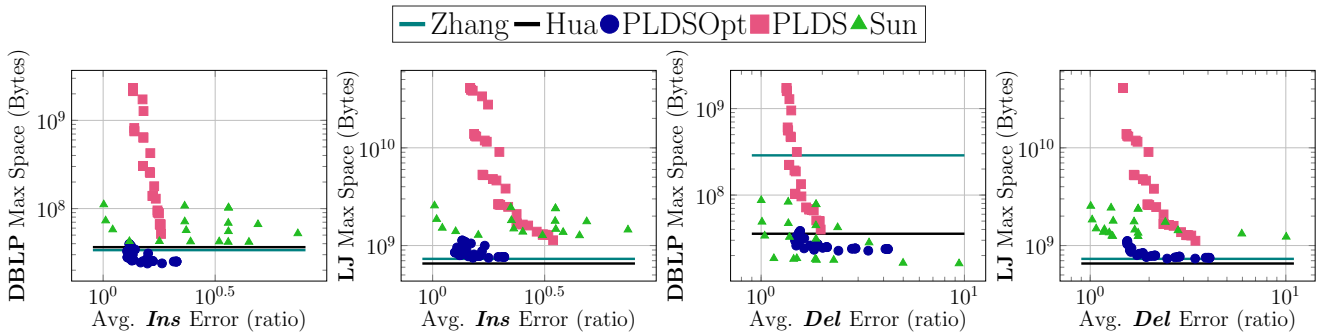


Figure 13: Maximum space usage in bytes for PLDSOpt, Hua, Zhang, PLDS, and Sun in terms of the average error. We varied δ and λ and computed the error ratio and space usage for the programs on *dblp* and *livejournal*. We tested against Ins and Del batches of size 10^5 for *dblp* and batches of size 10^6 for *livejournal*.

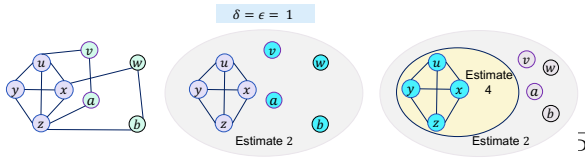
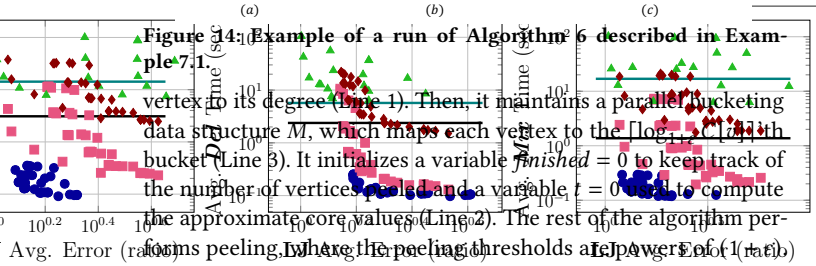


Figure 14: Example of a run of Algorithm 6 described in Example 7.1.



vertex to its degree (Line 1). Then, it maintains a parallel bucketing data structure M, which maps each vertex to the $\lceil \log_{1+\delta}(c[v]) \rceil$-th bucket (Line 3). It initializes a variable *finished* = 0 to keep track of the number of vertices peeled and a variable $t = 0$ used to compute the approximate core values (Line 2). The rest of the algorithm performs peeling, where the peeling thresholds are powers of $(1+\varepsilon)$. The peeling loop (Line 4–Line 20) first extracts the lowest non-empty bucket from M (Line 18), which consists of I, a set of vertex IDs of vertices that are being peeled, and the bucket number *bkt*. If more than $\log_{1+\delta}(n)$ rounds of peeling have occurred at the threshold $(2+\varepsilon)(1+\varepsilon)^t$ (where we set $\delta = \frac{2}{\varepsilon}$), the algorithm increments t (Line 6). Next, the algorithm computes in parallel an array R of pairs (v, r_v), where v is a neighbor of some vertex in I and r_v is the number of neighbors of v in I (Line 8). Finally, the algorithm computes in parallel the new buckets for the affected neighbors v (Line 10–Line 14). The coreness estimate is updated to the maximum of the peeling threshold of the previous level and the current induced degree of v after r_v of its neighbors are removed. Finally, the algorithm updates the buckets using the new coreness estimates for the updated vertices (Line 15), which can be done in parallel using our bucketing data structure.

We provide an example of this algorithm below.

Example 7.1. Fig. 14 shows a run of Algorithm 6 on an example graph. Given the parameters $\varepsilon = \delta = 1$, the two buckets that the vertices of the input graph (shown in (a)) are partitioned into are bucket index 1 (green vertices) and bucket index 2 (purple vertices). Vertices v, w, a and b have degree 2 so they are put into the bucket with index $\lceil \log_2(2) \rceil = 1$. Since u, x, y, and z have degree ≥ 3, they are put into the bucket with index $\lceil \log_2(3) \rceil = 2$.

Since the bucket with index 1 has the smaller bucket index, we peel off all the vertices in that bucket (the green vertices) and we assign the core estimate of $(1+\varepsilon)^1 = 2$ to all vertices in that bucket (shown in (b)). We update the buckets of all neighbors of the peeled vertices; however, since u, x, y, and z all still have degree ≥ 3, they remain in the bucket with index 2. Finally, we peel bucket index 2 and assign all vertices in that bucket an estimate of $(1+\varepsilon)^2 = 4$ (shown in (c)). In this example, the estimates produced are 3-approximations of the real coreness values.

We prove below that Algorithm 6 finds an $(2+\varepsilon)$-approximate k-core decomposition in $O(m)$ expected work and $O(\log^3 m)$ depth w.h.p., using $O(m)$ space, as stated in Theorem 3.8. We give the approximation guarantees of our algorithm using lemmas from [40], and use an efficient parallel semisort implementation [43] for our work bounds.

Theorem 7.2. *For a graph with m edges,*[10] *for any constant $\varepsilon > 0$, there is an algorithm that finds an $(2+\varepsilon)$-approximate k-core decomposition in $O(m)$ expected work and $O(\log^3 m)$ depth with high probability, using $O(m)$ space.*

Proof. Our approximation guarantee is given by Observation 4 of [40]. Using Observation 4, the number of vertices with core

[10] Our bounds in this paper assume $m = \Omega(n)$ for simplicity, although our algorithms work even if $m = o(n)$.

number $(1+\varepsilon)^t$ after one round of peeling in Algorithm 6 shrinks by a factor of $\frac{2}{2+\varepsilon}$. Let $V_{\leq(1+\varepsilon)^t}$ be the number of vertices with core number at most $(1+\varepsilon)^t$. After removing all vertices with degree at most $(2+\varepsilon)(1+\varepsilon)^t$, the number of vertices with core number $(1+\varepsilon)^t$ and with degree greater than $(2+\varepsilon)(1+\varepsilon)^t$ is at most $\frac{2(1+\varepsilon)^t|V_{\leq(1+\varepsilon)^t}|}{(2+\varepsilon)(1+\varepsilon)^t} = \frac{2|V_{\leq(1+\varepsilon)^t}|}{2+\varepsilon}$. Since $|V_{\leq(1+\varepsilon)^t}| \leq n$, the maximum number of rounds needed to peel all vertices with core number at most $(1+\varepsilon)^t$ is $\log_{(2+\varepsilon)/2}(n)$. By induction on t (Line 6), after $\log_{(2+\varepsilon)/2}(n)$ rounds, all vertices with core number at most $(1+\varepsilon)^t$ are removed. Hence, in round $t+1$, all vertices have core number greater than $(1+\varepsilon)^t$ and have core number at most $(2+\varepsilon)(1+\varepsilon)^{t+1}$; hence, we obtain a $(2+\varepsilon)(1+\varepsilon) = 2+\varepsilon'$ approximation (for any constant $\varepsilon' > 0$ and appropriate setting of ε) when we give coreness approximations of $(1+\varepsilon)^t$ to all vertices peeled for $t+1$.

Our algorithm uses a number of data structures that we use to obtain our work, depth, and space bounds. Our parallel bucketing data structure (Line 3) can be maintained via a sparse set (hash map), or by using the bucketing data structure from [27]. The outer loop iterates for $O(\log n)$ times (Line 4). Within each iteration of the outer loop, we iterate for $O(\log_{(1+\delta)} n) = O(\log n)$ rounds for constant $\delta = \frac{2}{\varepsilon}$. After obtaining a set of vertices, we update the buckets using semisort in $O(\log n)$ depth w.h.p. [27]. Thus the overall depth of the algorithm is $O(\log^3 m)$ for any constant $\delta > 0$.

The work of the algorithm can be bounded as follows. We charge the work for moving a vertex from its current bucket to a lower bucket within a given round to one of the edges that was peeled from the vertex in the round. Thus the total number of bucket moves done by the algorithm is $O(m)$. Each round of the algorithm also peels a number of edges and aggregates, for each vertex that has a neighbor in the current bucket, the number of edges incident to this vertex that are peeled (the r_v variable in the algorithm). We implement this step using a randomized semisort [43]. Since $2m$ edges are peeled in total, the overall work is $O(m)$ in expectation.

Lastly, we bound the space used by the algorithm. There are a total of $O(\log_{1+\varepsilon} n) = O(\log n)$ buckets for any constant $\varepsilon > 0$. Each vertex appears in exactly one bucket, and thus the overall space of the bucketing structure is $O(n)$. The algorithm also semisorts the edges peeled from the graph in each step. Since all m edges could be peeled and removed within a single step, and thus semisorted the overall space used by the algorithm is $O(m)$. □

The approximation guarantees provided by our algorithm are essentially the best possible, under widely believed conjectures. Specifically, Anderson and Mayr [5] show that the optimization version of the High-Degree Subgraph problem, namely to compute the largest core number, or *degeneracy* of a graph cannot be done better than a factor of 2. Thus, obtaining a polynomial work and polylogarithmic depth $(2-\varepsilon)$-approximation to the coreness value of each vertex would yield a $(2-\varepsilon)$-approximation to the optimization version of the High-Degree Subgraph problem, and show that P = NC, contradicting a widely-believed conjecture in parallel complexity theory.

In recent years, several results have given parallel algorithms that obtain a $(1+\varepsilon)$-approximation to the coreness values in distributed models of computation such as the Massively Parallel Computation model [33, 40]. These results work by performing a *random sparsification* of the graph into a subgraph that approximately preserves the coreness values. They then send this subgraph to a single machine, which runs the sequential peeling algorithm on the subgraph to find approximate coreness values. Crucially, this second peeling step on a single machine can have $\Theta(n)$ depth, and thus, this approach does not yield a polylogarithmic depth algorithm in the work-depth model of computation.

Algorithm 7 GraphProblemUpdate($G, \mathcal{B}$)

Input: A graph $G = (V, E)$ and a batch $\mathcal{B}$ of unique and valid updates.
Output: A solution to the relevant graph problem.
1: Update($\mathcal{B}$) [Algorithm 1].
2: $A \leftarrow$ LowOutdegreeOrient($\mathcal{B}$).
3: Perform parallel filter on $\mathcal{B}$ to obtain a batch of insertions, $\mathcal{B}_{ins}$, and a batch of deletions, $\mathcal{B}_{del}$.
4: BatchFlips(A, $\mathcal{B}_{ins}$, $\mathcal{B}_{del}$).
5: BatchDelete($\mathcal{B}_{del}$).
6: BatchInsert($\mathcal{B}_{ins}$).

8 FRAMEWORK FOR BATCH-DYNAMIC GRAPH ALGORITHMS FROM LOW OUT-DEGREE ORIENTATIONS

In this section, we introduce a framework that we will use in all of our batch-dynamic algorithms that use our batch-dynamic low out-degree orientation algorithm (Section 5.6). Our framework assumes three different methods for each of the problems (maximal matching, k-clique counting, and vertex coloring) that we solve. Specifically, these three methods handle batches of insertions and deletions separately; let BatchFlips, BatchInsert, and BatchDelete denote these three methods.

We assume for simplicity that all updates in the batch $\mathcal{B}$ are *unique*, which means that no edge deletion occurs on an inserted edge in the batch and vice versa. Furthermore, we assume that the updates are *valid*, meaning that if an edge insertion (u, v) is in $\mathcal{B}$, then (u, v) does not exist in the graph, and if an edge deletion (w, x) is in $\mathcal{B}$, then edge (w, x) exists in the graph. Such assumptions are only *simplifying* assumptions because it is easy to perform preprocessing on $\mathcal{B}$ in $O(|\mathcal{B}| \log n)$ work and $O(\log n)$ depth to ensure that these assumptions are satisfied. In fact, our implementations in Section 6 do perform this preprocessing on the input batches. To find all unique updates, we perform a parallel sort in $O(|\mathcal{B}| \log n)$ work and $O(\log n)$ depth [17, 28, 50]; we first sort on the edge and then the timestamp of the update. Then, we perform a parallel filter in $O(|\mathcal{B}|)$ work and $O(1)$ depth [17, 28, 50] where we keep each edge with the latest timestamp. Then, we perform another parallel filter to keep only edge insertions of nonexistent edges and edge deletions of edges that exist in the graph. This preprocessing ensures $\mathcal{B}$ follows our simplifying assumptions and do not exceed the complexity bounds of our PLDS, and hence, we assume all input batches contain unique and valid updates. The work and depth for preprocessing are subsumed by the bounds for the algorithms.

Detailed Framework. The pseudocode for our framework is shown in Algorithm 7. We first update the PLDS by calling the update procedure (Algorithm 1) on the batch of updates in Line 1. Afterwards, we call our low out-degree orientation algorithm to obtain the set of edges that were flipped, placed in set A (Line 2). Then, we take the batch of updates $\mathcal{B}$ and split the batch

into a batch of insertions, $\mathcal{B}_{ins}$, and a batch of deletions, $\mathcal{B}_{del}$ (Line 3). We call BatchFlips (Line 4) on the set of flipped edges A, which processes the edge flips accordingly for each problem. Finally, we call the problem specific functions BatchDelete and BatchInsert (Lines 5 and 6) on $\mathcal{B}_{del}$ and $\mathcal{B}_{ins}$, respectively; we first call BatchDelete and then BatchInsert.

Analysis. By Corollary 3.3, our low out-degree orientation algorithm gives a $O(\alpha)$ out-degree orientation. Furthermore, the amortized work of the algorithm indicates that $O(|\mathcal{B}|\log^2 n)$ amortized flips occur with each batch $\mathcal{B}$. Suppose that BatchFlips(A) takes $O(|A|W_{flips}(\alpha))$ work and $O(D_{flips})$ depth; BatchInsert($\mathcal{B}_{ins}$) takes $O(|\mathcal{B}_{ins}|W_{ins}(\alpha))$ work and $O(D_{ins})$ depth, and BatchDelete($\mathcal{B}_{del}$) takes $O(|\mathcal{B}_{del}|W_{del}(\alpha))$ work and $O(D_{del})$ depth; and the update methods require $O(S)$ space in total. Then, we show the following theorem about our framework.

Theorem 8.1. *Algorithm 7 takes*

$$O\left(|\mathcal{B}|W_{flips}(\alpha)\log^2 n + |\mathcal{B}|W_{ins}(\alpha) + |\mathcal{B}|W_{del}(\alpha)\right)$$

amortized work and

$$O\left(\log^2 n \log\log n + D_{flips} + D_{ins} + D_{del}\right)$$

depth w.h.p., in $O(n\log^2 n + m + S)$ space.

Proof. Theorem 3.2 states that updating the PLDS and getting the flipped edges require $O(|\mathcal{B}|\log^2 n)$ amortized work, $O(\log^2 n \log\log n)$ depth, and $O(n\log^2 n + m)$ space. Since the calls to the procedures are independent and sequential, the total work, depth and space equal the sum of the work, depth, and space of our PLDS algorithm and BatchFlips, BatchDelete and BatchInsert.

Then, the only additional information we need are the sizes of $\mathcal{B}_{ins}$ and $\mathcal{B}_{del}$. By our algorithm, $|\mathcal{B}_{ins}|, |\mathcal{B}_{del}| \leq |\mathcal{B}|$ since $\mathcal{B}_{ins} \cup \mathcal{B}_{del} = \mathcal{B}$. By Theorem 3.2, A has $O(|\mathcal{B}|\log^2 n)$ amortized flips; thus, the amortized work of BatchFlips is $O(|\mathcal{B}|W_{flips}(\alpha)\log^2 n)$. Finally, the PLDS uses $O(n\log^2 n + m)$ space; thus, with the additional $O(S)$ space, the total space used is $O(n\log^2 n + m + S)$. □

In addition, we assume that the algorithms BatchInsert and BatchDelete correctly maintain the desired properties required by each specific problem after processing $\mathcal{B}_{ins}$ and $\mathcal{B}_{del}$, respectively. Such an assumption ensures the correctness of the solutions produced by our framework. We show in Sections 9 to 11 that this is true for all of our procedures. Additionally, we can get rid of the $O(n\log^2 n)$ term in space at the expense of an extra $O(\log^2 n)$ factor in depth by using our space-efficient structures from Section 5.8.

Using this framework (with the PLDS guarantees given in Theorem 3.1), we present batch-dynamic algorithms for a number of problems in Sections 9 to 11 for maximal matching, k-clique counting, and vertex coloring.

9 MAXIMAL MATCHING

A maximal matching in a graph $G = (V, E)$ is a set of edges M in the graph such that no vertex is adjacent to two edges in M. Furthermore, no additional edges can be added to M without causing a vertex to be adjacent to two edges in M.

We provide the following parallel batch-dynamic algorithm for maximal matching using our framework given in Section 8. We instantiate BatchInsert and BatchDelete for the maximal matching problem in this section. We use the simple algorithm of Neiman and Solomon [75] as a starting point, although we will see that the batch-dynamic setting introduces several non-trivial challenges.

Sequential Algorithm of Neiman and Solomon [75]. The sequential algorithm of Neiman and Solomon [75] uses the dynamic orientation algorithm of Brodal and Fagerberg [19], which gives an $O(D)$ out-degree orientation for any $D > 2\alpha_{max}$. Given an edge insertion, they check whether both endpoints are in the maximal matching. If not, they match the endpoints to each other. For each vertex $u \in V$, they maintain the set of unmatched in-neighbors, $F(u)$, in a data structure consisting of an array augmented with a linked list. On an edge deletion (u, v) where (u, v) is in the matching, they check $F(u)$ (resp. $F(v)$) to see if any in-neighbors, u' (resp. v') are unmatched. If so, they match u to u' (resp. v to v'). If no in-neighbors are unmatched, they check whether any of their out-neighbors, u'' (resp. v'') are unmatched. If so, u (resp. v) matches with u'' (resp. v''). On an edge deletion (u, v) (where the edge is oriented from u to v), if u is unmatched, it removes itself from $F(v)$. On an edge insertion, (u, v), if u is unmatched, it adds itself to $F(v)$, and if u is matched we do not do anything. Finally, for an edge flip from (u, v) to (v, u), if u is unmatched, it removes itself from $F(v)$; if v is free, it adds itself to $F(u)$. Again if u is matched, we do not do anything. Maintaining the maximal matching and updating all data structures can be done in $O\left(\frac{\log n}{\log((\log n)/\alpha_{max})} + \alpha_{max}\right)$ amortized time for $\alpha_{max} = o(\log n)$. For $\alpha_{max} = \Omega(\log n)$, they obtain $O(\alpha_{max})$ amortized time.

Unfortunately, the batch-dynamic setting introduces several challenges, the most important of which is: edge deletions may unmatch many different vertices simultaneously, which need to be matched to potentially the same set of in-neighbors. Thus, we can no longer arbitrarily pick in-neighbors to match unmatched vertices since many vertices may be matched to the same in-neighbor. But we also cannot afford to look at *all* of the in-neighbors of an unmatched vertex since the in-degree is potentially $\omega(\alpha)$. Even for edge insertions, we cannot choose to add every edge insertion between two unmatched vertices to the maximal matching since many edge insertions may occur on the *same* unmatched vertex.

Batch-Dynamic Algorithm. Edge insertions are easier to handle; for each edge insertion, in parallel, we check whether both endpoints adjacent to the insertion are unmatched. If so, we run a static, parallel algorithm over all such vertices adjacent to an edge insertion but is unmatched; this finds a maximal matching among all vertices that want to be matched due to edge insertions. If not, we do nothing for these vertices.

Deletions are trickier to handle. For each vertex incident to an edge deletion, we check whether it is still matched or if it can be matched with any of its neighbors. However, such an operation could be expensive because although a vertex has bounded number of out-neighbors, it may have many in-neighbors. To find a new matching for unmatched vertices due to edge deletions, we make

Algorithm 8 MaximalMatchingBatchFlips($A, \mathcal{B}_{ins}, \mathcal{B}_{del}$)

Input: A set of edge flips A.
Output: Updated data structures.
1: **parfor** each flipped edge $(u, v) \in A$ **do** ▷ The edge is flipped from (u, v) to (v, u) and stored as (u, v) in A.
2: **if** (u, v) is in the matching **then**
3: Remove u from I_v.
4: Add v to I_u.

use of the best-known low-depth, parallel, static maximal matching algorithm which takes $O(m + n)$ work[11] and $O(\log^2 n)$ depth w.h.p. [14, 16, 36] combined with a scheme where we progressively double the number of in-neighbors we attempt to match. Details about these procedures are provided in the next subsections.

Data Structure. We maintain the following data structures in our algorithm. For each vertex v, we maintain a parallel hash table, I_v, of in-neighbors which are unmatched. Each time a vertex v becomes unmatched, we inform all out-neighbors of v that v is unmatched. Similarly, when v becomes matched, we inform all out-neighbors that it is matched. Then, each vertex that has been informed that v has been unmatched adds v to its hash table of unmatched incoming neighbors, in parallel. We assume that the out-neighbors of every vertex u are also maintained in a parallel hash table X_u, that is kept up to date by the edge orientation algorithm. These data structures require $O(m)$ in total space usage. Sequential versions of I_v and X_u are maintained by Neiman and Solomon [75].

9.1 Maximal Matching BatchFlips

The pseudocode for this procedure is given in Algorithm 8. To implement BatchFlips for maximal matching, we update the data structures I_w to accurately account for unmatched in-neighbors of vertices (which are stored in the I_v structures for each vertex v). To do this in parallel, for each flipped edge from (u, v) to (v, u) (Line 1), we remove u from I_v (Line 3) and add v to I_u (Line 4).

9.2 Maximal Matching BatchInsert

The pseudocode for this procedure is given in Algorithm 9. To implement BatchInsert for maximal matching, we need to check, in parallel, whether *both* endpoints of the inserted edge are unmatched (Line 3). If so, we know that they can potentially be matched to each other. However, there could be multiple edge insertions incident to the same unmatched vertex; thus, we cannot simply add every inserted edge between unmatched vertices to the maximal matching. Instead, we keep track of all edge insertions between two unmatched vertices in a dynamic array S (Line 1) and run a static, parallel maximal matching algorithm on the *induced subgraph* given by S (Line 5). We specifically use the work-efficient parallel, static maximal matching algorithm of Blelloch et al. [16] which was shown to have a better depth than previously stated in the analysis provided by Fischer and Noever [35]. Finally, each newly matched vertex from Line 5 updates its out-neighbors that it is now matched. For each such newly matched vertex v, each out-neighbors w of v removes v from I_w.

[11] The work of the parallel static matching algorithm given in [16] can be shown to be $O(m + n)$ w.h.p. when using the high probability analysis of parallel bucket sort given by Bercea and Even [9].

Algorithm 9 MaximalMatchingBatchInsert($\mathcal{B}_{ins}$)

Input: A batch $\mathcal{B}_{ins}$ of unique and valid insertion updates.
Output: A maximal matching.
1: $S \leftarrow \emptyset$. ▷ S contains matching candidate edges.
2: **parfor** each edge $\{u, v\} \in \mathcal{B}_{ins}$ **do**
3: **if** u **and** v are unmatched **then**
4: $S \leftarrow S \cup \{\{u, v\}\}$.
5: Run StaticMaximalMatching($G(S)$).
6: **parfor** each newly matched vertex v **do**
7: **parfor** each out-neighbor w of v **do**
8: Remove v from I_w.

Algorithm 10 MaximalMatchingBatchDelete($\mathcal{B}_{del}$)

Input: A batch $\mathcal{B}_{del}$ of unique and valid deletion updates.
Output: A maximal matching.
1: $U \leftarrow \emptyset$. ▷ U contains newly unmatched vertices.
2: $T \leftarrow \emptyset$. ▷ Contains the out-neighbors of unmatched vertices.
3: **parfor** each edge $\{u, v\} \in \mathcal{B}_{del}$ **do**
4: **if** $\{u, v\}$ is in the matching **then**
5: $U \leftarrow U \cup \{u, v\}$.
6: $T \leftarrow T \cup X_u \cup X_v$.
7: Run StaticMaximalMatching($G(U \cup T)$).
8: **parfor** each newly matched vertex v in $G(U \cup T)$ **do**
9: **parfor** each out-neighbor w of v **do**
10: Remove v from I_w.
11: $U \leftarrow U \setminus \{v\}$.
12: $c \leftarrow 1$. ▷ c is the number of incoming unmatched neighbors picked to run the static maximal matching algorithm.
13: **while** $U \neq \emptyset$ **do**
14: **parfor** each vertex $u \in U$ **do**
15: Pick c incoming unmatched neighbors arbitrarily.
16: **if** $I_u = \emptyset$ **then**
17: $U \leftarrow U \setminus \{u\}$.
18: Let G' be the induced subgraph consisting of all vertices in U and the picked incoming unmatched neighbors.
19: Run StaticMaximalMatching(G').
20: **parfor** each newly matched vertex v in G' **do**
21: **parfor** each out-neighbor w of v **do**
22: Remove v from I_w.
23: $U \leftarrow U \setminus \{v\}$.
24: Set $c \leftarrow 2 \cdot c$.
25: **parfor** each $v \in U$ **do**
26: **if** v remains unmatched **then**
27: **parfor** each out-neighbor w of v **do**
28: Add v to I_w.

The correctness of our procedure follows from the fact that only new edge insertions may be added to the matching. Because our algorithm always maintains a maximal matching, any previous edge that existed in the graph is either in the matching or is incident to a matched vertex. Thus, our procedure only needs to consider newly inserted edges and such edges can be determined using a parallel, static maximal matching algorithm [16].

9.3 Maximal Matching BatchDelete

The pseudocode for this algorithm is given in Algorithm 10. For any edge (u, v) that is part of the matching that has been removed by an edge deletion, we create an induced subgraph consisting of the set of such vertices and their out-neighbors (Lines 1, 2, 5 and 6). Given $|\mathcal{B}_{del}|$ such deletion updates, the induced subgraph of each vertex v

affected by the deletions and its out-neighbors has size $O(|\mathcal{B}_{del}|\alpha)$. We use the parallel, static algorithm of Blelloch et al. [16] to find a matching in this induced graph (Line 7).

For vertices that remain unmatched after the above procedure is run, we must now attempt to match these vertices with the set of incoming unmatched neighbors. To do this, we run the parallel, static maximal matching algorithm on some induced subgraphs of the remaining unmatched vertices and a subset of incoming vertices. Specifically, starting from $c = 1$ (Line 12), each vertex remaining in U queries exactly c of its in-neighbors (the in-neighbors can be chosen arbitrarily) (Line 15). Suppose that G' is the induced subgraph consisting of all vertices in U and the picked incoming unmatched neighbors of the vertices in U. We run [16] on G' to obtain matchings (Line 19). The matched vertices consists of vertices in U and (some) of their in-neighbors. For each newly matched vertex v, we remove it from the I_w of each of its out-neighbors w (Line 22). Then, for each vertex in U that becomes matched, we remove it from U (Line 23). We double c and proceed with this entire process again if there remains unmatched vertices u in U (Line 13) *and* I_u is not empty (Line 17).

The correctness of Algorithm 10 follows immediately from our procedures. Our algorithm always maintains a maximal matching after processing a batch of updates. A vertex becomes unmatched (if it was previously matched) if and only if it is incident to an edge deletion and the edge deletion deletes a matched edge. An unmatched vertex can be matched to one of its in-neighbors or out-neighbors. We check both sets of neighbors in our procedure in order to match all unmatched vertices adjacent to edge updates.

9.4 Work and Depth Analysis

Here we show the work and depth analysis of our maximal matching algorithms (Algorithms 8 to 10).

Lemma 9.1. *The depth of Algorithms 8 to 10 is* $O(\log^2 n(\log\Delta + \log\log n))$ *w.h.p.*

Proof. We first prove the depth of each algorithm separately and use Theorem 8.1 to find the total depth.

In Algorithm 8, we can process all flipped edges in parallel (Line 1). Adding and removing vertices from the hash tables I_v requires $O(\log^* n)$ depth w.h.p. to perform in parallel (Lines 3 and 4).

In Algorithm 9, finding all edges in $\mathcal{B}_{ins}$ that are between two unmatched vertices can be done in parallel in $O(1)$ depth (Lines 2 to 4). Then, by the analysis in [16, 35], the parallel, static algorithm we use in Line 5 runs in $O(\log^2 n)$ depth w.h.p. Finally, updating the I_w of each out-neighbor w of a newly matched vertex v can be done in parallel in $O(\log^* n)$ depth w.h.p. (Lines 6 to 8). Thus, Algorithm 9 can be done in $O(\log^2 n)$ depth w.h.p.

In Algorithm 10, finding all newly unmatched vertices and making the induced subgraph consisting of the these vertices and their out-neighbors can be done in $O(\log n)$ depth (Lines 3 to 4 and 6) using a parallel filter. As before, running the parallel, static algorithm takes $O(\log^2 n)$ depth w.h.p. (Line 7). Then, removing each newly matched vertex v from the I_w of each out-neighbor w of v takes $O(\log^* n)$ depth (Lines 8 to 10). Removing the matched vertices v from U can also be done in parallel in $O(\log^* n)$ depth (Line 11) if U is maintained as a parallel hash table. The depth of the outer while loop (Line 13) is $O(\log\Delta)$ since the while loop iterates to a value of c that is at most $c = O(\Delta)$. When $c = \Delta$, all incoming neighbors of every vertex would be included in the induced subgraph G' and, hence, a maximal matching is guaranteed in this final case. Because the value of c is doubled each time, the total number of iterations of the while loop is $O(\log\Delta)$. The depth of the static matching procedure is $O(\log^2 n)$ w.h.p., and so the total depth of Lines 12 to 24 is $O(\log\Delta\log^2 n)$ w.h.p. Finally, the last step of adding each remaining unmatched vertex from I_w of each of its out-neighbors w takes $O(\log^* n)$ depth. Note that the matched vertices have already been removed from I_w in the previous lines (Lines 10 and 22). Thus, Algorithm 10 takes $O(\log\Delta\log^2 n)$ depth w.h.p.

By Theorem 8.1, the total depth of Algorithms 8 to 10 is $O(\log^2 n(\log\Delta + \log\log n))$ w.h.p. □

Lemma 9.2. *Algorithms 8 to 10 require* $O(|\mathcal{B}|(\alpha+\log^2 n))$ *amortized work w.h.p.*

Proof. As in the depth proof, we first prove the work of each of the individual algorithms and then use Theorem 8.1 to show the final work bound.

By Theorem 3.2, the number of edge flips is $O(|\mathcal{B}|\log^2 n)$ amortized. Thus, in Algorithm 8, the number of edge flips we process in total is $O(|\mathcal{B}|\log^2 n)$ (Line 1). For each edge flip, we spend $O(1)$ work to add and remove, respectively, from I_u and I_v (Lines 3 and 4). Then, the total work of Algorithm 8 is $O(|\mathcal{B}|\log^2 n)$ amortized.

In Algorithm 9, there are at most $|\mathcal{B}|$ insertions and checking whether the endpoints of the edges are unmatched requires $O(|\mathcal{B}|)$ work. This procedure produces at most $O(|\mathcal{B}|)$ unmatched vertices in S since each edge is incident to two vertices. Then, running the parallel, static work-efficient maximal matching algorithm of [16] on the induced subgraph of the unmatched vertices requires $O(|\mathcal{B}|)$ work w.h.p. Then, removing the matched vertices from the I_w of each out-neighbor requires $O(|\mathcal{B}|)$ work. Thus, Algorithm 2 takes $O(|\mathcal{B}|)$ work w.h.p..

The remainder of the proof focuses on proving the work for Algorithm 10. First, we note that each vertex that becomes matched is either a vertex in $u \in U$ (Line 4) or is an unmatched incoming neighbor of u. There can be at most $4|U|$ unmatched vertices that may become matched since each deletion update can unmatch at most two vertices. Thus, the work of informing all out-neighbors throughout the procedure is $O(|\mathcal{B}|\alpha)$ since by Theorem 3.2, we maintain an $O(\alpha)$ out-degree orientation and there are $O(|\mathcal{B}|)$ deletion updates. Running the static algorithm (Line 7) on the induced subgraph of U and its out-neighbors requires $O(|\mathcal{B}|\alpha)$ work w.h.p. Then, the remaining work comes from the work of looking at c incoming neighbors of each node $u \in U$ of the set of vertices U that have not been matched and have at least one unmatched neighbor. This requires $O(c|U|)$ work to find a maximal matching in the induced subgraph of U and c incoming vertices of each vertex in U. We perform the following charging argument to calculate the work over all attempted c values.

The key to the charging argument is that we charge the cost of *attempted* matchings of a vertex to *when it or its in-neighbors are matched.* More specifically, let a *query* be an instance when an in-neighbor of a vertex is chosen in Line 15 and the static algorithm is run on the induced subgraph of the selected in-neighbors and the vertices remaining in U. When an in-neighbor becomes matched,

we charge to it the cost of *each previous vertex that queried the matched in-neighbor*. Since each matched in-neighbor has at most $O(\alpha)$ out-degree, each such matched in-neighbor will be queried at most $O(\alpha)$ times. The static algorithm we run in Line 19 takes work that is linear in the size of the induced subgraph G'; thus, this is $O(1)$ amortized per vertex in U and its chosen in-neighbors. Furthermore, as we stated before, there can be at most $O(|U|)$ matched in-neighbors. Thus, the total charged cost to each matched in-neighbor is $O(\alpha|U|)$. We now need to account for the cost of the in-neighbors that were queried but not matched.

To bound the number of such vertices, for each vertex $v \in U$, consider the last run of the static algorithm where v remains unmatched after the run. During this previous run, all queried in-neighbors of v were matched to some vertices in U; if there exists an in-neighbors that is unmatched, it would have been matched to v. In the final run of the static algorithm for v, we query at most two times the number of in-neighbors queried in the previous round. Thus, the remaining unmatched, queried in-neighbors in the final run for v can be charged to the *previous run* where all queried in-neighbors were matched. This results in an additional cost of $O(1)$ that is charged to each matched in-neighbor.

In total, our charging argument shows that finding the maximal matching in each subgraph takes $O(\alpha|U|)$ work. $|U| = O(|\mathcal{B}|)$, and so the total work of Algorithm 10 is $O(\alpha|\mathcal{B}|)$.

By Theorem 8.1, the total work of our batch-dynamic maximal matching algorithm is $O(|\mathcal{B}|(\log^2 n + \alpha))$ amortized, w.h.p. □

Theorem 9.3. *Our maximal matching algorithm takes $O(|\mathcal{B}|(\alpha + \log^2 n))$ amortized work and $O(\log^2 n(\log\Delta + \log\log n))$ depth w.h.p., and uses $O(n\log^2 n + m)$ space.*

Proof. The work and depth follow from Lemma 9.1 and Lemma 9.2. The algorithm uses space equal to the space required by the k-core decomposition algorithm since it only stores the additional I_v data structures which in total takes $O(m)$ additional space. Thus, the space required by our algorithm is $O(n\log^2 n + m)$. □

10 CLIQUE COUNTING

A k-clique is a set of k vertices where edges exist between all pairs of vertices in the set. Specifically, using our framework (Algorithm 7) and our problem specific methods, we obtain a k-clique counting algorithm (for constant k) that runs in $O(\alpha^{k-2}|\mathcal{B}|\log^2 n)$ work and $O(\log^2 n)$ depth w.h.p., using $O(m\alpha^{k-2} + n\log n)$ space.

10.1 Algorithm Overview

Due to the complexity of our algorithm, we first provide some intuition behind the core ideas before we give the specific details. First, we make the simple observation that any clique in a directed acyclic graph has a vertex where all edges in the clique that are adjacent to the vertex are directed out from the vertex. For a particular clique C, we call this vertex the ***source*** of C.

Observation 10.1. *Provided a directed acyclic orientation of a graph $G = (V, E)$, for any clique $C \in G$, there exists a unique vertex $v \in C$ where all edges from v to all other vertices $w \in C$ are directed from v to w.*

Proof. First, it is easy to see that the source is unique. This is because for any two vertices u and v in the clique, the edge $\{u, v\}$ must be directed either in the (u, v) direction or in the (v, u) direction, one of which makes v (resp. u) no longer the source.

Then, a simple proof by contradiction shows that the source exists. Suppose for contradiction that all vertices in C have at least one out-neighbor and one in-neighbor. We start with vertex v. Suppose that v's out-neighbor is w and v's in-neighbor is u. By our assumption, w must have at least one out-neighbor, x. $x \neq u$, otherwise, there exists a 3-cycle in the graph. ($x \neq v$ also since we're only considering simple graphs.) By the same argument, x must have at least one out-neighbor, y. $y \notin \{x, u, v, w\}$, otherwise, by the same argument, there would exist a cycle and we only consider simple graphs. Making the same argument for the k-th unique out-neighbor, we require a $(k+1)$-st unique vertex in order to not create a cycle. This contradicts the fact that C is a k-clique. □

We begin our description with an explanation of how to find the newly created cliques resulting from edge insertions. Using Observation 10.1, we make the second observation that for any edge update (u, v), we can count the number of k-cliques (for constant k) incident to (u, v) *and* where u is the source vertex of the clique in $O(\alpha^{k-2})$ work and $O(1)$ depth, provided an $O(\alpha)$ acyclic low out-degree orientation. This is because u and v must be in the clique and, thus, there are $\binom{c\alpha}{k-2} = O(\alpha^{k-2})$ additional vertices to choose from among u's out-neighbors (for some constant c hidden in the $O(\alpha)$). This observation also means that we do not have to worry about finding a clique *until after all edges adjacent to its source vertices* are added. The clique will be found by the last of these source edges when it is added. Thus, the main challenge of our algorithm is how to find the cliques resulting from edge updates to other vertices *aside from those adjacent to the source vertex*.

This leads to our third and final observation: a k-clique can be formed from a $(k-1)$-clique by attaching a source vertex where all edges from the source vertex are directed into the vertices of the $(k-1)$-clique. The last crucial observation allows us to count k-cliques inductively by counting $(k-1)$-cliques, which are in turn counted using $(k-2)$-cliques, and so on. This means that for any k-clique C, by Observation 10.1, there exists a set of *unique* source vertices responsible for the set of smaller cliques within C. Specifically, for every clique $C_i \subseteq C$ of size $i \in [2, \ldots, k]$, there exists a source for this clique. For every edge insertion, we first determine the possible sets of k vertices which can be *completed* by future edge insertions to form k-cliques. Potential cliques are determined using the above observation by assuming for each edge insertion (u, v), u is the source of the clique. Suppose C is one such set. We assign the responsibility of counting the potential k-clique C to the *largest incomplete clique, $C_i \subset C$, without a source*. This can occur when C_i does not yet have enough edges to determine the source (see (1) in Fig. 15 where $\{a, b, d, e\}$ does not yet have a source). The base case, the smallest possible largest incomplete clique without a source, is an edge; once this edge is inserted, the source of the edge counts the clique. The concept of the largest incomplete clique without a source is fundamental to our algorithm.

Given a batch of edge insertions, if a set of edges completes the largest incomplete clique without a source, C_i, of C, then the new source of this clique is responsible for counting C in the clique count.

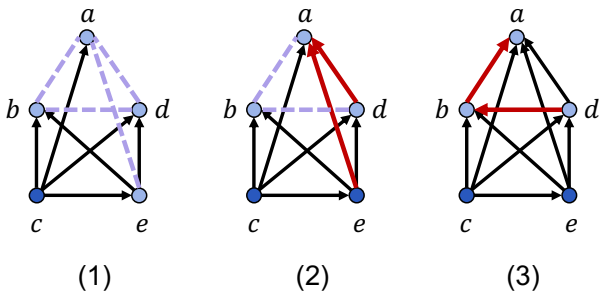


Figure 15: Example of incomplete cliques in our counting algorithm for counting $k = 5$ cliques. In (1), c is the source of a potential 5-clique. $\{a, b, c, d, e\}$ represents a potential 5-clique. We do not yet know the source of the 4-clique consisting of $\{a, b, d, e\}$ (the purple edges represent potential edges), and so we store $\{a, b, d, e\}$ in table I_4. (2) shows a set of edge insertions (indicated by the red edges) which determines a source (e) for the 4-clique. Thus, we insert $\{a, b, d\}$ in table I_3. Suppose that this is the only clique that would be counted when edges are inserted between all pairs in $\{a, b, d\}$. Thus, we associate with this key, a count of 1 in table I_3. Finally, (3) shows two edge insertions which completes the triangle; hence, we count the clique using the key $\{a, b, d\}$ and increment the k-clique count using the count associated with it (in this example, the count is 1) in table I_3.

Crucially, C cannot be counted until C_i is completed; furthermore, for any set of k vertices C with a source, but is not a clique, there exists a C_i that can count C. If the batch does not complete the clique but a source has been found for C_i, then, we determine the new largest incomplete clique without a source, C_j (where $j < i$), that will be responsible for counting C. In Fig. 15, (2) shows a set of insertions that determines that e is the source of $\{a, b, d, e\}$. Then, the new largest incomplete clique without a source is $\{a, b, d\}$.

This naturally leads to an algorithm for counting k-cliques. We create $k - 2$ parallel hash tables, where for each potential k-clique C, we store the indices of the vertices comprising the largest incomplete cliques, C_i, without a source, for $i \in [2, k - 1]$, in table I_i. The values stored in these hash tables are the numbers of k-cliques C that would be completed if C_i in table I_i is completed. Storing the indices of all vertices allows us to determine the source when the appropriate edges have been inserted. (More details are given in our detailed algorithm below.) Given this set of structures, we increase the k-clique count when a clique from table I_i is completed by a batch of insertions; to increase the k-clique count, we use the value stored for the clique. If there remains any incomplete cliques, we use each table I_j for $j > 2$ to update tables I_i for $i < j$ if any incomplete cliques in j have found sources. We give an example illustration of this part of the algorithm in Fig. 15.

Data Structures. We maintain the following data structures in our algorithm. We maintain $k-2$ parallel hash tables, I_i for $i \in [2, k-1]$. For each I_i, the keys are ordered sets of vertices of size i, and the values are counts. The counts represent the number of k-cliques that would form if all edges among the vertices in the keys exist. To prevent over-counting, one edge update incident to the new source of any newly completed clique stored in I_i is responsible for increasing the count by the stored value.

Algorithm 11 CliqueCountingBatchFlips($A, \mathcal{B}_{ins}, \mathcal{B}_{del}$)

Input: A set of edge flips A.
Output: Updates $\mathcal{B}_{ins}, \mathcal{B}_{del}$.
1: **parfor** each flipped edge $(u, v) \in A$ **do** ▷ The edge is flipped from (u, v) to (v, u) and stored as (u, v) in A.
2: $\mathcal{B}_{del} \leftarrow \mathcal{B}_{del} \cup (u, v)$.
3: $\mathcal{B}_{ins} \leftarrow \mathcal{B}_{ins} \cup (v, u)$.

Algorithm 12 CliqueCountingBatchInsert($\mathcal{B}_{ins}$)

Input: A batch $\mathcal{B}_{ins}$ of unique and valid insertion updates.
Output: An updated k-clique count and updated data structures.
1: Let count be the current count of the number of k-cliques in the graph.
2: Insert the edges in $\mathcal{B}_{ins}$ into the graph in the orientation specified by $\mathcal{B}_{ins}$. Mark all edges in $\mathcal{B}_{ins}$ in the graph.
3: Let R be the order of the edges in $\mathcal{B}_{ins}$.
4: **parfor** each edge $(u, v) \in \mathcal{B}_{ins}$ **do** ▷ The edge is oriented from u to v.
5: **for** $i \in [1, \ldots, k-2]$ **do**
6: **parfor** each subset T of i out-neighbors of u (excluding v) **do**
7: Let T' be the ordered set of $T \cup \{u, v\}$ sorted by vertex index.
8: **if** u is the source of T' **and** (u, v) is the earliest in R out of all marked edges from u to $w \in T'$ **then**
9: **if** all edges between each pair $x, y \in T'$ exists **then**
10: $j \leftarrow 2$.
11: **else**
12: Find largest incomplete clique without a source, C', in T'.
13: Let j be the size of C'.
14: $T_{sub} \leftarrow T'$.
15: **for** $l = i + 1$ to j **do**
16: Find s the source of T_{sub}.
17: $T_{sub} \leftarrow T_{sub} \setminus s$.
18: **if** $|T_{sub}| = k - 1$ **then**
19: **if** $l == i + 1$ and T' is a $(i + 2)$-size clique **then**
20: count $\leftarrow$ count + 1.
21: $I_l[T_{sub}] \leftarrow I_l[T_{sub}] + 1$.
22: **else if** $T' \in I_{i+2}$ **then**
23: **if** $l == i + 1$ and T' is a $(i + 2)$-size clique **then**
24: count $\leftarrow$ count + $I_{i+2}[T']$.
25: $I_l[T_{sub}] \leftarrow I_l[T_{sub}] + I_{i+2}[T']$.
26: Unmark all marked edges in the graph.

10.2 BatchFlips Implementation

Our algorithm uses the framework given in Section 8. We first instantiate the algorithm for BatchFlips($A, \mathcal{B}_{ins}, \mathcal{B}_{del}$), which creates a set of edge insertions and deletions from the flipped edges in A and appends these edges to $\mathcal{B}_{ins}$ and $\mathcal{B}_{del}$. The pseudocode is given in Algorithm 11. In parallel, for each edge that is flipped from (u, v) to (v, u) (Line 1), we add (u, v) to $\mathcal{B}_{del}$ (Line 2) and add (v, u) to $\mathcal{B}_{ins}$ (Line 3).

10.3 BatchInsert Implementation

We now instantiate BatchInsert for k-clique counting. The main basis of our BatchInsert and BatchDelete subroutines is to maintain our parallel hash tables throughout edge insertions and deletions. We first describe CliqueCountingBatchInsert, given in Algorithm 12. CliqueCountingBatchDelete is symmetric and is discussed in Section 10.4.

Before we dive into the details of the implementation of Algorithm 12, we first provide some intuition for how our algorithm implements our intuitive approach in Section 10.1. The key piece

of information that our algorithm maintains after all updates is how many sets of k vertices, C, can be a k-clique if a subset of $2 \leq i \leq k-1$ vertices, $C_i \subset C$, has edges between all pairs of vertices in the set. Table i keeps all C_i sets of vertices (as keys). In other words, the entry $I_i[C_i]$ precisely counts the number of unique sets of k distinct vertices, C, where the following two properties hold:

(1) $C_i \subset C$.
(2) For every $v \in C \setminus C_i$, the directed edge from v to w, (v, w), exists in G for every $w \in C_i$.

The bulk of Algorithm 12 is concerned with updating all of the tables I_i for $i \in [2, \ldots, k-1]$ such that the above counts hold for every entry. Using these counts, we can find the number of new k-cliques created by inserting $\mathcal{B}_{ins}$ by checking for each C, whether its largest incomplete clique without a source is completed. We can do this efficiently because (1) we do not need to check this individually for every C since our tables I_i already maintain these counts; and (2) if a largest incomplete clique without a source C_i is completed, then it must be incident to an edge update $(u, v) \in \mathcal{B}_{ins}$, where u is the new source of C_i and $v \in C_i$. Knowing (2), we can afford to enumerate sets of out-neighbors of C_i to determine whether C_i is a clique. If C_i is a clique, then we count all C that has it as its largest incomplete clique without a source by adding $I_i[C_i]$ to the cumulative count. The remaining parts of Algorithm 12 ensure that we do not over-count newly formed cliques.

We now describe Algorithm 12 in detail. For each edge insertion (u, v) where the edge is oriented from u to v (Line 4), we iterate through *all* possible subsets of i out-neighbors of u (*excluding* v, since we know v must be included in the clique) where $i \in [1, \ldots, k-2]$ (Lines 5 and 6). We iterate through $i \in [1, \ldots, k-2]$ because u and v necessarily need to be included in the clique. This is to account for all possible largest incomplete cliques without a source that are currently in our hash tables. In order to find whether $\mathcal{B}_{ins}$ completes any of these cliques, we must find these cliques by performing this enumeration. Let T be the subset of out-neighbors picked. We consider all cliques of size $i+2$ consisting of the vertices in $T' = T \cup \{u, v\}$. Then, Line 7 provides a canonical order for the vertices in T' so that we can search for T' in I_i. Note that we need to avoid duplicate counting. To avoid duplication, we use the order of the edge insertions in BatchInsert (Line 3) and assign the task of updating the clique count to the first insertion in this order, (u, w), where $w \in T'$. Hence, the if statement in Line 8 checks that all of these above conditions are satisfied. The if statement in Line 9 checks whether the newly inserted edges create a new clique, and if not (Line 11), the algorithm then finds the largest incomplete clique without a source, C', that contains a subset of the vertices in T' (Line 12). The algorithm then sets a parameter j to be the size of C' (Line 13). If T' is a completed clique, it passes the check on Line 9 and we assign $j = 2$ (Line 11). Now, we consider two possible scenarios.

First, (u, v) along with the other edge insertions in $\mathcal{B}_{ins}$ could complete a largest incomplete clique without a source. In this case, we should increase the k-clique count if u is also the new source of the clique. Furthermore, Lines 19 and 23 check if the clique is completed. If u is a new source, the clique for which it is the source is completed, and $T' \in I_{i+2}$, then we increment the clique count with the value $I_{i+2}[T']$ (Lines 22 and 23). The value stored in $I_{i+2}[T']$ is the number of new cliques that are created if T' is completed. Note that Line 24 is only called if $|T'| < k$ since $l \leq k-1$ and Line 18 handles the case when $|T'| = k$. This is because we do not store size-k sets of vertices in any of the tables; we do not need to store their values because we can enumerate them directly by checking all $(k-1)$-size subsets of out-neighbors of every u in every edge insertion (u, v). We denote the ordered set of vertices that gives the key in table I_l by T_{sub} (initially setting $T_{sub} \leftarrow T'$ (Line 14)). If $T' \notin I_{i+2}$ and the size of T_{sub} is $k-1$ (implying the size of T' is k), then we directly increment the clique count by 1 (Lines 18 to 20). As before, in this case, we directly enumerate the new clique for the edge insertion (u, v) without needing to check the tables. This also means that u is the source of the newly created k-clique consisting of the vertices in T'.

After we update the k-clique count, we must then update the $I_i[C_i]$ counts for each $C_i \subset T'$. We need to update these counts because now T' contains a vertex $v \in T' \setminus C_i$ where there exists an edge (v, w) for every $w \in C_i$ (this vertex did not exist previously). Thus, the count for C_i must be incremented by $I_{i+2}[T']$ if $i+2 \leq k-1$, and 1 if $i+2 = k$. This is because, as previously discussed, intuitively, $I_{i+2}[T']$ stores the number of k-cliques that would be created if T' were completed, so we must similarly maintain the number of k-cliques created now that each C_i is completed. We prove this more concretely in Section 10.5. When T' is a clique, there exists a C_i for every $2 \leq i < |T'|$ whose entry $I_i[C_i]$ needs to be updated. So, Line 15 loops through each of these possible sizes and the entries are updated by Line 21 or Line 25 depending on whether $|T'| = k$.

The second scenario is that (u, v) and the other edge insertions in $\mathcal{B}_{ins}$ do not complete a clique but create a new source among the vertices in T'. The means that we need to find a new largest incomplete clique without a source within T'. Again, to avoid duplication, we assign the task to the earliest edge update that is incident to u. Similar to the case when T' is a clique, the algorithm also needs to update tables I_{i+1} to I_j in this case (Line 15). However, we do not update *all* of the tables since T' still has a largest incomplete clique without a source (Line 12). Let $l \in [j, i+1]$, the table I_j is updated with the number of cliques that would be counted by it if it were the largest incomplete clique without a source. We need to update all of these tables (instead of just table I_j) in order to be able to handle deletions. This is due to the fact that when a smaller clique becomes incomplete due to a batch of deletions, it may cause a larger k-clique to become incomplete. We cannot afford to find all such affected k-cliques; thus, we must store this information in the tables.

To compute the key for table I_l, we need to remove the source of T_{sub} from T_{sub} (Lines 16 and 17). Then, there are two cases we must consider (Line 18 and Line 22). In the first case, when $|T'| = k$, u is a newly created source for a new potential k-clique (Line 18); thus, no entries in the tables have counted T' yet and we increment the count of $I_{k-1}[T_{sub}]$ by 1 (Lines 18 and 21) so that T' will be counted when T_{sub} is completed as the largest incomplete clique without a source. In the second case (Line 22), T' is already an entry in table I_{i+2}; this means that it already counts the a number of k-cliques that exist if T' is a clique. In this case, we increase the value for $I_l[T_{sub}]$ by $I_{i+2}[T']$ since by definition of the values we store in $I_l[T_{sub}]$, if T_{sub} is a clique, then all the k-cliques that are counted

Algorithm 13 CliqueCountingBatchDelete($\mathcal{B}_{del}$)

Input: A batch $\mathcal{B}_{del}$ of unique and valid deletion updates.
Output: An updated k-clique count and updated data structures.

1: Let count be the current count of the number of k-cliques in the graph.
2: Insert all edges in $\mathcal{B}_{del}$ into the graph in the orientation specified by $\mathcal{B}_{del}$. Mark all edges in $\mathcal{B}_{del}$ in the graph.
3: Let R be the order of the edges in $\mathcal{B}_{del}$.
4: **parfor** each edge $(u, v) \in \mathcal{B}_{del}$ **do** ▷ The edge is oriented from u to v.
5: **for** $i \in [k-2, \ldots, 0]$ **do**
6: **parfor** each subset T of i out-neighbors of u **do**
7: Let T' be the ordered set of $T \cup \{u, v\}$ sorted by vertex index.
8: **if** u is the source of T' **and** (u, v) is the earliest in R out of all marked edges from u to $w \in T'$ **then**
9: **if** all edges between each pair $x, y \in T'$ exists **then**
10: $j \leftarrow 2$.
11: **else**
12: Find largest incomplete clique without a source, C', in T'.
13: Let j be the size of C'.
14: $T_{sub} \leftarrow T'$.
15: **for** $l = i + 1$ to j **do**
16: Find s the source of T_{sub}.
17: $T_{sub} \leftarrow T_{sub} \setminus s$.
18: **if** $|T_{sub}| = k - 1$ **then**
19: **if** $l == i + 1$ and T' is a clique **then**
20: count $\leftarrow$ count -1.
21: $I_l[T_{sub}] \leftarrow I_l[T_{sub}] - 1$.
22: **else if** $T' \in I_{i+2}$ **then**
23: **if** $l == i + 1$ and T' is a clique **then**
24: count $\leftarrow$ count $- I_{i+2}[T']$.
25: $I_l[T_{sub}] \leftarrow I_l[T_{sub}] - I_{i+2}[T']$.
26: Delete all marked edges in the graph.

when T' is a clique will now be counted when T_{sub} is completed as the largest incomplete clique without a source (Line 25).

10.4 BatchDelete Implementation

Our CliqueCountingBatchDelete algorithm is nearly identical to our CliqueCountingBatchInsert algorithm; in places where we assign clique counts in the insertion algorithm, we instead remove clique counts in the deletion algorithm. Such changes are expected since deletions of edges remove cliques from the count and also remove assignments of cliques to largest incomplete cliques without a source. The pseudocode is provided in Algorithm 13. The few changes to the algorithm are highlighted in blue.

10.5 Correctness

To prove the correctness of our algorithm, we first show that Algorithm 12 and Algorithm 13 accurately store the counts associated with the largest incomplete cliques without sources. For simplicity, we provide separate lemmas for Algorithm 12 and Algorithm 13, although fundamentally, the proof techniques are the same for both algorithms.

We use the following notation in Lemma 10.2 and Lemma 10.3. Let c_L be the number of sets of k vertices in the graph which do not form a clique, contains a source, and whose largest incomplete cliques without sources is the set of vertices in L *after processing the current, input batch of updates*. We show that after running Algorithm 12 or Algorithm 13, $I_{|L|}[L] = c_L$. In fact, we show an even stronger lemma; suppose that J is a set of vertices in the graph where $2 \leq |J| \leq k - 1$. If we remove all edges between pairs of vertices in J, let c_J be the number of sets of k vertices that do not form a clique, contains a source, and whose largest incomplete cliques without sources is the set of vertices in J. We show that $I_{|J|}[J] = c_J$. This stronger form of the lemma is not necessary if we only consider insertion updates; however, under deletion updates, we require this stronger lemma in order to prove correctness. It is sufficient to assume that all data structures are maintained corrected at the beginning of Algorithm 12 and Algorithm 13 and they remain correct at the end of the algorithms (since by induction, this would prove that the data structures are always correctly maintained).

Lemma 10.2. *After running Algorithm 12 on $\mathcal{B}_{ins}$, $I_{|J|}[J] = c_J$ for every c_J where $2 \leq |J| \leq k - 1$.*

Proof. We prove this lemma via induction on the table index i, starting with $i = k - 1$. We first prove our base case for $i = k - 1$. In Algorithm 12, the value stored in $I_{k-1}[J]$ is only ever incremented in Line 21 since this is the only time when table I_{k-1} can be modified (the condition in Line 22 is never satisfied for any entries in table I_{k-1}). By the condition given in Line 8, $I_{k-1}[J]$ is only incremented when T' has a source s where $J = T' \setminus \{s\}$. Furthermore, the condition that $I_{k-1}[J]$ is incremented by the earliest edge update incident to s ensures that it is incremented at most once by each T'. The number of sets c_J of vertices T' where $J \in I_{k-1}$ is the largest incomplete clique without a source (if all edges in J are removed) is precisely the number of vertices s in the graph with edges directed into all vertices in J. Our argument above shows that $I_{k-1}[J]$ is incremented exactly once for each such vertex s; furthermore, it is incremented only if s is adjacent to an edge update $(s, x) \in \mathcal{B}_{ins}$ and $x \in J$. This proves our base case.

We assume for our induction step that $I_{|J|}[J] = c_J$ for all tables I_j for $j \in [k-1, \ldots, k-l]$ and prove the lemma holds for table I_{k-l-1}. The value $I_{k-l-1}[J]$ is increased in Line 25. Every T' with k vertices increases the value of $I_{k-l-1}[J]$ by 1 if its largest incomplete clique without a source has size $\leq |J|$. This is easy to see since if all edges from J are removed, then J would be the largest incomplete clique without a source for T'. By our induction hypothesis, the counts of these T's are correctly stored in tables I_j for $j \in [k-1, \ldots, k-l]$. Line 8 ensures that only one edge update is responsible for incrementing $I_{k-l-1}[J]$ for each T'; furthermore, it ensures that $I_{k-l-1}[J]$ is incremented with the value stored in $I_{|C|}[C]$ where $C \subseteq T'$ is the previous largest incomplete clique without a source before the current batch $\mathcal{B}_{ins}$ of insertions. $J \subset C$ by definition, and J is guaranteed to be the largest incomplete clique without a source (after the edge insertions in $\mathcal{B}_{ins}$) by our argument above. In addition, each T' is counted in at most one $C \subset T'$ in each table I_j where $j \in [k-1, \ldots, k-l]$. This is true by our induction hypothesis since each T' has one unique C where $|C| = j$ which is the largest incomplete clique without a source (if the edges in C are removed).

The last step we need to prove in order to prove our induction hypothesis is that $I_{k-l-1}[J]$ is incremented by 1 for T' with exactly one $I_{|C|}[C]$ where $C \subset T'$. We prove this via contradiction. Suppose there are two subsets $C' \subset C \subset T'$ which are used to increment $I_{k-l-1}[J]$. Let $s' \in C'$ be the source of C' and $s \in C$ be the source of C. This means that in order to satisfy Line 8, s and s' must be incident to some update $(s, x) \in \mathcal{B}_{del}$ (resp. $(s', x') \in \mathcal{B}_{del}$) where

$x, x' \in T'$. This means that C was the previous largest incomplete clique without a source for T' and so C' would not contain a count for T' by our induction hypothesis. Since, we process the tables in Line 5 starting with table I_2 in increasing order of table index, $I_{k-l-1}[J]$ cannot be incremented with the count for T' from C', a contradiction. Thus, $I_{|C|}[C]$ correctly counts all T' and hence, $I_{k-l-1}[J]$ is incremented exactly once for each T' and we have proven our inductive step. □

The proof of the property for Algorithm 13 is almost identical to Lemma 10.2 except to account for the few changes shown in blue in Algorithm 13. For simplicity, we present only the parts of the proof that requires more effort than replacing decrement for all mentions of increment in the proof of Lemma 10.2.

Lemma 10.3. *After running Algorithm 13 on $\mathcal{B}_{del}$, $I_{|J|}[J] = c_J$ for every c_J where $2 \leq |J| \leq k-1$.*

Proof. We prove this lemma via induction on the table index i, starting with $i = k-1$. The proof of our base case for $i = k-1$ directly follows from the proof of the base case in Lemma 10.2 when we replace instances of increment with decrement.

We assume for our induction step that $I_{|J|}[J] = c_J$ for all tables I_j for $j \in [k-1, \ldots, k-l]$ and prove the lemma holds for table I_{k-l-1}. The value $I_{k-l-1}[J]$ is increased in Line 25. The proof of the inductive step follows from the proof of the inductive step in the proof of Lemma 10.2 by replacing instances of increase by decrease, except for the last step which we prove below.

The last step we need to prove in order to prove our induction hypothesis is that $I_{k-l-1}[J]$ is decremented by 1 for T' with exactly one $I_{|C|}[C]$ where $C \subset T'$. We prove this via contradiction. The initial setup is the same as the setup in the proof of Lemma 10.2. Suppose there are two subsets $C' \subset C \subset T'$ which are used to decrement $I_{k-l-1}[J]$. Let $s' \in C'$ be the source of C' and $s \in C$ be the source of C. This means that in order to satisfy Line 8, s and s' must be incident to some update $(s, x) \in \mathcal{B}_{ins}$ (resp. $(s', x') \in \mathcal{B}_{ins}$) where $x, x' \in T'$.

This means that C *is now* the largest incomplete clique without a source for T' after processing the deletions in $\mathcal{B}_{del}$. Thus, because we process the tables in *decreasing order* by table index, starting with table I_2 (Line 5), C satisfies the conditions in Line 8 and by Line 25, C would have deleted the count of T' from $I_{|C'|}C'$. Thus, C' would not contain a count for T' and $I_{k-l-1}[J]$ cannot be incremented with the count for T' from C', a contradiction. $I_{|C|}[C]$ correctly counts all T' by our induction hypothesis and hence, $I_{k-l-1}[J]$ is decremented exactly once for each T' and we have proven our inductive step. □

We are now ready to prove that our algorithms correctly return the k-clique count provided batches of updates.

Theorem 10.4. *Our algorithms, Algorithm 12 and Algorithm 13, correctly returns the number of k-cliques in a given input graph, $G = (V, E)$, provided batches of updates $\mathcal{B}_{ins}$ and $\mathcal{B}_{del}$, respectively.*

Proof. Provided Lemma 10.2 and Lemma 10.3, we only need to show the following: given $\mathcal{B}_{ins}$, each k-clique C completed by $\mathcal{B}_{ins}$ (i.e. contains at least one edge in $\mathcal{B}_{ins}$), is counted exactly once, and by exactly one update edge incident to the source of its largest incomplete clique without a source (*prior* to the insertions); given $\mathcal{B}_{ins}$, each k-clique C destroyed by $\mathcal{B}_{del}$ (i.e. contains at least one edge in $\mathcal{B}_{del}$), is subtracted exactly once, and by exactly one update edge incident to the source of its largest incomplete clique without a source (*after* the deletions).

We first prove the above is true for insertions. The if statement in Line 8 ensures at most one update edge for a set of vertices $C \subset T'$, where T' is a newly formed clique, increments the clique count. Now, we prove that at most one subset of vertices increments the clique count for T'. Suppose for contradiction two sets of vertices $C' \subset C \subset T'$ increments the total clique count by 1 for T'. Then, in order to pass the if statement in Line 8, the sources of both C' and C must be adjacent to updates in $\mathcal{B}_{ins}$ that point to vertices in C' and C. Since $C' \subset C$, C was the previous largest incomplete clique without a source for T'. By Lemma 10.2, C' does not contain the count for T' and thus, only C increments the total clique count by 1 for T', a contradiction.

To prove that at least one subset of vertices increments the clique count for T', suppose that C was the previous largest incomplete clique without a source for T' but C does not increment the clique count. Since T' is a new clique, it must be incident to at least one edge update in $\mathcal{B}_{ins}$. Since C does not increment the clique count, it must not have found a source (and cannot satisfy Line 9). (It must satisfy Line 19 or Line 23 since T' is a clique and we iterate through all possible i). Since C does not have a source, by Observation 10.1, it must be missing at least one edge. Then, T' is not a clique, a contradiction.

The proof follows symmetrically for Lemma 10.3 except that instead of the previous largest incomplete clique without a source, we care about the largest incomplete clique without a source *after* processing $\mathcal{B}_{del}$. Suppose for contradiction two sets of vertices $C' \subset C \subset T'$ decrement the clique count. Then, their sources must both be incident to edge updates. Since, $C' \subset C$, C is processed first by Line 5 using Lines 15, 21 and 25. This means that the count of T' would have been subtracted from $I_{|C'|}[C']$ and it cannot decrement the clique count by 1 for T', a contradiction. Suppose instead, that C is the largest incomplete clique without a source for T' after processing $\mathcal{B}_{del}$ and it does not decrement the clique count. Either one of two scenarios can occur: either $I_{|C|}[C]$ no longer has the count for T' or the source s of C is not incident to any updates. No C'' where $C \subset C''$ can decrement $I_{|C|}[C]$ for T' since by our assumption, the source of C'' is not incident to any updates directed into vertices in C''. Thus the first scenario cannot occur and we consider the second scenario where the source s must not be incident to an edge update directed into the vertices in C. Then, s still has all its directed edges to the vertices in C and so is the source of C. This means that C has a source and cannot be the largest incomplete clique *without a source*, a contradiction. □

Together with the proof of correctness of our framework, Section 8, our algorithm correctly provides the k-clique count provided a batch of updates, $\mathcal{B}$.

10.6 Work and Depth Analysis

We note for the following result that α is defined as $\max(\alpha_b, \alpha_a)$ where α_b is the arboricity before the current batch of updates is processed and α_a is the arboricity after the current batch of updates is processed.

Theorem 10.5. *We obtain a batch-dynamic k-clique counting algorithm that takes $O(\alpha|\mathcal{B}|\log^2 n)$ amortized work and $O(\log^2 n \log\log n)$ depth w.h.p., using $O(m\alpha^{k-2} + n\log^2 n)$ space.*

Proof. We first show the work, depth, and space of our algorithms, Algorithms 11 to 13, and then use Theorem 8.1 to show the bounds for our algorithm. Note that the increments and decrements to the global k-clique count can be performed in $O(\log n)$ depth in parallel by writing each update to an array, and then using parallel reductions at the end to update the global k-clique count. We use the same strategy for updating the hash table counts. Furthermore, our parallel hash table primitives allow us to concurrently modify elements in parallel in $O(\log n)$ depth w.h.p.

In Algorithm 11, the batches $\mathcal{B}_{ins}$ and $\mathcal{B}_{del}$ can be obtained in $O(|\mathcal{B}|\log^2 n)$ work and $O(\log n)$ depth. Note that by construction, $|\mathcal{B}_{ins}|, |\mathcal{B}_{del}| = O(|\mathcal{B}| + |A|) = O(|\mathcal{B}|\log^2 n)$. All edges can be checked in parallel (Line 1) and inserted into parallel dynamic arrays; we can also use a simple parallel filter. For the remainder of this proof, we discuss the work and depth complexity for a batch $\mathcal{B}_{ins}$ of edge insertions in Algorithm 12; the deletion algorithm (Algorithm 13) has the same work, depth, and space complexity.

All edge insertions are processed in parallel using a parallel loop (Line 4). We then run a sequential for loop of depth $O(k)$ (Line 5). Let i be the current index of the sequential for loop. In order to process edge insertions (u, v), where u is a source, we iterate in parallel over all sets T of $i+1$ out-neighbors of u including v. Since there are at most $O(\alpha)$ out-neighbors of u, and since v is necessarily included, we have $\binom{c\alpha}{i} = O(\alpha^i)$ possible sets T (assuming constant k). For constant k, we perform a constant number of parallel hash table operations and checks for the existence of edges per set T (Lines 9, 10 and 12 to 14). We make $O(k)$ iterations of the for loop in Line 15; updating the hash tables (Lines 21 and 25) require $O(k)$ total work per edge update. Checking for the source of T_{sub} over all T_{sub} requires $O(k^2)$ work per edge update (Line 16). Thus, per edge insertion (u, v), for constant k, we incur $O(\alpha^i)$ work and $O(\log n)$ depth w.h.p. Over all $i \in [0, \ldots, k-2]$, this results in $\sum_{i=0}^{k-2} O(\alpha^i) = O(\alpha^{k-2})$ total work over all i, w.h.p. The depth is $O(\log^* n)$ w.h.p. due to the hash table operations and updating the table values by writing to an array and using a parallel reduction for each entry results in $O(\log n)$ depth.

Lastly, we update the global k-clique count by writing each update to an array and using a parallel reduction at the end, which maintains the same work and depth bound.

Processing the entire batch of insertions in parallel, we have $O(\alpha^{k-2}|\mathcal{B}|\log^2 n)$ amortized work and $O(\log n)$ depth w.h.p. Thus, in total, our k-clique counting algorithm takes $O(\alpha^{k-2}|\mathcal{B}|\log^2 n)$ amortized work and $O(\log^2 n)$ depth w.h.p. by Theorem 8.1.

Our space usage is proportional to the space required to store the contents of the parallel hash tables I_i for $i \in [2, \ldots, k-1]$. By construction, for each edge insertion (u, v), we create at most $\sum_{j=0}^{k-2} O(\alpha^{j-2}) = O(\alpha^{k-2})$ hash table entries across all I_i. This follows directly from our work analysis. Thus, in total, we use space proportional to $O(m\alpha^{k-2})$. □

10.7 Comparison with Previous Work

The best-known batch-dynamic algorithm for k-clique counting for graphs with low arboricity is given by Dhulipala et al. [29]. They give a $O(|\mathcal{B}|m\alpha^{k-4})$ expected work and $O(\log^{k-2} n)$ depth w.h.p. algorithm using $O(m + |\mathcal{B}|)$ space. Our algorithm improves upon the work of this previous result when $m = \omega(\alpha^2 \log^2 n)$. Note that $\alpha \leq \sqrt{m}$ [22]. Furthermore, in real-world graphs, often $\alpha \ll \sqrt{m}$, since real-world graphs tend to have small arboricity.

Our algorithm achieves better depth for all $k > 4$. For 4-cliques, our depth matches the previous algorithm while for larger cliques, we achieve a better depth. Finally, we obtain these gains with an increase in space of $O(\alpha^{k-2} + \log^2 n)$ multiplicative factor, but for bounded arboricity graphs, this increase in space is small.

11 COLORING ALGORITHMS

The vertex coloring problem looks to assign colors to assign colors to vertices such that no two adjacent vertices are assigned the same color. A c-vertex coloring uses at most c colors to color all vertices in the graph. In this section, we present two batch-dynamic algorithms. Although our algorithms are based heavily on the sequential algorithms by Henzinger et al. [47], we present them as an example of using our framework. One maintains an (***explicit***) coloring over the vertices and one maintains an ***implicit*** coloring. In the explicit setting, a valid coloring is always maintained in the graph among all vertices. In the implicit setting, the algorithm maintains a set of data structures and on ***queries*** of one or more vertices, returns a coloring that is valid on the induced subgraph of the queried vertices. Thus, in the implicit setting, both updates and queries could take $\Omega(1)$ work to process. Below, we give our vertex coloring algorithms.

11.1 Explicit $O(\alpha \log n)$-Coloring

In this section, we present a parallel batch-dynamic, randomized $O(\alpha \log n)$-coloring algorithm that is robust against an oblivious adversary and uses $O(\log^2 n)$ amortized work, matching the amortized running time in the sequential setting. Notably, α is the current arboricity of the graph, after processing the current batch of updates. This algorithm is inspired by the coloring algorithm of Henzinger et al. [47], and directly uses the PLDS.

Sequential Explicit Coloring Algorithm of Henzinger et al. [47]. The explicit vertex coloring algorithm of Henzinger et al. [47] uses a separate palette of colors for each level in the LDS. When a vertex moves to a new level, it chooses a color uniformly at random from among the free colors in the palette at its level; specifically, the free colors are colors that are not occupied by its neighbors. If an edge insertion occurs between two vertices with the same color, then an arbitrary endpoint chooses a new color uniformly at random from the free colors in its palette.

Since the sequential algorithm processes one vertex at a time, it does not have to deal with color conflicts when more than one vertex chooses a free color from the same palette. However, in the batch-dynamic case, this is an issue since more than one vertex on the same level may need to choose a free color. We show that allowing such vertices to keep choosing colors is sufficient to ensure

Algorithm 14 ExplicitColoringBatchFlips($A, \mathcal{B}_{ins}, \mathcal{B}_{del}$)

Input: A set of edge flips A.
Output: A list of vertices which changed levels.
1: $S \leftarrow \emptyset$.
2: **parfor** each edge $(u, v) \in A \cup \mathcal{B}_{ins} \cup \mathcal{B}_{del}$ **do**
3: **if** u (resp. v) changed levels **then**
4: $S \leftarrow S \cup \{u\}$ (resp. $\{v\}$).
5: **return** S.

Algorithm 15 ExplicitColoringBatchDelete($\mathcal{B}_{del}$)

Input: A batch $\mathcal{B}_{del}$ of unique and valid insertion updates.
Output: A valid $O(\alpha \log n)$-coloring.
1: **parfor** each edge $\{u, v\} \in \mathcal{B}_{del}$ **do**
2: **if** $u \in S$ (resp. $v \in S$) **then**
3: $S \leftarrow S \setminus \{u\}$ (resp. $\{v\}$).
4: **while** u (resp. v) has a neighbor with the same color **do**
5: Recolor u (resp. v) with a free color from $P_{\ell(u)}$ (resp. $P_{\ell(v)}$) picked uniformly at random.

Algorithm 16 ExplicitColoringBatchInsert($\mathcal{B}_{ins}$)

Input: A batch $\mathcal{B}_{ins}$ of unique and valid insertion updates.
Output: A valid $O(\alpha \log n)$-coloring.
1: **parfor** each edge $\{u, v\} \in \mathcal{B}_{ins}$ **do**
2: **if** $c(u) = c(v)$ **then**
3: **while** u has a neighbor with the same color **do**
4: Recolor u with a free color from $P_{\ell(u)}$ picked uniformly at random.
5: $S \leftarrow S \setminus \{u\}$.
6: **parfor** each remaining vertex $v \in S$ **do**
7: **while** v has a neighbor with the same color **do**
8: Recolor v with a free color from $P_{\ell(v)}$ picked uniformly at random.

both work-efficiency and low depth, w.h.p., provided we give a large enough palette.

Batch-Dynamic $O(\alpha \log n)$-Vertex Coloring. As in the previous sections, we use our framework given in Section 8 for our coloring algorithms. The pseudocode for our implementations of the methods are given in Algorithms 14 to 16. Given that Algorithm 16 and Algorithm 15 are very similar to each other, we explain all three algorithms here.

First, Algorithm 14 determines the set of vertices which changed levels after processing the batches of insertions and deletions. We can find these vertices in parallel (Line 2). The vertices which changed edges are added to the set S (Line 4 which can be accessed by Algorithm 16 and Algorithm 15.

Each level $\ell \in g_i$ is initialized with a unique palette with $2 \cdot (2 + 3/\lambda)(1 + \varepsilon)^i$ colors. Vertices on level ℓ will be colored only with colors from the palette on level ℓ. P_ℓ denotes the palette for level ℓ. A free color for a vertex v is a color from $P_{\ell(v)}$ that is not occupied by any neighbors of v.

When a vertex moves up or down one or more levels, it recolors itself using the palette of the new level $\ell \in g_i$.[12] In Algorithm 15, because deletions cannot cause two neighboring vertices to have the same color, we only need to recolor the vertices which changed levels (Line 2). To pick a free color (Algorithm 16 Line 4, Algorithm 15 Line 5), the vertex v looks at the colors occupied by all its up-neighbors and picks a color that does not collide with the colors of any of its up-neighbors. We only need to check the up-neighbors because the palettes are distinct across levels. In fact, a vertex can only conflict with the neighbors in its own level, but since we keep all up-neighbors in a single data structure, we check all up-neighbors.

In addition to checking the vertices which changed levels, given a batch of insertions, $\mathcal{B}_{ins}$, in Algorithm 16, we iterate over all insertions in parallel (Line 1) and check whether any insertions are between two vertices with the same color (Line 2). Then, for each edge insertion $\{u, v\}$ between two vertices with the same color, we arbitrarily select one vertex, u, to recolor itself (Line 4). v selects a free color uniformly at random from its palette (Line 4). If v still conflicts with any of its up-neighbors (Line 3), it recomputes its palette of free colors again by looking at the colors of its up-neighbors and picks a color uniformly at random. This process repeats until no vertices conflict with their up-neighbors in color. Finally, the remaining vertices which changed levels choose colors from their respective palettes (Lines 6 to 8).

Analysis. Since each level has a unique palette and at most $(2 + 3/\lambda)(1 + \varepsilon)^i$ neighbors of a vertex v can be on the same level as v if $\ell(v) \in g_i$, v has at least $(2 + 3/\lambda)(1 + \varepsilon)$ free colors that it can choose from its palette of size $2 \cdot (2 + 3/\lambda)(1 + \varepsilon)^i$. We first show that this strategy only requires $O(\alpha \log n)$ colors. Part of the proofs of Lemmas 11.1 and 11.2 follow the analysis provided in [47] but we present it for completeness.

Lemma 11.1. *At most $O(\alpha \log n)$ colors are required in our algorithm.*

Proof. Each level $\ell \in g_i$ has $O((1+\varepsilon)^i)$ colors (assuming λ is constant). We showed in Lemma 5.13 that our coreness estimate is upper bounded by $(2 + \varepsilon)k(v)$. This means that the largest group index where a vertex is on a level in the group is $\log_{(1+\varepsilon)}((2 + \varepsilon)k(v)) + 1$. Hence, $i = \max_v \left(\log_{(1+\varepsilon)}(k(v)) + \log_{(1+\varepsilon)}(2 + \varepsilon) + 1\right)$. Since each group contains $O(\log n)$ levels, the number of colors used is

$$\sum_{i=0}^{\log_{(1+\varepsilon)}(\max_v(k(v)))+\log_{(1+\varepsilon)}(2+\varepsilon)+1} 2(2 + 3/\lambda)(1 + \varepsilon)^i \log n$$
$$= O(\max_v(k(v)) \log n) = O(\alpha \log n).$$

□

We now show that our procedure requires $O(\mathcal{B} \log^2 m)$ amortized work using PLDS.

Lemma 11.2. *For a batch $\mathcal{B}$, under the oblivious adversary assumption,[13] our coloring algorithm requires $O(|\mathcal{B}| \log^2 n)$ amortized work, in expectation.*[14]

Proof. For a vertex that moves to a different level, the work to recolor it can be charged to the work of the PLDS update procedure. The bulk of this proof is devoted to proving this fact.

[12] This step is necessary to maintain our bound in terms of the current arboricity, α, for the number of colors used in the coloring.

[13] An oblivious adversary cannot see our algorithms outputs (i.e., they cannot see our coloring) before determine the set of updates.

[14] We can show the work to be $O(|\mathcal{B}| \log^3 n)$ amortized w.h.p. somewhat tediously using the Chernoff bound. However, the bound requires an additional $O(\log n)$ factor of work compared to the $O(|\mathcal{B}| \log^2 n)$ amortized work bound in expectation.

First, we show that the oblivious adversary cannot cause recolorings too often via adversarial edge insertions between two vertices with the same color. The proof for this part is similar to the proof of Lemma 8 in [47]. To show this, we crucially rely on the fact that the adversary *cannot* see the colors of the vertices before they pick the updates. For an edge insertion (u, v) between two vertices, u and v, that are on the same level $\ell \in g_i$ and causes a conflict, u and v have at most $(2 + 3/\lambda)(1 + \varepsilon)^i$ neighbors (at the same level, using the same palette) but $2 \cdot (2 + 3/\lambda)(1 + \varepsilon)^i$ total colors in its palette. The algorithm arbitrarily picks one of the two endpoints, without loss of generality u, to recolor itself. u has at least $(2+3/\lambda)(1+\varepsilon)^i$ free colors and it picks each color uniformly at random from these $(2 + 3/\lambda)(1 + \varepsilon)^i$ free colors.[15] This means that u picks any particular color c in its palette with probability at most $\frac{1}{(2+3/\lambda)(1+\varepsilon)^i}$. The same argument holds for a vertex that moved to a new level and needs to be recolored. Since the adversary is oblivious, they have to guess which color u picked. Even assuming the much stronger assumption that the adversary knows the colors of all vertices except u (it does not in actuality), the adversary still only has at most a $\frac{1}{(2+3/\lambda)(1+\varepsilon)^i}$ chance of picking c (u's color) and creating an edge insertion between u and a vertex with color c. Thus in, expectation, the adversary must create $(2 + 3/\lambda)(1 + \varepsilon)^i$ edge insertions incident to u before they pick one that conflicts with u's color. The $O((2 + 3/\lambda)(1 + \varepsilon)^i)$ cost of finding a color for u can be amortized over these edge insertions.

For the vertices that were recolored due to level movements, we can charge their cost to the cost of moving levels. In expectation, the vertex tries at most two times (by what we showed above) before it is successfully recolored, resulting in $O(1)$ total cost, in expectation. Thus, the amortized update time is equal to the number of conflicts and vertices that moved to a different level. This takes $O(|\mathcal{B}| \log^2 n)$ work in expectation since there are $O(|\mathcal{B}| \log^2 n)$ edge flips and updates and is the same as the PLDS update time.

In addition to the cost of recoloring due to adversarial insertions, recall that our batch-dynamic algorithm also requires multiple vertices to keep picking colors uniformly at random until they pick unique colors not occupied by their neighbors (Algorithm 15-Line 5, Algorithm 16-Line 4). We need to show that this procedure does not add too much additional cost to the cost of recoloring due to adversarial updates. In fact, next we show that additionally picking random colors until all vertices pick a non-conflicting color does not add additional work, asymptotically, w.h.p. It is easy to show that this running time also holds with high probability. Given a set X of vertices in level $\ell \in g_i$ that randomly picked the same color or that moved to a different level, in expectation, after a round of recoloring, the number of vertices in X that again result in conflicts is $\frac{|X|}{(2+3/\lambda)(1+\varepsilon)^i} \leq \frac{|X|}{(2+3/\lambda)} \leq \frac{|X|}{2}$. Since each vertex is independently picking a color, we can show via the Chernoff bound that with probability at most $\exp(\varepsilon^2|X|/6)$, more than $(1 + \varepsilon) \cdot \frac{|X|}{2}$ vertices need to pick colors again. Thus, when $|X| \geq c \log n$, for large enough constant $c > 0$, with high probability, the number of vertices that need to re-pick their colors decreases by a factor of 2. Then, each time we recolor, the number of vertices that obtain their final color decreases by a constant fraction w.h.p., and we can charge the cost of all subsequent recolorings to the first time we recolor the vertices.

When $|X| < c \log n$, the probability that any $c \log n$ consecutive trials results in a vertex $v \in X$ picking the same color is $(\frac{1}{2})^{c \log n} \leq \frac{1}{n^c}$. By the union bound over the $|\mathcal{B}|$ updates, the total probability that any two vertices conflict is $\frac{|\mathcal{B}|}{n^c}$. We can pick $c \geq 3$ to obtain with high probability that each insertion results in $O(\log n)$ conflicts. Thus, the amortized update time is $O(\log n)$, w.h.p., since each edge gets charged $O(\log n)$ for up to two endpoints. The total amortized work is $O(|\mathcal{B}| \log^2 n)$ which can be charged to the time necessary to perform the orientation algorithm. By Theorem 8.1, we obtain that the total work of our algorithm is $O(|\mathcal{B}| \log^2 n)$, in expectation, $O(|\mathcal{B}| \log^3 n)$ w.h.p. □

Lemma 11.3. *Our coloring algorithm requires $O(\log^2 n \log \log n)$ depth per batch, with high probability.*

Proof. All vertices that need to change colors pick their new colors independently in parallel, and picking a new color takes $O(1)$ depth. All vertices, first, in parallel, record the colors of their up-neighbors. Then, vertices, in parallel, pick colors not occupied by their up-neighbors.

In the remainder of the proof, we prove that we need $O(\log n)$ depth w.h.p. to resolve the conflicts resulting from multiple neighboring vertices picking the same color. The probability that a vertex conflicts with a neighbor is at most $\frac{1}{(2+3/\lambda)(1+\varepsilon)^i} \leq \frac{1}{(2+3/\lambda)} \leq \frac{1}{2}$. The probability that we still have conflicts after $c \log n$ tries, for some constant $c > 0$, is then at most $(\frac{1}{2})^{c \log n} \leq \frac{1}{n^c}$. Picking $c \geq 2$ and applying a union bound over all vertices gives a polynomially small probability that conflicts occur after $c \log n$ tries. Thus, we need to randomly choose colors $O(\log n)$ times, w.h.p..

Since the depth of picking a color for each vertex is $O(1)$, the total depth for picking colors is $O(\log n)$ w.h.p. This depth is additive to the depth of our orientation algorithm because we first use our PLDS to move vertices to their final levels and then recolor the vertices. By Theorem 8.1, the overall depth is $O(\log^2 n \log \log n)$ w.h.p.. □

Theorem 11.4. *Our batch-dynamic $O(\alpha \log n)$-vertex coloring algorithm requires $O(|\mathcal{B}| \log^2 n)$ amortized work, $O(\log^2 n \log \log n)$ depth, w.h.p., using $O(m + n \log^2 n + \alpha \log n)$ space.*

Proof. Our work and depth bounds hold by Lemma 11.2 and Lemma 11.3, respectively. Finally, our coloring algorithm uses an additional space equal to the total number of colors used by the algorithm to maintain the palettes. Thus, the algorithm requires an additional $O(\alpha \log n)$ space; by Theorem 8.1, this results in $O(m + n \log^2 n + \alpha \log n)$ space. □

11.2 Implicit $O(2^\alpha)$-Coloring

In this section, we present an implicit $O(2^\alpha)$-coloring algorithm, where α is the current arboricity of the graph, after processing the most recent batch of updates. As defined previously, an *implicit* coloring is maintained by a set of data structures whereby on *queries* of one or more vertices, the data structures return a valid coloring in the induced subgraph of the queried vertices.

To do this, we maintain a batch-dynamic version of the arboricity forest decomposition structure of Henzinger et al. [47]. We

[15] u may pick a random color multiple times, and we consider the palette that is last used by u to pick its final color.

construct these forests using the batch-dynamic Euler tour data structure of Tseng et al. [85]. As in previous sections, we use the framework provided in Section 8. The sequential, dynamic algorithm of Henzinger et al. [47] turns out to be somewhat tricky to parallelize. Specifically, Henzinger et al. [47] prevents cycles in the forests they create by sequentially inserting edges into one of two possible trees. In the batch-dynamic setting, since we are inserting multiple edges simultaneously, we need to run a cycle-breaking algorithm to split the cycles among the trees; such an algorithm is somewhat cumbersome to implement. Instead, we present a simpler version of their algorithm below that is much easier to parallelize, provided an acyclic low out-degree orientation; our simpler algorithm provides the same guarantees.

Our Simplified Dynamic Arboricity Forest Decomposition Structure. We first provide a simplified version of the arboricity forest decomposition structure of [47] here. We are able to simplify the structure since we assume an acyclic orientation algorithm, while the arboricity decomposition structure in [47] can use any orientation algorithm, not necessarily only acyclic ones. We also present some new proofs that our simplified structure still solves the $O(2^{\alpha})$-coloring problem in the same work bounds as the structure presented in [47]. Then, we build on this simplified structure to design our parallel batch-dynamic algorithm.

Provided an σ out-degree orientation, the key idea behind the arboricity forest decomposition structure of [47] is to create 2σ *undirected* forests. However, we show here that σ undirected forests is sufficient for this problem if the out-degree orientation is also acyclic. Below, we present a simpler version of the algorithm using only σ undirected forests via a simple lemma (Lemma 11.5) we prove.

They use two different types of data structures to maintain the forests: the top tree data structures of [2] and an array for each node maintaining which trees contain an outgoing edge of that node. We denote the array for node v by A_v. Furthermore, we denote the i^{th} forest by F_i. The forests maintain the following invariants:

(1) There exists a unique root for each tree in each forest.
(2) For each $l \in \{0, \ldots, \sigma - 1\}$ and each $v \in V$, no forest F_l contains two or more outgoing edges of v.
(3) No forest where $j \geq |N^+(v)|$ contains an outgoing edge of v.
(4) $A_v[i] = 1$ (for $i \in [\sigma]$) if and only if F_i contains an outgoing edge of v. Otherwise, $A_v[i] = 0$.

The forests support the following two operations:

(1) **Insert oriented edge:** A new directed edge (u, v) is inserted into the structure in $O(\log n)$ time. Let $d(u) = |N^+(u)|$ (where $N^+(u)$ is the out-degree of u before the new edge insertion). This is done by inserting the edge into $F_{d(v)}$ and setting $A_u[i] = 1$. The top tree allows this operation to be done in $O(\log n)$ time. The out-degree of v is now $d(v) + 1$ and all invariants remain satisfied.
(2) **Delete oriented edge:** A directed edge (u, v) is deleted from the structure in $O(\log n)$ time. We first find the location of the edge in the forests. This can be done by maintaining pointers from edges to their respective locations in the forests. Let F_i be the forest that contains (u, v). Delete (u, v) from F_i. The top tree allows this operation to be done in $O(\log n)$ time. Then, we find the tree with the largest index that contains an outgoing edge of u. (By our invariants, this should be $F_{d(u)-1}$ where $d(u)$ is the out-degree of u before the edge deletion.) Let e be the outgoing edge of u in $F_{d(u)-1}$. Then, delete e from $F_{d(u)-1}$ and insert e into F_i. The top tree allows us to perform both operations in $O(\log n)$ time.

Any edge flips from (u, v) to (v, u) in our orientation algorithm can be handled by first performing an edge deletion of (u, v) followed by an edge insertion of (v, u) using the algorithms above.

We now prove the correctness of the simplified structure for acyclic orientation algorithms.

Lemma 11.5. *Let σ be the maximum out-degree of our acyclic orientation algorithm. Then, $F_0, \ldots, F_{\sigma-1}$ provides an arboricity decomposition of the graph.*

Proof. This proof relies on proving two parts of the simplified algorithm. First, we need to show that the union of all forests in $F_0, \ldots, F_{\sigma-1}$ gives all of the edges in the input graph. To show this first property, we need only show that no outgoing edge of any vertex $v \in V$ is in any forest F_j where $j \geq d(v)$ and $d(v)$ is the out-degree of vertex v. This directly follows from the invariants. Furthermore, each inserted edge is inserted into at least one of the forests.

Then, we need to show that no cycles exists in any of the forests (i.e., each F_i is properly a forest). To do this, it is sufficient to show that no cycles exist in any F_i. We prove this via contradiction. Suppose that a cycle exists in F_i. By our invariant, this means that no vertex v incident to two edges, (v, w) and (v, u), in the cycle has both (v, w) and (v, u) oriented outwards from v. Otherwise, this would violate the invariant that at most one outgoing edge of v is in any forest. Thus, for every vertex in the cycle, one of the incident edges must be directed out and one directed in, in the orientation produced by the orientation algorithm. This is a contradiction to the acyclicity of the orientation algorithm. □

Batch-Dynamic Algorithm. We implement the batch-dynamic algorithm in the following way. We implement the trees using the batch-dynamic Euler tour trees of Tseng et al. [85]. These trees allow inserts/deletes and distance-to-root operations in $O(|\mathcal{B}| \log n)$ work and $O(\log n)$ depth w.h.p.[16] Furthermore, we maintain a parallel hash table, T, which contains the edges as keys and pointers to the tree containing each edge as the values.

In this algorithm, let our batch of edges to insert into our forests be the set of edge insertions, deletions, and edge flips. To implement BatchFlips, we create two sets of updates per flipped edge: an edge deletion and an edge insertion. Then, following the framework in Section 8, we first process the deletions and then the insertions.

For each oriented deleted edge, we check, in parallel, in the hash table T for the location of each edge (which tree each edge is in). Then, we perform, in parallel, deletions of the edges in the respective parallel Euler tour trees. All of this can be done in $O(|\mathcal{B}| \log^3 n)$ amortized work and $O(\log n)$ depth w.h.p. The work results from performing Euler tree operations on the set of edges in $\mathcal{B}_{ins}$ and $\mathcal{B}_{del}$. There are $O(|\mathcal{B}| \log^2 n)$ updates in these batches and each Euler tree operation requires $O(\log n)$ work; thus, our total work is $O(|\mathcal{B}| \log^3 n)$. We can perform all updates to our Euler trees in

[16] The high probability bound directly follows from the high probability bounds of parallel skip-lists.

parallel, hence, the total depth is the depth of performing these updates, $O(\log n)$ w.h.p.

For each vertex, we maintain the number of edges deleted from its trees as well as the trees from which edges are deleted. This can be done in $O(|\mathcal{B}| \log^2 n)$ amortized work and $O(\log n)$ depth w.h.p. Finally, we find, in parallel, the X_i outgoing edges of v_i in the last X_i trees that contain an outgoing edge of v_i, where X_i is the number of edges that were deleted from v_i's trees. In parallel, we arbitrarily pick a unique slot for each edge and assign it to to its respective empty slots in the trees. This last step can also be done in $O(|\mathcal{B}| \log^3 n)$ amortized work and $O(\log n)$ depth w.h.p.

For the insertion edges, we first sort the edges by their outgoing endpoint. Then, we determine how many edges we are inserting in each v_i's trees by doing a parallel count. Then, finally, in parallel, we insert each v_i's edge into the next X_i empty trees where X_i is the number of edge inserts that are oriented out from v_i. All of this requires $O(|\mathcal{B}| \log^3 n)$ amortized work and $O(\log n)$ depth.

The correctness of our procedure follows from Lemma 11.5. Altogether, we obtain the following theorem of our batch-dynamic algorithm implicit $O(2^\alpha)$-coloring, using Theorem 8.1.

Theorem 11.6. *For a batch $\mathcal{B}$, our batch-dynamic implicit coloring algorithm provides a $O(2^\alpha)$-coloring in $O(|\mathcal{B}| \log^3 n)$ amortized work and $O(\log^2 n)$ depth w.h.p. for updates, and $O(Q\alpha \log n)$ work and $O(\log n)$ depth, w.h.p., for Q queries, using $O(n \log^2 n + m)$ space.*

Proof. The work and depth follow from our above arguments and Theorem 8.1. For queries, we parallelize the algorithm of [47]. For a set of Q vertices, for each vertex, we find the set of forests $[1, \ldots, d(v)]$ where $d(v) = O(\alpha)$, containing each of the outgoing edges of v. As in [47], we let $p_v(i)$ be the parity of the distance for the i-th Euler tree. Then, in parallel, we determine the distance of v from the root of the Euler tree in each of these forests. If the distance is odd, we assign $p_v(i) = 1$ and $p_v(i) = 0$ otherwise. The color given to v is then $(p_v(1), \ldots, p_v(d(v))) \in \{0,1\}^{O(\alpha)}$. Querying the Euler trees require $O(\log n)$ work per tree query. We have $O(Q\alpha)$ total queries, resulting in $O(Q\alpha \log n)$ total work. Then, processing all queries simultaneously requires $O(\log n)$ depth w.h.p.

Finally, the extra space required is the space to store the extra Euler trees and the hash table T. T uses $O(m)$ space. All of the Euler trees store $O(m)$ edges; thus, the total additional space used is $O(m)$. □

12 CONCLUSION

We design the first shared-memory, multi-core parallel batch-dynamic level data structure that returns a $(2+\varepsilon)$-approximation for the k-core decomposition problem, drawing inspiration from the sequential level data structures of Bhattacharya et al. [13] and Henzinger et al. [47] which were used for dynamic densest subgraphs and dynamic low out-degree orientation, respectively. Our algorithm achieves $O(\log^2 m)$ amortized work and has $O(\log^2 m \log\log m)$ depth. We also present a proof of the $(2+\varepsilon)$-factor of approximation for our data structure, a new proof that is also applicable (with a simple change) to the original sequential level data structures of Bhattacharya et al. [13] and Henzinger et al. [47].

In addition to our batch-dynamic k-core decomposition results, we also give a batch-dynamic algorithm for maintaining an $O(\alpha)$ out-degree orientation, where α is the *current* arboricity of the graph. We demonstrate the usefulness of our low out-degree orientation algorithm by presenting a new framework to formally study batch-dynamic algorithms in bounded-arboricity graphs. Our framework obtains new provably-efficient parallel batch-dynamic algorithms for maximal matching, clique counting, and vertex coloring.

We perform extensive experimentation of our parallel batch-dynamic k-core decomposition algorithm on large real-world data sets that show that our PLDS is not only theoretically but also practically efficient. Our experiments tested error vs. runtime, batch size vs. runtime, number of hyper-threads vs. runtime, and space vs. error. We also tested the sensitivity of our implementation to the various tunable parameters of our algorithm. Finally, we tested our algorithm against six other benchmarks on 11 real-world graphs, including graphs orders of magnitude larger than previously studied by other dynamic algorithms. We see an improvement in performance against all other benchmarks in our experiments. Specifically, we achieve speedups of up to 114.52× against the best parallel implementation, up to 544.22× against the best approximate sequential algorithm, and up to 723.72× against the best exact sequential algorithm. Such speed-ups exceed the expected speed-up gained from parallelism alone (since we only use 60 hyper-threads) and are also due to the theoretical improvements of our algorithm as well as our choice of heuristic optimizations.

An interesting open problem is to design a parallel batch-dynamic algorithm that is space-efficient (uses linear space), without incurring additional costs in depth.

ACKNOWLEDGEMENTS

This research is supported by NSF GRFP #1122374, DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, Google Research Scholar Award, FinTech@CSAIL Initiative, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-Based Community Search: A Truss-Equivalence Based Indexing Approach. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1298–1309.
[2] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. 2005. Maintaining Information in Fully Dynamic Trees with Top Trees. *ACM Trans. Algorithms* 1, 2 (oct 2005), 243–264.
[3] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large Scale Networks Fingerprinting and Visualization Using the K-Core Decomposition. In *International Conference on Neural Information Processing Systems*.
[4] Altaf Amin, Yoko Shinbo, Kenji Mihara, Ken Kurokawa, and Shigehiko Kanaya. 2006. Development and implementation of an algorithm for detection of protein complexes in large interaction networks. *BMC bioinformatics* 7 (02 2006), 207.
[5] Richard Anderson and Ernst W. Mayr. 1984. *A* P*-complete Problem and Approximations to It.* Technical Report.
[6] Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. 2016. Distributed K-Core Decomposition and Maintenance in Large Dynamic Graphs. In *ACM International Conference on Distributed and Event-Based Systems*. 161–168.
[7] Gary D. Bader and Christopher WV Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics* 4, 1 (Jan. 2003), 2.

[8] Luis Barba, Jean Cardinal, Matias Korman, Stefan Langerman, André van Renssen, Marcel Roeloffzen, and Sander Verdonschot. 2019. Dynamic Graph Coloring. *Algorithmica* 81, 4 (2019), 1319–1341.

[9] Ioana O. Bercea and Guy Even. 2021. Upper tail analysis of bucket sort and random tries. *Theor. Comput. Sci.* 895 (2021), 55–67.

[10] Edvin Berglin and Gerth Stølting Brodal. 2020. A Simple Greedy Algorithm for Dynamic Graph Orientation. *Algorithmica* 82, 2 (feb 2020), 245–259.

[11] Aaron Bernstein and Cliff Stein. 2015. Fully Dynamic Matching in Bipartite Graphs. In *International Colloquium on Automata, Languages, and Programming*, Vol. 9134. 167–179.

[12] Aaron Bernstein and Cliff Stein. 2016. Faster Fully Dynamic Matchings with Small Approximation Ratios. In *ACM-SIAM Symposium on Discrete Algorithms*. 692–711.

[13] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. 2015. Space- and Time-Efficient Algorithm for Maintaining Dense Subgraphs on One-Pass Dynamic Streams. In *ACM Symposium on Theory of Computing (STOC)*. 173–182.

[14] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. 2013. Efficient Parallel and External Matching. In *International Conference on Parallel Processing (Euro-Par)*. 659–670.

[15] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. Brief Announcement: ParlayLib – A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symp. on Parallel Alg. (SPAA)*.

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

[17] Guy E. Blelloch and Bruce M. Maggs. 1996. Parallel Algorithms. *Commun. ACM* 39 (1996), 85–97.

[18] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core Decomposition of Uncertain Graphs. In *ACM SIGKDD*. 1316–1325.

[19] Gerth Stølting Brodal and Rolf Fagerberg. 1999. Dynamic Representations of Sparse Graphs. In *Proc. 6th International Workshop on Algorithms and Data Structures (WADS)*. 342–351.

[20] Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt, and Eran Shir. 2007. A model of Internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences* 104, 27 (2007), 11150–11154.

[21] T.-H. Hubert Chan, Mauro Sozio, and Bintao Sun. 2021. Distributed approximate k-core decomposition and min-max edge orientation: Breaking the diameter barrier. *J. Parallel Distributed Comput.* 147 (2021), 87–99.

[22] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (Feb. 1985), 210–223.

[23] Deming Chu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang, Yinglong Xia, and Chenyi Zhang. 2020. Finding the Best k in Core Decomposition: A Time and Space Optimal Solution. In *IEEE ICDE*. 685–696.

[24] Martino Ciaperoni, Edoardo Galimberti, Francesco Bonchi, Ciro Cattuto, Francesco Gullo, and Alain Barrat. 2020. Relevance of temporal cores for epidemic spread in temporal networks. *Scientific Reports* 10, 1 (July 2020).

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3. ed.).* MIT Press.

[26] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. 2008. *Implementation Challenge for Shortest Paths.* 395–398.

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

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

[29] Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. 2021. Parallel Batch-Dynamic k-Clique Counting. In *2nd Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 129–143.

[30] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2009. Extraction and Classification of Dense Implicit Communities in the Web Graph. *ACM Trans. Web* 3, 2, Article 7 (April 2009), 36 pages.

[31] Zdenek Dvorák and Vojtech Tuma. 2013. A Dynamic Data Structure for Counting Subgraphs in Sparse Graphs. In *International Workshop on Algorithms and Data Structures (WADS)*. 304–315.

[32] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. 2019. Efficient Computation of Probabilistic Core Decomposition at Web-Scale. In *International Conference on Extending Database Technology*. 325–336.

[33] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. 2018. Parallel and Streaming Algorithms for K-Core Decomposition. In *International Conference on Machine Learning*. 1397–1406.

[34] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective Community Search over Large Spatial Graphs. *Proc. VLDB Endow.* 10, 6 (Feb. 2017), 709–720.

[35] Manuela Fischer. 2020. Improved deterministic distributed matching via rounding. *Distributed Comput.* 33, 3-4 (2020), 279–291.

[36] Manuela Fischer and Andreas Noever. 2018. Tight Analysis of Parallel Randomized Greedy MIS. In *ACM-SIAM Symposium on Discrete Algorithms*. 2152–2160.

[37] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. 2003. Fully dynamic shortest paths in digraphs with arbitrary arc weights. *J. Algorithms* 49, 1 (2003), 86–113.

[38] Kasimir Gabert, Ali Pinar, and Ümit V. Çatalyürek. 2021. Shared-Memory Scalable k-Core Maintenance on Dynamic Graphs and Hypergraphs. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 998–1007.

[39] Edoardo Galimberti, Francesco Bonchi, Francesco Gullo, and Tommaso Lanciano. 2020. Core Decomposition in Multilayer Networks: Theory, Algorithms, and Applications. *ACM Trans. Knowl. Discov. Data* 14, 1, Article 11 (Jan. 2020).

[40] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrović. 2019. Improved Parallel Algorithms for Density-Based Network Clustering. In *International Conference on Machine Learning*. 2201–2210.

[41] Christos Giatsidis, Fragkiskos D. Malliaros, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2014. CoreCluster: A Degeneracy Based Graph Clustering Framework. In *AAAI*. 44–50.

[42] J. Gil, Y. Matias, and U. Vishkin. 1991. Towards a theory of nearly constant time parallel algorithms. In *IEEE FOCS*. 698–710.

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

[44] Kathrin Hanauer, Monika Henzinger, and Qi Cheng Hua. 2021. Fully Dynamic Four-Vertex Subgraph Counting. *CoRR* abs/2106.15524 (2021).

[45] Meng He, Ganggui Tang, and Norbert Zeh. 2014. Orienting Dynamic Graphs, with Applications to Maximal Matchings and Adjacency Queries. In *International Symposium on Algorithms and Computation*. 128–140.

[46] John Healy, Jeannette Janssen, Evangelos Milios, and William Aiello. 2007. Characterization of Graphs Using Degree Cores. In *International Workshop on Algorithms and Models for the Web-Graph (WAW)*. 137–148.

[47] Monika Henzinger, Stefan Neumann, and Andreas Wiese. 2020. Explicit and Implicit Dynamic Coloring of Graphs with Bounded Arboricity. *CoRR* abs/2002.10142 (2020).

[48] Q. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen. 2020. Faster Parallel Core Maintenance Algorithms in Dynamic Graphs. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2020), 1287–1300.

[49] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying K-Truss Community in Large and Dynamic Graphs. In *ACM SIGMOD International Conference on Management of Data*. 1311–1322.

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

[51] H. Jin, N. Wang, D. Yu, Q. Hua, X. Shi, and X. Xie. 2018. Core Maintenance in Dynamic Graphs: A Parallel Approach Based on Matching. *IEEE Transactions on Parallel and Distributed Systems* 29, 11 (2018), 2416–2428.

[52] H. Kabir and K. Madduri. 2017. Parallel k-Core Decomposition on Multicore Platforms. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1482–1491.

[53] Haim Kaplan and Shay Solomon. 2018. Dynamic Representations of Sparse Distributed Networks: A Locality-Sensitive Approach. In *ACM Symposium on Parallelism in Algorithms and Architectures*. 33–42.

[54] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-Core Decomposition of Large Networks on a Single PC. *Proc. VLDB Endow.* 9, 1 (Sept. 2015), 13–23.

[55] Maksim Kitsak, Lazaros K. Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H. Eugene Stanley, and Hernán A. Makse. 2010. Identification of influential spreaders in complex networks. *Nature Physics* 6, 11 (Nov. 2010), 888–893.

[56] Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. 2014. Orienting Fully Dynamic Graphs with Worst-Case Time Bounds. In *International Colloquium on Automata, Languages and Programming (ICALP)*. 532–543.

[57] Lukasz Kowalik. 2007. Adjacency queries in dynamic sparse graphs. *Inf. Process. Lett.* 102, 5 (2007), 191–195.

[58] Lukasz Kowalik. 2010. Fast 3-coloring Triangle-Free Planar Graphs. *Algorithmica* 58, 3 (2010), 770–789.

[59] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *International Conference on World Wide Web*. 591–600.

[60] Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. 2010. A Survey of Algorithms for Dense Subgraph Discovery. In *Managing and Mining Graph Data*. 303–336.

[61] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[62] Conggai Li, Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2019. Efficient Progressive Minimum K-Core Search. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 362–375.

[63] R. Li, J. Yu, and R. Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Transactions on Knowledge & Data Engineering* 26, 10 (oct 2014), 2453–2465.

[64] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Transactions on Knowledge and Data Engineering*

26, 10 (2014), 2453–2465.
[65] Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. 2021. Hierarchical Core Maintenance on Large Dynamic Graphs. *Proc. VLDB Endow.* 14, 5 (2021), 757–770.
[66] Ying Liu, Ming Tang, Tao Zhou, and Younghae Do. 2015. Core-like groups result in invalidation of identifying super-spreader by k-shell decomposition. *Scientific Reports* 5 (May 2015), 9602–9602.
[67] Shangqi Lu and Yufei Tao. 2021. Towards Optimal Dynamic Indexes for Approximate (and Exact) Triangle Counting. In *24th International Conference on Database Theory (ICDT 2021)*, Vol. 186. 6:1–6:23.
[68] Qi Luo, Dongxiao Yu, Zhipeng Cai, Xuemin Lin, and Xiuzhen Cheng. 2021. Hypercore Maintenance in Dynamic Hypergraphs. In *IEEE ICDE*. 2051–2056.
[69] Qi Luo, Dongxiao Yu, Feng Li, Zhenhao Dou, Zhipeng Cai, Jiguo Yu, and Xiuzhen Cheng. 2019. Distributed Core Decomposition in Probabilistic Graphs. In *Computational Data and Social Networks*. 16–32.
[70] Qi Luo, Dongxiao Yu, Hao Sheng, Jiguo Yu, and Xiuzhen Cheng. 2021. Distributed Algorithm for Truss Maintenance in Dynamic Graphs. In *Parallel and Distributed Computing, Applications and Technologies*. 104–115.
[71] Fragkiskos D. Malliaros, Maria-Evgenia G. Rossi, and Michalis Vazirgiannis. 2016. Locating influential nodes in complex networks. *Scientific Reports* 6, 1 (2016).
[72] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and Clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (July 1983), 417–427.
[73] Sourav Medya, Tianyi Ma, Arlei Silva, and Ambuj Singh. 2020. A Game Theoretic Approach For K-Core Minimization. In *19th International Conference on Autonomous Agents and MultiAgent Systems*. 1922–1924.
[74] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. 2015. Scalable Large Near-Clique Detection in Large-Scale Networks via Sampling. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 815–824.
[75] Ofer Neiman and Shay Solomon. 2015. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. on Alg. (TALG)* 12, 1 (2015), 1–15.
[76] Merav Parter, David Peleg, and Shay Solomon. 2016. Local-on-Average Distributed Tasks. In *ACM-SIAM Symposium on Discrete Algorithms*. 220–239.
[77] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. 4292–4293. http://networkrepository.com
[78] Ahmet Erdem Sariyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal* 25, 3 (2016), 425–447.
[79] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for K-Core Decomposition. *Proc. VLDB Endow.* 6, 6 (April 2013), 433–444.
[80] Saurabh Sawlani and Junxing Wang. 2020. Near-Optimal Fully Dynamic Densest Subgraph. In *ACM SIGACT Symposium on Theory of Computing*. 181–193.
[81] Julian Shun and Guy E Blelloch. 2014. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 96–107.
[82] Shay Solomon and Nicole Wein. 2020. Improved Dynamic Graph Coloring. *ACM Trans. on Alg. (TALG)* 16, 3, Article 41 (June 2020).
[83] Bintao Sun, T.-H. Hubert Chan, and Mauro Sozio. 2020. Fully Dynamic Approximate K-Core Decomposition in Hypergraphs. *ACM Trans. Knowl. Discov. Data* 14, 4, Article 39 (May 2020).
[84] Bintao Sun, T-H. Hubert Chan, and Mauro Sozio. 2020. *Fully Dynamic Approximate k-Core Decomposition in Hypergraphs.* https://github.com/btsun/DynHyperCoreDecomp
[85] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. 2019. Batch-Parallel Euler Tour Trees. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*. 92–106.
[86] Kai Wang, Xin Cao, Xuemin Lin, Wenjie Zhang, and Lu Qin. 2018. Efficient Computing of Radius-Bounded k-Cores. In *IEEE ICDE*. 233–244.
[87] Na Wang, Dongxiao Yu, Hai Jin, Chen Qian, Xia Xie, and Qiang-Sheng Hua. 2017. Parallel Algorithm for Core Maintenance in Dynamic Graphs. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2366–2371.
[88] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. Yu. 2019. I/O Efficient Core Graph Decomposition: Application to Degeneracy Ordering. *IEEE Transactions on Knowledge & Data Engineering* 31, 01 (Jan 2019), 75–90.
[89] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (Jan. 2015), 181–213.
[90] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2017. When Engagement Meets Similarity: Efficient (k,r)-Core Computation on Social Networks. *Proc. VLDB Endow.* 10, 10 (June 2017), 998–1009.
[91] Haohua Zhang, Hai Zhao, Wei Cai, Jie Liu, and Wanlei Zhou. 2010. Using the k-core decomposition to analyze the static structure of large-scale software systems. *J. Supercomput.* 53, 2 (2010), 352–369.
[92] Y. Zhang, J. Yu, Y. Zhang, and L. Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *IEEE ICDE*. 337–348.
[93] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and Efficiency of Truss Maintenance in Evolving Graphs. In *ACM SIGMOD International Conference on Management of Data*. 1024–1041.
[94] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *IEEE ICDE*. 337–348.