Parallel Batch-Dynamic k-Core Decomposition and Low Out-Degree Orientation

Quanquan C. Liu
MIT CSAIL
quanquan@mit.edu

Jessica Shi
MIT CSAIL
jeshi@mit.edu

Shangdi Yu
MIT CSAIL
shangdiy@mit.edu

Laxman Dhulipala
MIT CSAIL
laxman@mit.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Abstract

Maintaining a k-core decomposition quickly in a dynamic graph is an important problem in many applications, including social network analytics, graph visualization, centrality measure computations, and community detection algorithms. A k-core decomposition is a well-established tool that partitions vertices in the graph into layers with less connected vertices in the outer layers and more well-connected vertices in the inner layers. The main challenge for designing efficient k-core decomposition algorithms is that a single change to the graph can cause the decomposition to change significantly.

We present the first parallel batch-dynamic algorithm for maintaining an approximate k-core decomposition that is efficient in both theory and practice. Given an initial graph with m edges, and a batch of $\mathcal{B}$ updates, our algorithm maintains a $(2+\delta)$-approximation of the coreness values for all vertices (for any constant $\delta > 0$) in $O(\mathcal{B}\log^2 m)$ amortized work and $O(\log^2 m \log\log m)$ depth (parallel time) with high probability. Our algorithm also maintains a low out-degree orientation of the graph in the same bounds. We implemented and experimentally evaluated our 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 a $YYx - ZZx$ speedup against the best multicore implementation and $SSx - ZZx$ speedup against the best sequential algorithm, obtaining results for graphs that are orders-of-magnitude larger than those used in previous studies.Laxman: need to fill in these numbers

In addition, we present the first approximate static k-core algorithm with linear work and polylogarithmic depth. We show that on a 30-core machine with two-way hyper-threading, our implementation achieves up to a 3.90× speedup in the static case over the previous state-of-the-art parallel algorithm.

PVLDB Reference Format:
Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. Parallel Batch-Dynamic k-Core Decomposition and Low Out-Degree Orientation. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS. Julian: fill

1 Introduction

Quanquan: We need to explain the difference between connected and not connected k-cores and why the version that doesn't require the cores to be connected is still useful. 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, which induces a natural hierarchical clustering on the input graph. The k-core decomposition of a graph and its related concepts of arboricity, low out-degree orientation, and densest subgraph are widely used in machine learning applications, including community detection [6, 35], analyzing social network dynamics [8], visualizing large-scale complex networks [1], analyzing protein networks [2], approximating network centrality measures [24], and many more. In addition, low out-degree orientations can lead to efficient graph algorithms for sparse graphs.Laxman: should we add some citations here, e.g., to the maximal matching and graph coloring applications?

Many well-known algorithms for k-core decomposition are inherently sequential. The classic algorithm for finding such a decomposition is to iteratively select and remove all vertices v with smallest degree from the graph until the graph is empty. Unfortunately, the length of the sequential dependencies (the *depth*) of such a process can be $\Omega(n)$ given a graph with n vertices. Due to the potentially large depth, this algorithm cannot fully take advantage of parallelism on modern machines, and can therefore be too costly to run on large graphs. As k-core decomposition is a P-complete problem [3], it is unlikely that there is a parallel algorithm with polylogarithmic depth for this problem. To obtain parallel methods for this problem, we relax the condition of obtaining an *exact* k-core decomposition to one of obtaining a close *approximate* k-core decomposition.

Parallel Static Approximate k-core. In this paper, we present a novel parallel static approximate k-core decomposition algorithm based on a relaxed version of the peeling algorithm of [14] that achieves both optimal linear work and has polylogarithmic depth. In contrast to existing parallel approximate k-core algorithms [18, 19] which are designed for distributed memory (in the MPC model), our algorithm is designed for shared-memory multicore machines,

which have been shown to be able to process the largest graphs with hundreds of billions of edges efficiently [15, 33]. To the best of our knowledge, ours is the first parallel algorithm to achieve work-efficiency and polylogarithmic depth. Our bounds are summarized in Theorem 4.2.

Theorem 1.1. *For a graph with m edges,[1] for any constant $\delta > 0$, there is an algorithm that finds an $(2+\delta)$-approximate k-core decomposition in $O(m)$ expected work and $O(\log^3 n)$ depth with high probability,[2] using $O(m)$ space.*

Parallel Batch-Dynamic Approximate k-core. Because of the rapidly changing nature of today's graphs, it is inefficient to re-compute the k-core decomposition of the graph from scratch on every update. For this purpose, dynamic k-core decompositions are especially useful. In this paper, we design an approximate k-core algorithm with strong theoretical guarantees for the *batch-dynamic* setting, where updates to the graph are provided in *batches* that can be processed in parallel. Our batch-dynamic algorithm efficiently maintains the k-core decomposition in parallel given batches of updates, and uses a level data structure inspired by [7, 25] to maintain a partition of the vertices satisfying certain degree properties. Our algorithm takes $O(\log^2 m)$ amortized work per update, matching the best sequential algorithm [42], while achieving polylogarithmic depth at the same time. The bounds for our algorithms are as follows.

Theorem 1.2. *Provided an input graph with m edges, and a batch of $\mathcal{B}$ updates, our algorithm maintains a $(2+\delta)$-approximation of the coreness values for all vertices (for any constant $\delta > 0$) in $O(\mathcal{B}\log^2 m)$ amortized work and $O(\log^2 m \log\log m)$ depth with high probability, using $O(n\log^2 m + m)$ space.*

Moreover, our parallel batch-dynamic algorithm can be used to maintain low out-degree orientations and approximate densest subgraphs in the same complexity bounds.

Experimental Evaluation. In addition to our theoretical contributions, we also provide optimized multicore implementations of both our static and batch-dynamic 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 [14], and achieves a 14.76–36.07x self-relative speedup. We show that our parallel batch-dynamic k-core algorithm achieves a 11.15–57.43x speedup over the state-of-the-art sequential k-core algorithm [42], while achieving comparable accuracy. Furthermore, our batch-dynamic algorithm is able to outperform our static approximate k-core algorithm by up to 7x on small batch sizes. We also achieve up to a 3.4x speedup compared to the reported numbers of the state-of-the-art parallel batch-dynamic k-core algorithm of [26].

Related Work. Numerous parallel k-core algorithms exist for the static setting (e.g., [12, 14, 16, 29, 34, 36, 37, 44]). The algorithms of [18, 19] are approximate algorithms that achieve a logarithmic or sub-logarithmic number of rounds in the distributed MPC model. However, the local computation in their algorithms is free in the MPC model, but have linear depth in the shared-memory setting, worse than the polylogarithmic depth bound that we achieve. [18, 19] do not report running times in their paper, and hence it is difficult to compare actual performance. However, we are able to process graphs used in their papers using just a single multicore machine.

There have been dynamic k-core algorithms that have been developed for the sequential [32, 39, 42, 46, 47] and parallel [4, 26, 28] settings. However, to the best of our knowledge, there are no parallel dynamic k-core algorithms with polylogarithmic depth, which our algorithm achieves.

2 Preliminaries

Graph Definitions. For an unweighted, undirected graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, let $n = |V|$, the number of vertices, and $m = |E|$, the number of edges. We denote the set of neighbors of vertex v as $N(v)$. For a subset of vertices, $S \subseteq V$, $N(S) = \bigcup_{v\in S} N(v)$. The ***arboricity***, α, of G is the minimum number of forests into which E can be partitioned, and is upper bounded by $O(\sqrt{m})$ [10].

Definition 2.1 (Low Out-degree Orientation)**.** *A **low out-degree orientation** of an undirected graph is an orientation of its edges such that the out-degree of every vertex is $O(\alpha)$.*

The k-***core*** of a graph is the maximal subgraph H of G such that the induced degree of every vertex in H is at least 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.*

The above defines an *exact* k-core decomposition. A $(1+\delta)$-*approximate* k-core decomposition is defined as follows. Julian: later we use $(2+\delta)$ approximate. so maybe we should just say c-approximate here instead of $(1+\delta)$

Definition 2.3 ($(1+\delta)$-Approximate k-Core Decomposition)**.** *A $(1+\delta)$-**approximate** k-**core decomposition** is a partition of vertices into layers such that a vertex v is in layer k if its degree, d, in the induced subgraph consisting of all vertices in layer k satisfies $\frac{k}{(1+\delta)} \le d \le (1+\delta)k$. Laxman: Shouldn't the definition say "induced subgraph consisting of all all vertices in layers k and above" if the layers are a partition of V? Julian: i think the next sentence should be separate, since it is unclear how $\hat{k}(v)$ is related to approx k-core, and it does not necessarily have to satisfy the error bounds above (although we would like it to). $\hat{k}(v)$ denotes the estimate of k's coreness.*

Fig. 1 shows a k-core decomposition and a (3/2)-approximate k-core decomposition.

Definition 2.4 (Densest Subgraph)**.** *A **densest subgraph** S of a graph $G = (V, E)$ has density $|E(S)|/|S| = \max_{S'\subseteq V}(|E(S')|/|S'|)$. Julian: check if we need thisQuanquan: This is used in the proofs and we mention it in the intro also I think that we obtain a result for this.*

Model Definitions. We analyze the theoretical efficiency of our parallel algorithms in the *work-depth model*. The work-depth

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

[2] ***With high probability (whp)*** is defined as with probability at least $1 - 1/n^c$ for any $c \ge 1$.

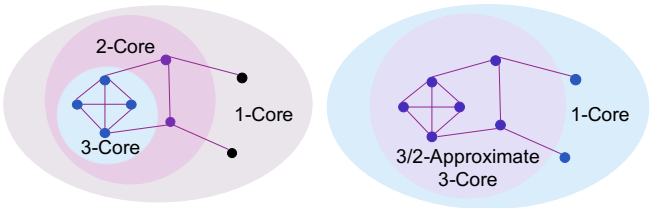

Figure 1: Exact k-core decomposition (left) and $(3/2)$-approximate k-core decomposition (right).

model is a fundamental tool in analyzing parallel algorithms, e.g., see [9, 15, 22, 43, 45] for a sample of recent practical uses of this model. The work-depth model is defined in terms of two complexity measures ***work*** and ***depth***, which are standard measures for analyzing shared-memory algorithms [11, 27]. The ***work*** of an algorithm is the total number of operations executed by the algorithm. The ***depth*** is the longest chain of sequential dependencies in the algorithm. 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.

Our parallel algorithms operate in the batch-dynamic setting. A ***batch-dynamic*** algorithm processes updates (vertex or edge insertions/deletions) in batches 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.

Problem Definition. 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 $(1+\delta)$-approximate k-core decomposition (for any constant $\delta > 0$) 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 we have a $(1+\delta)$-approximate k-core decomposition on G_i; then, for $\mathcal{B}_{i+1}$, our goal is to efficiently find a $(1+\delta)$-approximate k-core decomposition of $G_{i+1} = \left(V, (E_i \cup E^{i+1}_{insert}) \setminus E^{i+1}_{delete}\right)$. Julian: next sentence isn't part of definition and can be dropped In order to efficiently do this, we do not perform unnecessary computation between batches.

All notations used in our paper are summarized in Table 1.

3 Batch-Dynamic $(2+\delta)$-Approximate k-Core Decomposition

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

3.1 Algorithm Overview

We provide a *parallel level data structure (PLDS)* that maintains a $(2+\delta)$-approximate k-core decomposition and low out-degree orientation that builds off the sequential level data structures (LDS) of [7, 25]. Our algorithm achieves $O(\log^2 m)$ amortized work per update and $O(\log^2 m \log\log m)$ depth *whp*. In our full paper Julian: remember to add citation to full paper, we also present a deterministic version of our algorithm that achieves the same work bound with $O(\log^3 m)$ depth. Our data structure can also handle batches of vertex insertions/deletions, which we discuss in the full paper.

As in the case of [25], our data structure can handle *changing arboricity* that is not known a priori to the algorithm. This means

Table 1: Table of notations used in the paper.

Notation	Definition
$G = (V, E)$	undirected/unweighted graph
n, m	number of vertices, edges, resp.
α	current arboricity 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
ℓ	a level (starting with level $\ell = 0$)
$\ell(v)$	current level of vertex v
g_i	set of levels in group i (starting with g_0)
V_ℓ	set of vertices in level ℓ
Z_ℓ	set of vertices in levels $\geq \ell$
$g(v)$	*group number* of vertex v
$gn(\ell)$	index i where level $\ell \in g_i$
$k(v)$	coreness of v
$\hat{k}(v)$	estimate of the coreness of v
$\mathrm{up}(v)$	*up-degree* of v
$\mathrm{up}^*(v)$	*up*-degree* of v
λ	a constant where $\lambda > 3$
ε	a constant where $\varepsilon > 0$

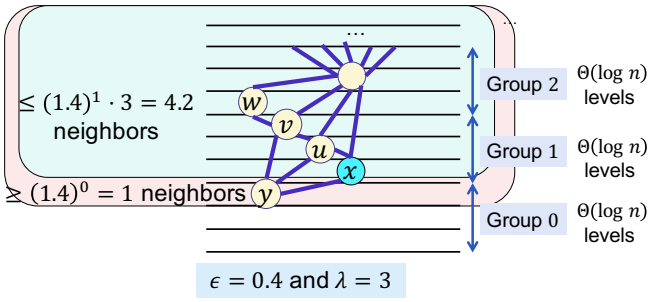

Figure 2: Example of invariants maintained by the PLDS for $\epsilon = 0.4$ and $\lambda = 3$.

that our data structure can handle the case when the arboricity α of the underlying graph changes throughout the execution of batches of updates and successfully maintains approximations of the coreness of each node with respect to the *current* arboricity.

The level data structure consists of a partition of the vertices of the graph into $K = O(\log^2 m)$ levels. [3] The levels are partitioned into equal-sized *groups* of consecutive levels. 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 up or down. Using information about the assigned level of a vertex, we obtain a $(2+\delta)$-approximation on the coreness of the vertex.

Fig. 2 shows an illustration of this data structure. There are $\Theta(\log n)$ groups, each with $\Theta(\log n)$ levels (labeled on the right). Each vertex is in exactly one level of the structure and moves up and down by some movement rules. We describe our parallel level data structure in more detail in Section 3.3.

3.2 Low Out-Degree Orientations and LDS

Low out-degree orientations have been used in many efficient static and dynamic algorithms for graphs (e.g., [5, 17, 21, 41]). Our work is based on the sequential level data structures (LDS) of [7, 25], which

[3] A more refined analysis shows that we only need $O(\log \Delta \log m)$ levels, where Δ is the current maximum degree in the graph.

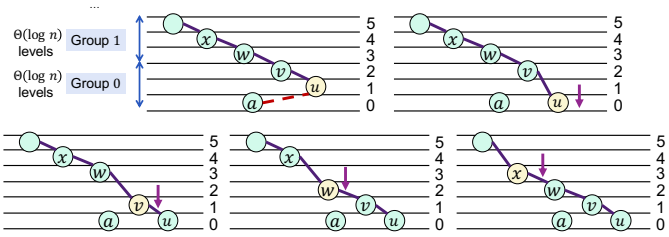

Figure 3: Example of a cascade of vertex movements caused by edge deletion.

maintain a low out-degree orientation under dynamic updates. Within their LDS, a vertex moves up or down levels one by one, meaning that 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.

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 (illustrated in Example 3.1). Then, if the vertex moves again, the same cascade of movements may occur.

Example 3.1. In Fig. 3, suppose a vertex u at level 1 is incident to an edge deletion (a, u) (dashed edge in red) and must move down one level. This in turn could cause another one of its neighbors v at level 2 to move down one level, leading to a cascade of vertices which must move down (w at level 3, then x at level 4, etc.). If u could move down one additional level, it may cause another cascade of movements.

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. We must also be careful not to perform unnecessary work; one example is a vertex that moves up due to edge insertions but then later moves down due to edge deletions (from the same batch).

Thus, our PLDS algorithm solves several challenges posed by the sequential algorithm. Provided a batch $\mathcal{B}$ of edge insertions and deletions: **(1)** our algorithm processes the levels in a careful order that yields provably low depth for batches of updates; **(2)** our algorithm processes insertions and deletions in separate batches to avoid excess work; **(3)** our insertion algorithm processes vertices on each level at most once, which is key to bounding the depth–after a vertex moves up from level i, no future step in the algorithm requires a vertex to move up from level i; and **(4)** our deletion algorithm moves vertices to their final level in one step. In other words, a vertex moves at most once in a batch of deletions before a new batch arrives.

3.3 Detailed PLDS Algorithm

As mentioned previously, the vertices of the input graph $G = (V, E)$ 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 $\varepsilon > 0$, the levels are partitioned into ***groups*** $g_0, \ldots, g_{\lceil \log_{(1+\varepsilon)} m \rceil}$, where each group contains $\lceil \log_{(1+\varepsilon)} m \rceil$ consecutive levels. Each $\ell \in \left[i\lceil \log_{(1+\varepsilon)} m \rceil, (i+1)\lceil \log_{(1+\varepsilon)} m \rceil - 1 \right]$ is a level in group i. Our data structure consists of $K = O(\log^2 m)$ levels. The PLDS satisfies the following invariants, which also govern how the data structure is maintained. The invariants assume a given constant $\varepsilon > 0$ and a constant $\lambda > 0$. [4] The below invariants follow invariants defined in [7, 25].

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

All vertices with no neighbors are placed in level 0. An example partitioning of vertices and the maintained invariants is shown in Example 3.2.

Example 3.2. In Fig. 2, where $\epsilon = 0.4$ and $\lambda = 3$, vertex x (blue) is on level 3 in group 1. This means that by Invariant 1, it has at most $(2 + 3/\lambda)(1+\epsilon)^1 = 4.2$ neighbors at level 3 and above. In Fig. 2, the green box highlights all levels ≥ 3; indeed x has 2 neighbors in levels ≥ 3, satisfying Invariant 1. By Invariant 2, x has more than $(1+\epsilon)^0 = 1$ neighbor in levels ≥ 2 (note that level 2 is in group 0). In the example, x has 3 neighbors, satisfying the invariant (levels ≥ 2 are highlighted by the pink box).

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 group g where $\ell(v) \in g$. We define the ***up-degree***, $\mathsf{up}(v)$, of a vertex v to be the number of its neighbors in $Z_{\ell(v)}$ (up-neighbors), and ***up*-degree***, $\mathsf{up}^*(v)$, 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 that our data structure maintains. Lastly, the ***desire-level*** $\mathsf{dl}(v)$ of a vertex v is the *closest level to the current level of the vertex* that satisfies both Invariant 1 and Invariant 2.

Definition 3.3 (Desire-level). *The* desire-level, $\mathsf{dl}(v)$, *of vertex v is the level ℓ' which minimizes $|\ell(v) - \ell'|$, and where $\mathsf{up}^*(v) \geq (1+\varepsilon)^{i'}$ and $\mathsf{up}(v) \leq (2 + 3/\lambda)(1+\varepsilon)^i$ 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 $\ell(v)$ where both Invariant 1 and Invariant 2 are satisfied.*

We show that the invariants above 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*, during which the invariants may be violated. The main update procedure is shown in Algorithm 1. It separates the updates into insertions and deletions (Line 2–Line 4), and then calls RebalanceInsertions (Line 5: Algorithm 2) and RebalanceDeletions (Line 6: Algorithm 3). We make note of two crucial observations that we prove in the full paper: when processing a batch of insertions, Invariant 2 is never violated; and similarly, when processing a batch of deletions, Invariant 1 is not violated. This means that no vertex needs to move *down* when processing a batch of insertions and no vertex needs to move *up* when processing a batch of deletions. We first describe the data structures that we maintain and then the RebalanceInsertions and RebalanceDeletions procedures below.

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, $L_v[j]$

[4] The magnitude of both ε and λ impact the approximation factor and the work, practically.

Algorithm 1 Update(B)

Input: A batch of edge updates B.

1: Initialize dynamic arrays B_{ins} and B_{del}.
2: **parfor** each edge update $e = (u, v) \in B$ **do**
3: **if** e is an edge insertion **then** Add e to B_{ins}.
4: **else** Add e to B_{del}.
5: Call RebalanceInsertions(B_{ins}). [Algorithm 2]
6: Call RebalanceDeletions(B_{del}). [Algorithm 3]

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 $[0, \ell(v) - 1]$, keyed by level number.
3: **parfor** each edge insertion $e = (u, v) \in \mathcal{B}$ **do**
4: Insert e into the graph.
5: **for** each level $l \in [0, K - 1]$ starting with $l = 0$ **do**
6: **parfor** each vertex v incident to B_{ins} or is marked, where $\ell(v) = l \cap \text{up}(v) > (2 + 3/\lambda)(1 + \varepsilon)^{gn(l)}$ **do**
7: 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 $\text{up}(u) > (2 + 3/\lambda)(1 + \varepsilon)^{gn(\ell+1)}$.
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: $L_x[l] \leftarrow L_x[l] \setminus \{v\}$, $L_x[l + 1] \leftarrow L_x[l + 1] \cup \{v\}$.

contains the neighbors of v at level $j < \ell(v)$. We describe specific data structure implementation details in Appendix A.

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 $\ell = K - 1$ (Algorithm 2, Line 5). For each level, in parallel we check the vertices incident to edge insertions in B_{ins} to see if they violate Invariant 1 (Line 4). 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 $L_v[l]$ (Line 9). We do not need to make any data structure modifications for w since v stays in $U[w]$. Similarly, no data structure modifications are necessary when both v and w move to level $l + 1$. Finally, for any neighbor of v on level $l + 1$ that now violates Invariant 1, we mark it (Line 11) and process it when we process level $l + 1$. We also update its U and L arrays (Line 12). We conclude by making appropriate modifications to L for each neighbor on levels $\geq l + 2$ (Line 13–Line 14). All neighbors of vertices that moved can be checked and processed in parallel.Julian: check if we already defined $gn(l)$ in text. the pseudocode uses it

A detailed example of this procedure is below.

Example 3.4. Fig. 4 shows an example of our entire insertion procedure described in Algorithm 2. The red lines in the example represent the batch of edge insertions. Thus, in (a), the newly

Algorithm 3 RebalanceDeletions(B_{del})

Input: A batch of edge deletions B_{del}.

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 $[0, \ell(v) - 1]$, keyed by level number.
3: **parfor** each edge deletion $e = (u, v) \in B_{del}$ **do**
4: Remove e from the graph.
5: **parfor** each vertex v where $\text{up}^*(v) < (1 + \varepsilon)^{gn(\ell(v)-1)}$ **do**
6: Calculate $\text{dl}(v)$ using CalculateDesireLevel(v).
7: **for** each level $l \in [0, K - 1]$ starting with level $l = 0$ **do**
8: **parfor** each vertex v where $\text{dl}(v) = l$ **do**
9: Move v to level l.
10: **parfor** each neighbor w of a vertex v that moved to level l where $\ell(w) \geq \ell(v)$ **do**
11: Let p_v and p_w be the previous levels of v and w, respectively, before the move.
12: **if** $\ell(w) = \ell(v) = l$ **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: $U[w] \leftarrow U[w] \setminus \{v\}$, $L_v[\ell(w)] \leftarrow L_v[\ell(w)] \setminus \{v\}$.
18: **else if** $p_v = \ell(w)$ **then**
19: $U[w] \leftarrow U[w] \setminus \{v\}$, $U[v] \leftarrow U[v] \setminus \{w\}$. Julian: I think the second statement on this line is not needed. w is going to be an up-neighbor of 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** $\text{up}^*(w) < (1 + \varepsilon)^{gn(\ell(w)-1)}$ **then**
23: Recalculate $\text{dl}(w)$ using Algorithm 4.

inserted edges are the edges (u, v), (u, x), and (x, w). We iterate from the bottommost level (starting with 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 $(2 + 3/\lambda)(1 + \varepsilon)^0 = 3$ provided $\lambda = 3$ and $\varepsilon = 0.4$. 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 $3 \cdot (1.4) = 4.2$ since level 3 is in group 1); w becomes unmarked because it no longer violates Invariant 1. Julian: the unmarking does not show up in the previous discussion and pseudocode. do we assume that vertices that move are automatically unmarked? if so Figure 4(c) should not have w marked

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}). Deletions are handled in a similar way to insertions, except for one major difference. Instead of moving vertices down one level at a time, we ensure that when we move a vertex, we move it to its final level (i.e., we will not move this

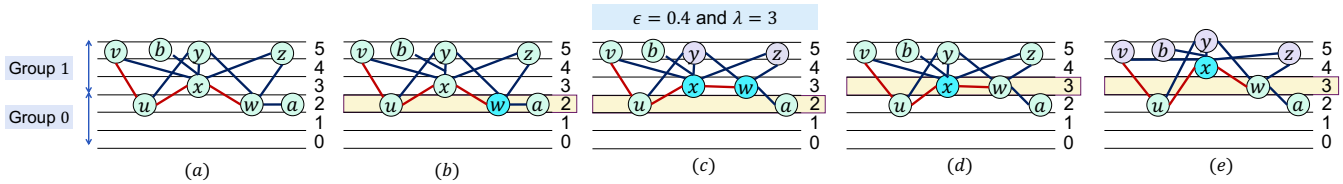

Figure 4: Example of RebalanceInsertions described in Example 3.4 for $\epsilon = 0.4$ and $\lambda = 3$.

vertex again during this procedure). As we show in the analysis, this guarantee is *crucial to obtaining low depth.*

Algorithm 3 shows the pseudocode. For each vertex v incident to an edge deletion, first we check whether it violates Invariant 2 (Algorithm 3, Line 5). In Line 5, $gn(\ell(v)-1)$ gives the group number i where $\ell(v)-1 \in g_i$. If it violates Invariant 2, we calculate its desire-level, $\mathsf{dl}(v)$, using CalculateDesireLevel (Algorithm 3, Line 6). We describe how to do this at the end of this subsection. As in the insertion procedure, we iterate through the levels from $l = 0$ to $l = K-1$ (Line 7). Then, in parallel for each vertex whose desire-level is l, we move the vertex to level l (Line 8–Line 9). As with insertions, we update the data structures of v and $w \in N(v)$ where $\ell(w) \geq l$ (Line 10–Line 21). Finally, we update the desire-level of neighbors of v that no longer satisfy Invariant 2 (Line 22–Line 23). We process all vertices that move as well as their neighbors in parallel.

An example of the RebalanceDeletions procedure is given below.

Example 3.5. Fig. 5 shows an example of our entire deletion procedure described in Algorithm 3 for $\epsilon = 1$ and $\lambda = 3$. The red dotted lines in the example represent the batch of edge deletions. Thus, in (a), the newly deleted edges are the edges (x, z) and (y, w). For each vertex adjacent to an edge deletion, we calculate the vertex's desire-level, or the closest level to its current level that satisfies Invariant 2.

In this example, shown in (b), 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 $(1+\varepsilon)^1 = 2$ since $\varepsilon = 1$ 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 calculated 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 data 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, in this example, y now violates the invariant. We recompute the desire-level of y and its desire-level is now 4 (shown in (d)). We proceed to process level 4 and move y to that level (shown in (e)).

CalculateDesireLevel(v). Algorithm 4 shows the procedure for calculating the desire-level, $\mathsf{dl}(v)$, 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 our desire-level. We initialize a variable d to $\mathrm{up}^*(v)$ (number of neighbors at or above $\ell(v)-1$). Starting with level $\ell(v)-2$, we add the number of neighbors in level $\ell(v)-2$ to d (Algorithm 4, Line 3). 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 double the number of levels from which we count neighbors (Line 6). In our example, in the next iteration, we sum the number of neighbors in levels $[\ell(v)-4, \ell(v)-3]$. 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 $[\ell', \ell_{prev}]$ to find the *closest* level to $\ell(v)$ that satisfies Invariant 2 (Line 5). The sum on Line 3 is done using a parallel reduce. Julian: define reduce

Algorithm 4 CalculateDesireLevel(v)

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

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

3.4 Coreness Estimates and Low-Outdegree Orientation

To obtain the coreness estimates from our PLDS, we only need to maintain the current level of each vertex and the number of levels in a group (recall that all groups have equal numbers of levels). Then, we calculate the estimate of the coreness of v to be $\hat{k}(v) = (1+\varepsilon)^{\max(\lfloor \ell(v) / \lceil \log_{1+\varepsilon} n \rceil \rfloor - 1, 0)}$, where the number of levels in a group is $\lceil \log_{1+\varepsilon} n \rceil$. In other words, our estimate of the coreness of v is $(1+\varepsilon)^i$, where i is the group index of the *highest level* that is the last level in a group *and* is equal to or lower than $\ell(v)$. We need to perform this calculation in order to obtain our optimum approximation ratios both in theory (Lemma 3.18) and in practice. To see an example, consider vertex y in Fig. 5 (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 so our coreness estimate for y is $(1+\varepsilon)^0 = 1$. This is a 2-approximation of its actual coreness 2.

To obtain a low-outdegree orientation we maintain an orientation of each edge from the endpoint at a lower level to the endpoint with a higher level. So edges are directed from lower to higher levels. For any two vertices on the same level, we direct the edge from the vertex with higher ID to the vertex with lower ID. Performing this procedure results in an 8α outdegree orientation, where α is the arboricity of the graph.

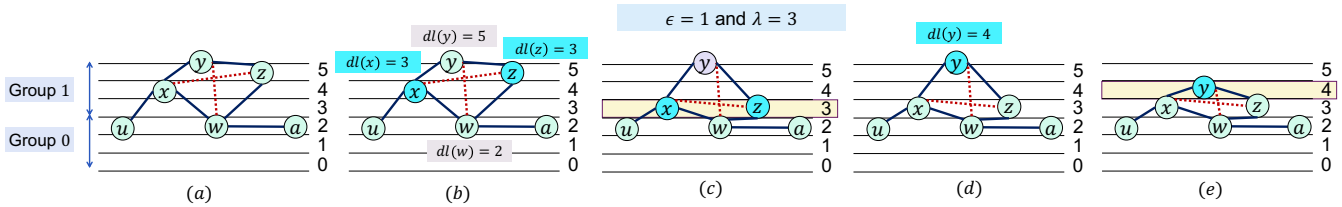

Figure 5: Example of RebalanceDeletions described in Example 3.5 for $\epsilon = 1$ and $\lambda = 3$.

3.5 Data Structure Implementations

Due to space constraints, we only specify the data structures used to obtain our randomized (high probability) bounds. We implement these data structures in our experiments. Our data structures presented here result in $O(n \log^2 m + m)$ space usage. However, we present two sets of data structures in the full paper: one that achieves $O(m)$ space with an $O(\log^2 m)$ factor increase in depth and another that results in a *deterministic* PLDS.

Our data structure uses parallel hash tables, which support x concurrent insertions, deletions, or finds in $O(x)$ amortized work and $O(\log^* x)$ depth *whp* [20]. We 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 $[n]$. We also maintain an array U of size n keyed by vertex ID that returns a hash table containing neighbors of v on levels $\geq \ell(v) - 1$. 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 $\mathsf{dl}(v) \neq \ell(v)$, partitioned by their levels. This allows us to quickly determine which vertices to move up (in Algorithm 2) or move down (in Algorithm 3).

3.6 Analysis

We now present the lemmas used to analyze our algorithms, as well as our overall bounds and approximation guarantee. We defer proofs to the full paper due to space constraints.

Depth and Work Bound. First, it is easy to show that there exists a level where both invariants are satisfied. This allows our PLDS data structure to assign each vertex to a single level.

Lemma 3.6. *If a vertex v violates Invariant 1, then there exists a level $i > \ell(v)$ where v satisfies both Invariant 1 and Invariant 2. If a vertex w violates Invariant 2, then there exists a level $i < \ell(w)$ where w satisfies both invariants or $i = 0$ (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 $(2 + 3/\lambda)(1 + \varepsilon)^{g(v)}$ 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 $(1+\varepsilon)^{gn(\ell(v))}$; hence, v also does not violate Invariant 2. The very last level of the K levels has up-degree bound $(2 + 3/\lambda)(1 + \varepsilon)^{\lceil \log_{1+\varepsilon}(m) \rceil} \geq 2n'$ where n' is the number of vertices with at least one adjacent edge. 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 show that the batch of insertions never violates Invariant 2 and a batch of deletions never violates Invariant 1. Using these two facts, we can immediately obtain our depth bounds for our insertion and deletion procedures.

Lemma 3.7 (Batch Insertions)**.** *Given a batch of insertions, $\mathcal{B}_{ins}$, Invariant 2 is never violated at any point during the rebalance procedure given by Algorithm 2.*

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 3.6. 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 3.6. □

Batch Insertion Depth Bound. 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.

Lemma 3.8. *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 $\mathsf{dl}(w) \leq i$.*

This means that each level is processed exactly once, resulting in at most $O(\log^2 n)$ levels to be processed sequentially.

Proof. Vertices on level $j \leq i$ already contain vertices on levels $\geq i$ in its up-degree. Such vertices when moved to levels $\geq i$ are still part of their up-degree. 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 $\ell(w) > i$ have $\mathsf{dl}(w) \leq i$, some neighbors of w must have moved to a level $\leq i$. By Lemma 3.7, no vertices move down during Algorithm 2, so this is not possible. □

Lemma 3.9. *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 Deletion Depth Bound. For the batch deletion algorithm (Algorithm 3), we prove that after all vertices which have $\mathsf{dl}(w) = i$ are moved to the i'th level, no vertex will have $\mathsf{dl}(v) \leq i$.

Lemma 3.10. *After processing all vertices that move to level i in Algorithm 3, no vertex needs to be moved to any level $j \leq i$ in a future iteration of Line 7; in other words, no vertex v has $\mathsf{dl}(v) \leq i$ after level i has been processed.*

As in the insertion case, this means that each level is processed exactly once, resulting in at most $O(\log^2 n)$ levels to be processed sequentially.

Proof. We prove this invariant 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 $< (1 + \varepsilon)$. Thus, all vertices that do not have $\mathsf{dl}(v) = 0$ have degree $\geq (1 + \varepsilon)$ 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 $< (1 + \varepsilon)^{gn(j-1)}$ at level $j > i + 1$ and up*-degree $\geq (1 + \varepsilon)^{gn(i)}$ 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 level $j > i + 1$. Suppose for the sake of contradiction that there exists some vertex w on level $j > i + 1$ 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. □

The above invariants will help us prove the depth of our algorithm.

Deterministic Setting In the deterministic setting, we maintain the list of neighbors using dynamic arrays, which also means that we maintain and access the sizes of these arrays in $O(1)$ work and depth. Because we are using dynamic arrays, we need to occasionally resize the arrays in $O(1)$ amortized work and $O(1)$ depth. Finally, we can modify the arrays in $O(1)$ work and depth (not counting the depth for resizing).

Randomized Setting In the randomized setting, we maintain the list of neighbors using hash tables keyed by level. Each vertex has one hash table which contains their neighbors at each level below them as well as all its neighbors at the same or higher levels. Then, vertices themselves are contained in separate hash tables for each level. Parallel lookups into the hash tables each require $O(1)$ work and depth, *whp*. Then, modifying (inserting and deleting) elements within the hash tables require $O(\log \log m)$ depth and work proportional to the number of inserted elements, *whp*.

Space-Efficient Setting In the space-efficient setting, we replace the structure used to represent L_{v_i} with a linked list. Inserting and deleting nodes from the linked list requires $O(1)$ work and depth (assuming we are given a pointer to the node). Then, resizing the dynamic arrays (pointed to by the linked lists to maintain the set of elements in each level) require $O(1)$ amortized work and $O(1)$ depth.

The only additional depth we need to consider is the depth acquired from Algorithm 4. Both the doubling search and the binary search require $O(\log K) = O(\log \log m)$ depth. All other depth comes from concurrently modifying and accessing dynamic arrays and hash tables, which can be done in $O(\log * m)$ depth *whp*.

Using the above, we prove the depth of Algorithm 1.

Lemma 3.11. *Algorithm 1 returns a deterministic level data structure that maintains Invariant 1 and Invariant 2 and has $O(\log^3 m)$ worst-case depth.*

Proof. All edge updates can be partitioned into $\mathcal{B}_{ins}$ and $\mathcal{B}_{del}$ in parallel in $O(1)$ depth. Then, it remains to bound the depth of Algorithm 2 and Algorithm 3.

Algorithm 2 iterates through all $K = O\left(\log^2 n\right)$ levels sequentially. By Lemma 3.8, no vertices on level $\leq i$ will violate Invariant 1 after processing level i. Thus, by the end of the procedure no vertices violate Invariant 1. By Lemma 3.7, 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. To resize the dynamic array, we require $O(\log n)$ depth whenever the array becomes too large or too small to compute the offsets by which to insert the new elements. For each level, an additional depth of $O(\log n)$ might be

necessary to compute the element offsets and then resize the arrays in $O(1)$ depth. Then, Algorithm 2 requires $O(\log^3 n)$ worst-case depth.

Algorithm 3 iterates through all $K = O\left(\log^2 n\right)$ levels sequentially. By Lemma 3.9 and Lemma 3.10, after processing level i, no vertices have $\mathsf{dl}(v) \leq i + 1$. Thus, by the end of the procedure all vertices satisfy Invariant 2. Furthermore, Invariant 1 was never violated due to Lemma 3.9. 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.

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

Corollary 3.12. *Algorithm 1 returns a randomized level data structure that maintains Invariant 1 and Invariant 2 and has $O(\log^2 m \log \log m)$ depth, whp and $O(n \log^2 m + n)$ space.*

Proof. The proof is the same as the proof of Lemma 3.14 except we replace dynamic arrays with hash tables. Simultaneously changing the values within the hash table require $O(\log^* m)$ depth *whp*. Then, the depth per level of the structure is dominated by Algorithm 4. Thus, the total depth of our randomized algorithm is $O(\log^2 m \log \log m)$ *whp*, and $O(n + m)$ space. □

The additional $n \log^2 m$ space comes from needing to store the L_v dynamic arrays for each vertex v in the graph. We show that with slightly more complicated data structures, we can obtain *space-efficient* structures in the full version of our paper.

Corollary 3.13. *Algorithm 1 returns a space-efficient, deterministic, level data structure that maintains Invariant 1 and Invariant 2 and has $O(\log^4 m)$ worst-case depth.*

Proof. The proof is the same as the proof of Lemma 3.14 except we replace Algorithm 4 with an $O(\log^2 m)$ linear in number of levels search. Thus, the total depth is $O(\log^4 m)$. □

Lemma 3.14. *Algorithm 1 returns a deterministic level data structure that maintains Invariant 1 and Invariant 2 and has $O(\log^3 n)$ worst-case depth.*

Proof. All edge updates can be partitioned into $\mathcal{B}_{ins}$ and $\mathcal{B}_{del}$ in parallel in $O(1)$ depth. Then, it remains to bound the depth of Algorithm 2 and Algorithm 3.

Algorithm 2 iterates through all $K = O\left(\log^2 n\right)$ levels sequentially. By Lemma 3.8, no vertices on level $\leq i$ will violate Invariant 1 after processing level i. Thus, by the end of the procedure no vertices violate Invariant 1. By Lemma 3.7, 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. To resize the dynamic array, we require $O(\log n)$ depth whenever the array becomes too large or too small. Thus, for each level, an additional depth of $O(\log n)$ might be necessary to resize the arrays for each level. Then, Algorithm 2 requires $O(\log^3 n)$ worst-case depth.

Algorithm 3 iterates through all $K = O\left(\log^2 n\right)$ levels sequentially. By Lemma 3.9 and Lemma 3.10, after processing level i, no vertices have $\mathsf{dl}(v) \leq i + 1$. Thus, by the end of the procedure all vertices satisfy Invariant 2. Furthermore, Invariant 1 was never violated by Lemma 3.9. 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 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.

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

Our work bound uses potential functions similar to those presented in Section 4 of [7]. We show our parallel algorithm serializes to a set of sequential steps that can be analyzed using these potential functions. Together, we obtain the work and depth bounds shown in the following lemma.

Julian: this is a repeated of theorem 1.2. should we just refer to it instead?

Lemma 3.15. *For a batch of $\mathcal{B} < m$ updates, Algorithm 1 returns a PLDS that maintains Invariant 1 and Invariant 2 in $O(\mathcal{B} \log^2 m)$ amortized work and $O(\log^2 m \log \log m)$ depth whp, using $O(n \log^2 m + m)$ space.*

Proof. Instead of the dynamic array used in Lemma 3.14, we use hash tables to maintain each of our neighbors which can be maintained in $O(\log^* n)$ with high probability. Then, running Algorithm 4 require $O(\log \log n)$ depth. Thus, Algorithm 2 requires $O(\log^2 n \log^* n)$ depth and Algorithm 3 requires $O(\log^2 n \log \log n)$ depth.

In conclusion, Algorithm 1 requires $O(\log^2 n \log \log n)$ depth with high probability. □

3.7 $(2 + \delta)$-Approximation of Coreness

The *coreness estimate*, $\hat{k}(v)$, is an estimate of the coreness of a vertex v. As mentioned previously in Section 3.4, we compute an estimate of the coreness using information about the level of the vertex v. Here, we show that such an estimate provides us with a $(1 + \delta)$-approximation to the actual coreness of v for any constant $\delta > 0$. (We can find an approximation for any fixed δ by appropriately setting our parameters ε and λ.) To calculate $\hat{k}(v)$, we first 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 3.16 (Coreness Estimate)**.** *The* coreness estimate $\hat{k}(v)$ *of vertex v is $(1 + \varepsilon)^i$, where $i = \arg\max_j \left(\ell(v) \geq j \cdot \lceil \log_{(1+\varepsilon)} m \rceil - 1\right)$.* Julian: to be consistent can we use the same expression as in section 3.4?

We prove that our PLDS maintains a $(2 + 3/\lambda)(1 + \varepsilon)$ approximation of the coreness value of each vertex, for any constants $\lambda > 3$ and $\varepsilon > 0$, if we use $\hat{k}(v)$ as computed in Definition 3.16.

Therefore, we obtain the following lemma giving the desired $(2+\delta)$-approximation, a 2-factor improvement on the previous approximation bounds. Previous works were only able to obtain a $(4+\delta)$-approximation theoretical bound [42]. We see in our experimental analysis that such a bound improves the maximum error of our algorithm compared to previous algorithms that obtain similar average errors. Given fixed ε and λ, the maximal error of our algorithm is given by $(2+3/\lambda)(1+\varepsilon)$. The proof of the error bound can be found in our full paper.

We first show some properties of $\hat{k}(v)$ and then we show that we can obtain an approximate coreness number by looking at $\hat{k}(v)$.

Lemma 3.17. *Let $\hat{k}(v)$ be the coreness estimate and $k(v)$ be the coreness of v, respectively. If $k(v) > (2+3/\lambda)(1+\varepsilon)^{g'}$, then $\hat{k}(v) \geq (1+\varepsilon)^{g'}$. Otherwise, if $k(v) < \frac{(1+\varepsilon)^{g'}}{(2+3/\lambda)(1+\varepsilon)}$, then $\hat{k}(v) < (1+\varepsilon)^{g'}$.*

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

Now we prove that if $k(v) < \frac{(1+\varepsilon)^{g'}}{(2+3/\lambda)(1+\varepsilon)}$, then $\hat{k}(v) < (1+\varepsilon)^{g'}$. We assume for sake of contradiction that $k(v) < \frac{(1+\varepsilon)^{g'}}{(2+3/\lambda)(1+\varepsilon)}$ and $\hat{k}(v) \geq (1+\varepsilon)^{g'}$. 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 $d_S(v) < \frac{(1+\varepsilon)^{g'}}{(2+3/\lambda)(1+\varepsilon)}$. 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 $\hat{k}(v) \geq (1+\varepsilon)^{g'}$, $\ell(v) \geq T(g')$. If we consider a level $\ell(v) > T(g')$, then the up*-degree cannot decrease due to 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

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

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

Thus, at least $(1+\varepsilon)^{g'} - \frac{(1+\varepsilon)^{g'}}{(2+3/\lambda)(1+\varepsilon)}$ 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* $(1+\varepsilon)^{g'}$ 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 $\frac{(1+\varepsilon)^{g'}X}{2}$ edges. Each vertex that is pruned can remove *at most* $\frac{(1+\varepsilon)^{g'}}{(2+3/\lambda)(1+\varepsilon)}$ edges when it is pruned, by definition of our pruning procedure. Thus, the *minimum* number of vertices we must prune in order to prune the $X = \left(\frac{(2+3/\lambda)(1+\varepsilon)}{2}\right)^{j-1}\left((1+\varepsilon)^{g'} - \frac{(1+\varepsilon)^{g'}}{(2+3/\lambda)(1+\varepsilon)}\right)$ vertices from the previous step is

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

Thus, we have proven our argument for the $(j+1)$-st induction step. Note that for $j = \lceil \log_{(2+3/\lambda)(1+\varepsilon)/2}(2m+1) \rceil$, we have $j \leq 2\lceil \log_{(1+\varepsilon)}(m) \rceil$. This is because, since we pick λ to be a constant, $2+3/\lambda > 2$ and for large enough m, $\log_{(2+3/\lambda)(1+\varepsilon)/2}(2m+1) \leq 2\lceil \log_{(1+\varepsilon)}(m) \rceil$. Then, by our induction, we must prune at least $2m+1$ vertices at this step, which we cannot because there are at most $2m$ vertices. 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 $n \leq 2m$. Thus, our assumption is incorrect and we have proven our desired property. □

Lemma 3.18. *The coreness estimate $\hat{k}(v)$ of a vertex v satisfies $\frac{k(v)}{(2+\delta)} \leq \hat{k}(v) \leq (2+\delta)k(v)$ for any constant $\delta > 0$.*

Proof. Suppose $\hat{k}(v) = (1+\varepsilon)^{g}$. Then, by Lemma 3.17,

$$\frac{(1+\varepsilon)^{g}}{(2+3/\lambda)(1+\varepsilon)} \leq k(v) \leq (2+3/\lambda)(1+\varepsilon)^{g+1}.$$

Then, solving the above bounds, $\frac{k(v)}{(2+3/\lambda)(1+\varepsilon)} \leq \hat{k}(v) \leq (2+3/\lambda)(1+\varepsilon)k(v)$. For any constant $\delta > 0$, there exists constants $\lambda, \varepsilon > 0$ where $\frac{k(v)}{2(1+\delta)} \leq \hat{k}(v) \leq 2(1+\delta)k(v)$. □

By Lemma 3.18, it suffices to return $\hat{k}(v)$ as the estimate of the core number of $\hat{k}(v)$.

Lemma B.1 and Lemma 3.18 together prove Theorem 3.19. Furthermore, based on the analysis of [25], our PLDS directly gives a low out-degree orientation with the same complexity bounds if we orient all edges towards neighbors at higher levels and arbitrarily orient edges between neighbors in the same level.

Theorem 3.19. *Provided an input graph with m edges, and a batch of $\mathcal{B}$ updates, our algorithm maintains a $(2+\delta)$-approximation of the coreness values and a $(8+\delta)\alpha$-low-outdegree orientation for all vertices (for any constant $\delta > 0$) in $O(\mathcal{B}\log^2 m)$ amortized work and $O(\log^2 m \log\log m)$ depth whp, using $O(n\log^2 m + m)$ space.*

Corollary 3.20. *Provided an input graph with m edges, and a batch of $\mathcal{B}$ updates, our algorithm maintains a $(2+\delta)$-approximation of the coreness values for all vertices (for any constant $\delta > 0$) in $O(\mathcal{B}\log^2 m)$ amortized work and $O(\log^3 m)$ depth worst-case, using $O(n\log^2 m + m)$ space.*

Corollary 3.21. *Provided an input graph with m edges, and a batch of $\mathcal{B}$ updates, our algorithm maintains a $(2+\delta)$-approximation of the coreness values for all vertices (for any constant $\delta > 0$) in $O(\mathcal{B}\log^2 m)$ amortized work and $O(\log^4 m)$ depth worst-case, using $O(n+m)$ space.*

Lemma B.1 and Lemma 3.18 together prove Theorem 3.19 (Theorem 1.2 in the main paper). Furthermore, based on the analysis of [25], our PLDS directly gives a low out-degree orientation with the same complexity bounds if we orient all edges towards neighbors at higher levels and arbitrarily orient edges between neighbors in the same level.

Corollary 3.22. *The PLDS maintained by Algorithm 1 provides a low out-degree orientation of out-degree at most $(8+\delta)\alpha$ where α is the arboricity, if all edges are oriented from vertices in lower levels to vertices in higher levels and edges between vertices in the same level are oriented arbitrarily.*

Proof. 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 $d \leq 2\alpha$. By Lemma 3.18, the vertices in the largest k-core in the graph are in a level with group number at most $\log_{(1+\varepsilon)}((2+3/\lambda)d)+1$. This means that the up-degree of each vertex in that group is at most $(2+3/\lambda)(1+\varepsilon)^{\log_{(1+\varepsilon)}((2+3/\lambda)d)} = (4+\delta)d$ for any constant $\delta > 0$ for an appropriate setting of $\lambda > 3$. We then obtain an $(8+\delta)\alpha$ out-degree orientation where α is the arboricity of the graph. □

Our data structure also provides an approximation of the densest subgraph within the input graph by the Nash-Williams theorem.

Corollary 3.23. *The PLDS maintained by Algorithm 1 provides an $(8+\delta)$-approximation on the density of the densest subgraph within the input graph.*

4 Static $(2+\delta)$-Approximate k-core

Due to the P-completeness of k-core decomposition for $k \geq 3$ [3], all known static exact k-core algorithms do not achieve polylogarithmic depth. We introduce a linear work and polylogarithmic depth $(2+\delta)$-approximate k-core decomposition algorithm based on the parallel bucketing-based peeling algorithm for static *exact* k-core decomposition of Dhulipala et al. [14]. 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 k, increments k, and repeats until the graph becomes empty. The k-core value of v is 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 5 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

Algorithm 5 Static Approximate k-core Decomposition

Input: An undirected graph $G(V, E)$.
Output: An array of $(2+\delta)$-approximate coreness values.
1: $\forall v \in V$, let $C[v] = |N(v)|$
2: Let M be a bucketing structure formed by initially assigning each $v \in V$ to the $\lceil \log_{1+\delta} C[v] \rceil$'th bucket.
3: *finished* $\leftarrow 0$, $t \leftarrow 0$.
4: **while** (*finished* $< |V|$) **do**
5: $(I, bkt) \leftarrow$ Vertex IDs and bucket ID of next (peeled) bucket in M.
6: **if** $> \log_{1+\epsilon}(n)$ iterations with $t = bkt$ **then** $t \leftarrow t+1$
7: **else if** $bkt \neq t$ **then** $t \leftarrow bkt$
8: $R \leftarrow \{(v, r_v) \mid v \in N(I), r_v = |\{(u, v) \in E \mid u \in I\}|\}$
9: $U \leftarrow$ Array of length $|R|$.
10: **parfor** $R[i] = (v, r_v)$, $i \in [0, |R|)$ **do**
11: *inducedDeg* $= C[v] - r_v$
12: $C[v] = \max(inducedDeg, \lceil (1+\delta)^{t-1} \rceil)$
13: $newbkt = \max(\lceil \log_{1+\delta} C[v] \rceil, t)$
14: $U[i] = (v, newbkt)$
15: Update M for each $(u, newbkt)$ in U.

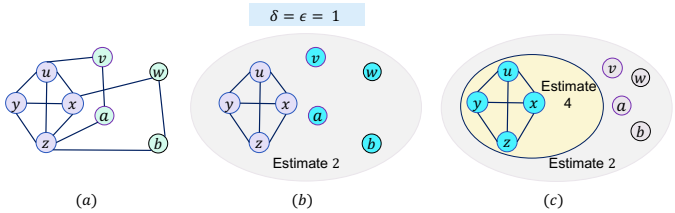

Figure 6: Example of a run of Algorithm 5 described in Example 4.1.

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 2). 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 3). The rest of the algorithm performs peeling, where the peeling thresholds are powers of $(1+\delta)$. The peeling loop (Line 4–Line 15) first extracts the lowest non-empty bucket from M (Line 5), which consists of I, a set of vertex IDs of vertices that are being peeled, and the bucket number bkt. If more than $\log_{1+\epsilon}(n)$ rounds of peeling have occured at the threshold $(1+\delta)^t$, 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 4.1. Fig. 6 shows a run of Algorithm 5 on an example graph. Given the parameters $\delta = \varepsilon = 1$, 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 $(1+\delta)^1 = 2$ 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 $(1+\delta)^2 = 4$ (shown in (c)). In this example, the estimates produced are 4/3-approximations of the real coreness values. Julian: since $\delta = 1$ we should have a 3-approximation? also it's not clear whether ϵ is related to the approximation factor. it would be good to explain this earlier

In the full paper, we prove that Algorithm 5 finds an $(2+\delta)$-approximate k-core decomposition in $O(m)$ expected work and $O(\log^3 m)$ depth *whp*, using $O(m)$ space, as stated in Theorem 4.2. Our proof uses the quality guarantees in Lemma 3.18, as well as an efficient parallel semisort implementation [23].

Julian: this is a repeated of theorem 1.1. should we just refer to it instead?

Theorem 4.2. *(Theorem 1.1 of the Main Paper) For a graph with m edges, for any constant $\delta > 0$, Algorithm 5 finds an $(2+\delta)$-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 Lemma 8 of [19]. 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 2) can be maintained via a sparse set (hash map), or by using the bucketing data structure from [14]. The outer loop iterates for $O(\log n)$ times (Line 4). Within each iteration of the outer loop, we iterate for $O(\log_{(1+\varepsilon)} n) = O(\log n)$ rounds for constant ϵ. After obtaining a set of vertices, we update the buckets using semisort in $O(\log n)$ depth *whp* [14]. Thus the overall depth of the algorithm is $O(\log^3 m)$ for any constant $\epsilon > 0$.

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 [23]. 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 $O(\log_{1+\delta} n) = O(\log n)$ buckets for any constant $\delta > 0$. 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 $O(m)$. □

Julian: we have not defined P and NC. we should either define them, or make the discussion more vague The approximation guarantees provided by our algorithm are essentially the best possible, under widely believed conjectures. Specifically, Anderson and Mayr [3] 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 1/2 Julian: 2? unless P = NC. Thus, obtaining a polynomial work and polylogarithmic depth $(2-\epsilon)$-approximation Julian: $2-\delta$? to the coreness value of each vertex would yield $\frac{1+\delta}{2}$ Julian: $2-\delta$? approximation to the optimization version of the High-Degree Subgraph problem, and imply that P = NC.

In recent years, several results have given parallel algorithms that obtain a $(1+\epsilon)$-approximation to the coreness values in distributed models of computation such as the Massively Parallel Computation model [18, 19]. 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.

Table 2: Sizes of graph inputs.

Graph Dataset	Num. Vertices	Num. Edges
com-dblp	425,957	2,099,732
brain	784,262	267,844,669
wiki-talk-temporal	1,140,149	2,787,967
com-youtube	1,138,499	5,980,886
sx-stackoverflow	2,601,977	28,183,518
com-lj	4,847,571	85,702,474
com-orkut	3,072,627	234,370,166
central-usa	14,081,816	16,933,413
full-usa	23,072,627	28,854,312
twitter	41,652,231	2,405,026,092
com-friendster	65,608,366	3,612,134,270

5 Experimental Evaluation

Setup. We use `c2-standard-60` Google Cloud instances, which have 30 cores with two-way hyper-threading (3.1 GHz Intel Xeon Cascade Lake) and 236 GiB memory, and `m1-megamem-96` Google Cloud instances, which have 48 cores with two-way hyper-threading (2.0 GHz Intel Xeon Skylake) and 1433.6 GB memory. Our programs use a work-stealing scheduler, and are compiled using g++ (version 7.5.0) with the `-O3` flag. We terminate experiments that take over 3 hours. We test our algorithms on real-world undirected graphs from SNAP [31], the DIMACS Shortest Paths challenge road networks [13], and Network Repository [38], shown in Table 2, namely ***com-dblp***, ***brain***, ***wiki-talk-temporal***, ***com-orkut***, ***com-friendster***, ***sx-stackoverflow***, ***full-usa***, ***central-usa***, ***com-youtube***, and ***com-lj***. We also used ***twitter***, a symmetrized version of the Twitter network [30]. For each network, we propose the graph to remove duplicate edges as well as zero degree edges and self-loops. The table reflects the size of each network *after* this preprocessing. Both *sx-stackoverflow* and *wiki-talk-temporal* are temporal networks obtained from the SNAP database. For these two networks, we maintain the same order of edge insertions and deletions in the order that they occur temporally as provided by SNAP. *full-usa* and *central-usa* are two high diameter road network graphs and *brain* is a highly dense network of the human brain where the largest k-core has size $\geq 1,2000$.

All experiments are run on the `c2-standard-60` instances, except for *twitter* and *com-friendster*, which are run on the

m1-megamem-96 instances as they require more memory. The edge updates for the dynamic algorithms (for all except the temporal networks) are generated by taking a random permutation of a list containing two copies of each edge, where the first appearance of an edge is taken as a insertion, and the second appearance is taken as a deletion. Batches are generated by taking regular intervals of the list. For the experiments on insertion-only batches, we ignore the second appearance of each edge and, likewise, the first for deletion-only batches.

For the static algorithms in the batch-dynamic setting, within each batch we order all insertions in the batch before all deletions in the batch. Then, we generate two static graphs per batch, one following all insertions in the batch, and the other following all deletions in the batch. We re-run the static algorithms on each static graph generated in this manner, to obtain comparable per-batch running times. [5]

Algorithms. Unless otherwise noted, we run our parallel algorithms using all available cores with two-way hyper-threading. Notably, in our implementation, we optimized the performance of our PLDS by considering $\frac{\lceil \log_{(1+\varepsilon)} n \rceil}{50}$ levels per group instead of $\lceil \log_{(1+\varepsilon)} n \rceil$ (that ensures our max-error approximation bounds). We also implemented a version of our structure that exactly follows our theoretical algorithm and compared the performance of both structures. We see through our experiments that even such a simple optimization resulted in massive gains in performance.

We test the following implementations: ***LDS***: the *sequential batch-dynamic* approximate algorithm using our level data structure; ***PLDS***: our *parallel batch-dynamic* approximate algorithm; ***PLDSOpt***: our optimized *parallel batch-dynamic* approximate algorithm (that uses less levels per group); ***ApproxKCore***: our *parallel static* approximate algorithm; ***ApproxKCore***: our *parallel static* approximate algorithm; ***ExactKCore***: the *parallel static* algorithm by Dhulipala et al. [14]; ***Hua***: the *parallel, batch-dynamic* exact algorithm by Hua et al. [26]; and ***Sun***: the *sequential dynamic* approximate algorithm by Sun et al. [42].

We implemented our algorithms using the Graph Based Benchmark Suite [15], and we use the atomic compare-and-swap and fetch-and-add instructions. We also used a concurrent hash table with linear probing [40] for the level sets $L[v][j]$ and $U[v]$. 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.

Accuracy vs. Running Time. We evaluated the empirical error ratio of the per-vertex core estimates given by our algorithms (LDS, PLDS, and PLDSOpt) and Sun on *com-dblp* and *com-lj*, using batches of size 10^5 and 10^6 respectively of both insertions and deletions; the results are shown in Fig. 7. The figure shows the average batch time against the average and maximum per-vertex core estimate error ratio. [6] The parameters that we use for LDS, PLDS, and PLDSOpt are $\epsilon = \{0.1, 0.2, 0.4, 0.8, 1.6, 3.2\}$ and $\lambda = \{3, 6, 12, 24, 48, 96\}$. For Sun, we use the parameters $\epsilon_{\text{sun}} = \lambda_{\text{sun}} = \{0.1, 0.2, 0.4, 0.8, 1.6, 3.2\}$, and $\alpha_{sun} = \{2(1 + 3\epsilon_{\text{sun}})\}$ (these are parameters used in their algorithm [42]). We also tested $\alpha_{sun} = 1.1$, as done in Sun et al.'s code [42]. In this setting, the theoretical bounds given by Sun et al. [42] no longer hold, but it gives better estimates empirically. We compare this heuristic setting to a similar heuristic in our PLDS and LDS algorithms, where we pick a $\lambda < 3$ such that $(2 + 3/\lambda) = 1.1$ for $\epsilon = \{0.4, 0.8, 1.6, 3.2\}$.

[5] The insertions and deletions are processed separately in the static algorithms, because in large batches many of the insertions and deletions in the generated updates cancel each other out.

[6] This error ratio is computed as $\max(\frac{approx}{exact}, \frac{exact}{approx})$. If the exact core number is 0, we ignore the vertex in our error ratio; for vertices of non-zero degree, the lowest estimated core number is 1.

Table 3: Running times per edge in an insertion-only batch of size 10^5, comparing Hua et al. to PLDS. Note that the superscript $^\circ$ refers to running times obtained from the second-to-last batch, while the superscript * refers to running times obtained from the average running time over all batches.

	COM-ORKUT	COM-LJ	COM-YOUTUBE
Hua et al.	$\sim 20\mu s$	$\sim 10\mu s$	$\sim 3\mu s$
PLDS$^\circ$	$5.89\mu s$	$6.84\mu s$	$3.40\mu s$
PLDS*	$5.59\mu s$	$5.80\mu s$	$2.68\mu s$

Overall, we see that using theoretically efficient parameters, our PLDSOpt, PLDS and LDS algorithms are faster than Sun, for parameters that give similar average and maximum per-vertex core estimate error ratios. For *com-dblp*, PLDSOpt achieves 22.35–195.82x and PLDS achieves 11.15–25.02x speedups over Sun and for *com-lj*, PLDSOpt achieves 27.64–497.63x and PLDS achieves 36.44–57.43x speedups over Sun. Using the same parameters, for *com-dblp*, PLDSOpt achieves a 3.43–57.124x speedup over PLDS; for *com-lj*, the speedups are 1.70–51.36x. Moreover, PLDS achieves 1.41–5.23x speedup over LDS on *com-dblp*, and 3.30–9.57x speedup over LDS on *com-lj*, using the same parameters; PLDS does not show as much speedup over LDS on *com-dblp* because it is a small graph. PLDS also has much lower maximum error ratios compared to Sun and PLDSOpt on parameters that give similar average error ratios. PLDSOpt gives similar maximum errors to Sun with a maximum error factors of 4–58 compared to Sun's maximum error factors of 1.5–47 for *com-dblp*; for *com-lj*, PLDSOpt gives maximum error factors of 3–136.3 compared to Sun's maximum error of 4–182.

The heuristic settings in both our algorithms and Sun give better approximations at the cost of speed. We achieve similar error ratios with up to 3.96x speedup using PLDS on *com-dblp*, and up to 22.64x speedup using PLDS on *com-lj*. Also, PLDS achieves 4.28–11.01x speedup over LDS on *com-dblp*, and 8.71–11.43x speedup over LDS on *com-lj*. Unfortunately, PLDSOpt was not able to achieve comparable error ratios for these settings.

For the rest of the experiments, we fix $\epsilon = 0.4$ and $\lambda = 3$; these parameters offer a reasonable tradeoff between approximation error and speed, as shown in Fig. 7. We also focus on insertion-only and deletion-only batches since inputs are initially filtered anyways into such batches.

Comparison with Hua et al. We also compare with a parallel batch-dynamic algorithm for exact k-core by Hua et al. [26]. We obtained a multicore implementation of their code and tested their code under the same experimental conditions as our code. Although the experiments in [26] tested the runtime of randomly sampling a batch of insertions or deletions, we also tested our code under such settings and found it obtained similar speedups to the experiments run under our setting; thus, we present here the results under our experimental setting. Hua et al. included a timing function in their code which we use to time their code. However, this timing function

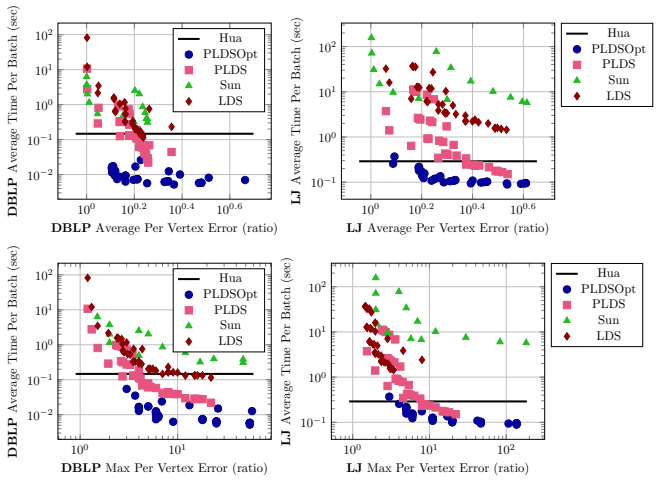

Figure 7: Comparison of the average per-batch time versus the average (a) and maximum (b) per-vertex core estimate error ratio of LDS, PLDS, PLDSOpt and Sun, using varying parameters, on the *com-dblp* and *com-lj* graphs, with a batch size of 10^5 and 10^6, respectively. The data uses theoretically efficient (T.E.) parameters as well as those using $(2+3/\lambda) = \alpha_{sun} = 1.1$ (non T.E.). The first row shows the average error vs. the average runtime per batch for *com-dblp* and *com-lj* while the second row shows the maximum error for both datasets. The runtime for Hua is shown as a black horizontal line.

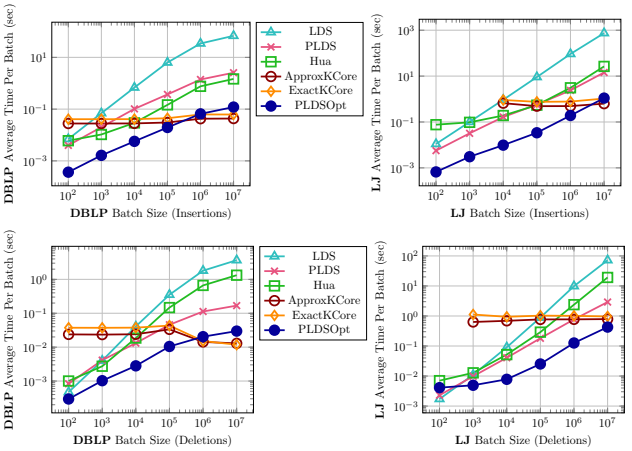

Figure 8: Average insertion-only and deletion-only per-batch running times on varying batch sizes for LDS, PLDS, ExactKCore, and ApproxKCore on *com-dblp* and *com-lj*. The missing batch sizes for ApproxKCore and ExactKCore timed out at 3 hours. The first row shows the running times for insertion-only batches and the second row shows the running times for deletion-only batches.

does not include the time to process the graph and maintain their data structures; we include all such times (for updating the graph and maintaining our data structures) in our code. If we include this time in Hua et al's implementation, their running times receives an up to 8x increase for some experiments. But for our experiments, we use the original timing functionality in Hua et al.'s code *without* this additional time.

Their experiments randomly sample a batch from the graph, delete the batch from the graph, and then re-insert the batch into

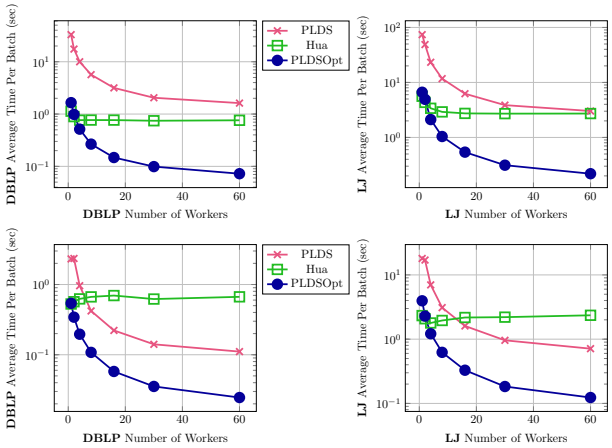

Figure 9: Parallel speedup of PLDSOpt, PLDS and Hua, with respect to their single-threaded running times on *com-dblp* and *com-lj*, using batches of size 10^6 for all algorithms. The last "60" on the x-axis indicates 30 cores with hyper-threading.

the graph, using batches of size 10^5. As such, we compare their per-edge running times to PLDS's per-edge running times, using running times from both the second-to-last batch.[7] The results are shown in Table 3, and we see that PLDS is up to 3.4x faster than Hua et al., although their algorithm is exact.

5.1 Experiments on Insertions

Batch Size vs. Running Time. Fig. 8 (first row) shows the average insertion-only per-batch running times on varying batch sizes for LDS, PLDS, PLDSOpt, ExactKCore, and ApproxKCore on *com-dblp* and *com-lj*. ExactKCore and ApproxKCore recompute the k-core decomposition on the current batch as well as all previously inserted edges. We see that PLDSOpt is 16.55–6205.19x and PLDS is 1.76–51.81x faster than LDS, and ApproxKCore is 1.37–1.67x faster than ExactKCore overall. PLDSOpt is also 8.47-119.77x faster than PLDS. Moreover, for *com-lj*, ExactKCore and ApproxKCore both timeout for small batch sizes. For *com-dblp*, we see that PLDSOpt is up to 75.94x faster, PLDS is up to 7x faster than ApproxKCore for small batch sizes, and even our sequential LDS is up to 3.98x faster than ApproxKCore. Against Hua, PLDSOpt achieves a speedup over all batches from 5.17–16.43x for *com-dblp* and 15.97–114.52x for *com-lj*. PLDS achieves speedups over Hua for the smaller batches, a speedup of 1.51x for *com-dblp* and up to 13.51x speedup for *com-lj*. Finally, ApproxKCore achieves speedups of up to 33.90x for *com-dblp* and 42.02x for *com-lj* over Hua for the larger batch sizes.

Thread Count vs. Running Time. We focus on the parallel speedups of our dynamic algorithms since the parallel speedups of static algorithms have been well-studied in the past. Fig. 9 (top row) shows the scalability of PLDSOpt, PLDS, and Hua with respect to their single-thread running times on *com-dblp* and *com-lj* for insertion-only batches. We display the scalability of the algorithms on dynamic inputs, of insertion-only size 10^6. PLDSOpt and PLDS achieve up to 30.28x and 26.46x self-relative speedup, respectively. Hua achieves up to a 2.07x self-relative speedup. We see that our PLDS algorithms achieve greater self-relative speedups than Hua.

[7] We use second-to-last batch because the last batch might not be a full 10^5 sized batch.

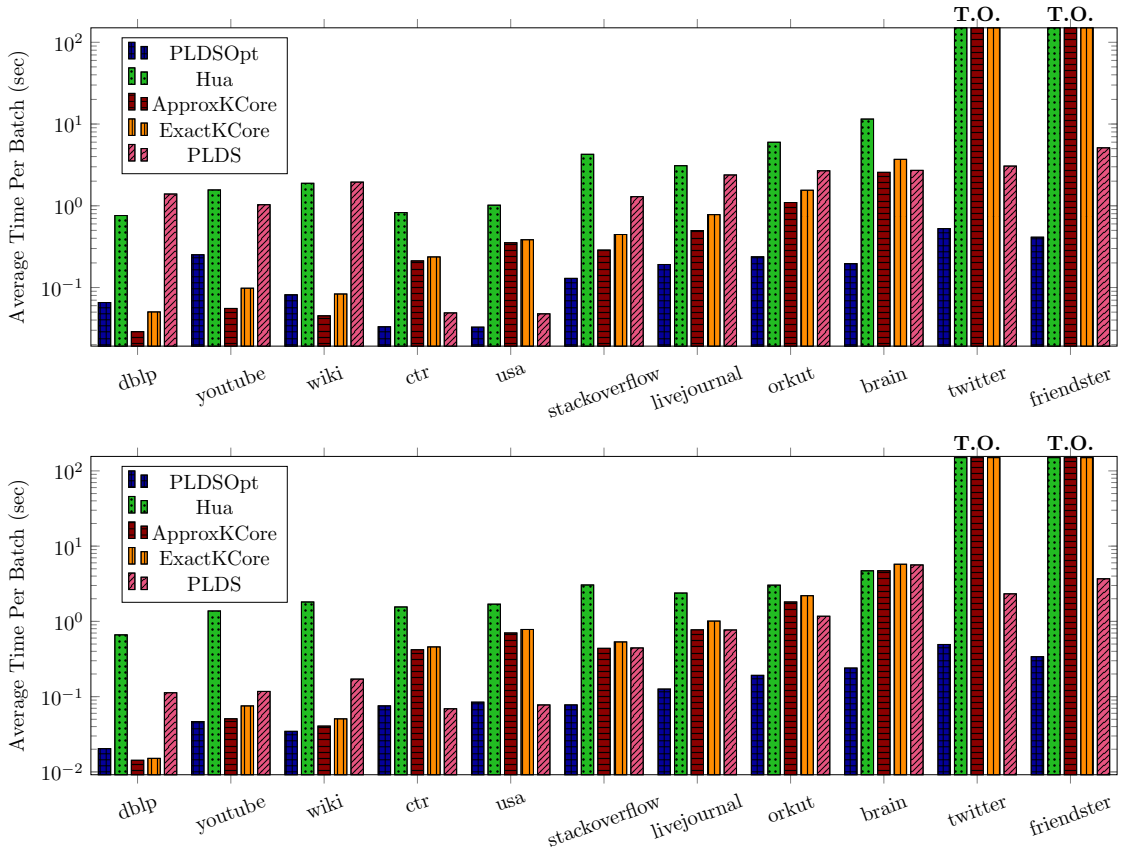

Figure 10: Average per-batch running times for PLDSOpt, Hua, PLDS, ApproxKCore, and ExactKCore, on *com-dblp*, *com-youtube*, *wiki-talk-temporal*, *central-usa*, *full-usa*, *sx-stackoverflow*, *com-livejournal*, *com-orkut*, *brain*, *twitter*, and *com-friendster* with batches of size 10^6 (and approximation settings $\varepsilon = 0.4$ and $\delta = 3$ for PLDSOpt and PLDS). Hua, ApproxKCore and ExactKCore timed out (T.O.) at 3 hours for *twitter* and *com-friendster*. Top graph shows insertion-only batch runtimes and the bottom graph shows deletion-only batch runtimes.

Additional Graphs. Fig. 10 (top) shows the runtimes of PLDS, ExactKCore, and ApproxKCore on additional graphs, using insertion-only batches, all of size 10^6. Hua, ExactKCore and ApproxKCore timed out on *twitter* and *com-friendster*. For the remaining graphs, ApproxKCore is up to 1.85x faster than ExactKCore on average per batch. For the smaller graphs (*com-dblp*, *com-youtube* and *com-orkut*), ApproxKCore is up to 4.55x faster than PLDSOpt and up to 48.29x faster than PLDS on average per batch, because 10^6 is a relatively large batch size for these graphs, so it is faster to re-run our static algorithm compared to our batch-dynamic algorithm. For the larger graphs (except for the graphs where ApproxKCore times out), PLDSOpt achieves up to a 13.09x speedup and PLDS achieves up to a 7.41x speedup against ApproxKCore. We see that among these graphs both PLDSOpt and PLDS performs particularly well in the high-diameter road networks suggesting that the dynamic algorithms perform many less changes to the data structures than the static algorithm for high-diameter (sparser) networks. For the largest graphs (*twitter* and *com-friendster*), PLDSOpt and PLDS are orders of magnitude faster as ApproxKCore times out.

Compared against Hua, PLDSOpt achieves speedups of 6.20–58.66x on all graphs that Hua did not timeout and is orders of magnitude faster on *twitter* and *com-friendster* since Hua times out. Except for the two smallest graphs, *com-dblp* and *wiki-talk-temporal*, PLDS achieves speedups of 1.30–49.02x for all graphs that Hua did not timeout and is orders of magnitude faster on the two largest graphs. On *brain*, *twitter* and *com-friendster*, PLDSOpt achieves 13.83x, 5.82x, and 12.40x speedups, respectively, over PLDS, suggesting the optimization is most beneficial for dense graphs.

We also compared the performance of ApproxKCore and ExactKCore using the full graph on all datasets. We found ApproxKCore to be 1.7–3.9x faster than ExactKCore, with an average coreness error ratio of 1.044–1.172.

Accuracy of Approximation Algorithms. We also computed the average and maximum errors of all of our approximation algorithms for our experiments shown in Fig. 10. The data for these error ratios are shown in **??**. We tested the errors for $\varepsilon = 0.4, \delta = 3$. According to our theoretical proofs, the maximum error (for PLDS) should be $(1.4) * (3) = 4.2$. We see that indeed the maximum empirical error for PLDS falls under this hard constraint. PLDSOpt achieves an average error of 1.359–2.118 on insertion batches (top half) of Fig. 10 compared to errors of 1.593–3.363x for PLDS and 1.010–4.175 for ApproxKCore. For max error, PLDSOpt achieves a max error of 3 (less than the theoretical optimal) compared to 3–4.193 for PLDS and 3–4.305 for ApproxKCore. For *twitter* and

friendster, we sampled error counts uniformly-at-random with probability 1/10 due to our timeout constraints. PLDS and ApproxKCore computations timed out on both of these datasets (even with sampling). We see that decreasing the number of levels improves the error ratio for insertion-only batches.

Space Usage. Finally, we tested the space usage of our parallel batch-dynamic programs against the space usage needed by Hua. We implemented functions that counted the space usage of the entire level data structure used in our programs and the data structures used in the Hua code using the `sizeof` operator. **??** shows the results of our experiment. As expected since our data structures require $O(n \log^2 m + m)$ asymptotic space, our space usage for PLDS is up to 63.87x more and 62.50x more than Hua for *com-dblp* and *com-lj*, respectively. However, our PLDSOpt uses less memory than Hua in most settings for *com-dblp* and up to 1.08x additional space in a few cases; for *com-lj*, it uses up to 1.72x additional space.

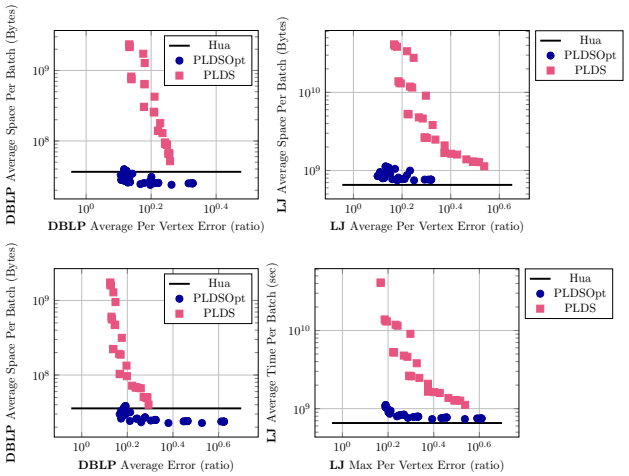

Figure 11: Average space usage in bytes for PLDSOpt, Hua, and PLDS in terms of the average error. We varied ε, δ and computed the error ratio and space usage for the programs on *com-dblp* and *com-lj*. We tested against 10^5 insertion-only (top row) and deletion-only (bottom row) batches for *com-dblp* and 10^6 batch sizes for *com-lj*

.

5.2 Experiments on Deletions

In this subsection, we present the additional experimental results of our LDS, PLDSOpt, PLDS, and ApproxKCore algorithms using the same environment as our insertion-only experiments. To reiterate, the edge updates for the dynamic algorithms are generated by taking a random permutation of a list containing two copies of each edge, where the first appearance of an edge is taken as an insertion, and the second appearance is taken as a deletion. Batches are generated by taking regular intervals of the list. For our experiments on deletion-only batches, we ignore the first appearance of each edge. Moreover, we fix $\epsilon = 0.4$ and $\lambda = 3$; these parameters offer a reasonable tradeoff between approximation error and speed, as shown previously above.

In general, for our batch-dynamic implementations, deletions are faster than insertions; we believe that this is because of the lower computational overhead in computing the desire level for each vertex when processing deletions. Specifically, when computing the new desire levels for vertices while rebalancing insertions, in order to check the requisite invariants, each vertex must maintain its neighbors in the levels below, keyed by level number. This takes more work than the symmetric computation while rebalancing deletions, where each vertex simply maintains the set of neighbors in the levels above and needs not key these neighbors by their level numbers.

Furthermore, our PLDSOpt results in slightly greater error because we decrease the number of levels. This follows from our theoretical result given in Lemma 3.18; namely, decreasing the number of levels affects the lower bound on the approximation and does not affect the upper bound. To see why this is the case, please refer to the of proof of Lemma 3.18 in the full version of our paper [**?**].

Batch Size vs. Running Time. Fig. 8 (bottom row) shows the average deletion-only per-batch running times on varying batch sizes for LDS, PLDSOpt, PLDS, ExactKCore, and ApproxKCore on *com-dblp* and *com-lj*. ExactKCore and ApproxKCore recompute the k-core decomposition on the graph after removing the current batch, as well as all previously deleted edges. We see that PLDSOpt is up to 7.17x over PLDS, PLDS is up to 24.75x faster than LDS, and ApproxKCore is up to 1.59x faster than ExactKCore overall. Compared to using insertion-only batches, PLDS is slower than LDS on smaller batches, and the speedup of PLDS over LDS is also smaller. We believe that this is due to the lower amount of work in the desire level computations for deletion-only batches, where LDS is faster at processing deletion-only batches compared to insertion-only batches. However, the speedups of PLDSOpt and PLDS over Hua are 3.39–44.58x and 1.2–7.89x, respectively, for *com-dblp* and 1.71–45.01x and 1.27–6.59x, respectively, for *com-lj*. The additional speedup of PLDS over Hua is due to our more efficient deletion procedure that does less work than our insertion procedure.

Moreover, for *com-lj*, ExactKCore and ApproxKCore both timeout for the small batch sizes. For *com-dblp*, we see that PLDS is up to 26.61x faster than ApproxKCore for small batch sizes, and even our sequential LDS is up to 50.38x faster than ApproxKCore. For ApproxKCore and ExactKCore, running times are similar to the insertion-only case, except for large batch sizes on *com-dblp*, where the deletion-only batches are faster to process. This is because *com-dblp* is small, with fewer than 3×10^6 edges, and so in the deletion-only setting, we are left with a relatively small graph (or an empty graph in the 10^7 batch size case) after the first batch. On the other hand, in the insertion-only setting, we have almost half of the graph (or the full graph in the 10^7 batch size case) after the first batch. Thus, there is greater processing required overall in the insertion-only setting, for small graphs with large batch sizes.

Thread Count vs. Running Time. Fig. 9 (bottom row) shows the scalability of PLDS with respect to its single-thread running times on *com-dblp* and *com-lj* for deletion-only batches of size 10^6. PLDSOpt achieves up to a 32.02x self-relative speedup and PLDS achieves up to 25.33x self-relative speedup, similar to the speedup obtained in the insertion-only setting. Unlike the case with insertions, Hua achieves no speedup on *com-dblp* and up to a 1.30x self-relative speedup on *com-lj*.

Additional Graphs. Fig. 10 (bottom graph) shows the runtimes of PLDSOpt, PLDS, ExactKCore, and ApproxKCore on the remaining graphs, using deletion-only batches of size 10^6. ExactKCore and ApproxKCore timed out at 3 hours on *twitter* and *com-friendster*.

Table 4: Average and maximum errors of PLDSOpt, PLDS, and ApproxKCore on insertion-only and deletion-only batches of size 10^6. Insertion-only batches errors are shown on the top above the triple horizontal lines and deletion-only batches are shown at the bottom. * indicates that the error was obtained via sampling the error probability with $1/10$ probability. T.O. indicates that the program timed out at 3 hours (even when performing the sampling with $1/10$ probability.

Graph Dataset	PLDSOpt Avg.	PLDSOpt Max	PLDS Avg.	PLDS Max	ApproxKCore Avg.	ApproxKCore Max
com-dblp	1.9345	3	2.635	4	1.15	3.875
brain	1.834	3	3.363	4.193	1.315	4.305
wiki-talk-temporal	1.590	3	1.780	4.172	1.010	3
com-youtube	1.359	3	1.593	4	1.1283	3.75
sx-stackoverflow	1.826	3	2.272	4.067	1.048	3.875
com-lj	1.660	3	2.321	4.175	4.175	1.165
com-orkut	1.926	3	3.115	4.175	1.204	4.2
central-usa	1.601	3	1.683	3	1.374	3
full-usa	1.826	3	1.683	3	1.379	3
twitter	2.118*	3*	T.O.	T.O.	T.O.	T.O.
com-friendster	1.851*	3*	T.O.	T.O.	T.O.	T.O.
com-dblp	1.187	6	1.507	2	1.236	3.0
brain	1.575	6	1.943	4.186	1.315	5.0
wiki-talk-temporal	1.423	4	1.494	4	1.013	3.875
com-youtube	1.268	4	1.317	4	1.137	3.706
sx-stackoverflow	1.630	6	1.792	4.172	1.045	3.908
com-lj	1.613	6	1.704	4.14	1.167	3.984
com-orkut	1.681	6	1.913	4.175	1.205	4.2
central-usa	1.243	3	1.257	3	1.524	3.0
full-usa	1.253	3	1.278	3	1.522	3.0
twitter	1.893*	4*	T.O.	T.O.	T.O.	T.O.
com-friendster	1.685*	3*	T.O.	T.O.	T.O.	T.O.

PLDSOpt is uniformly faster than all other programs for all graphs except *dblp*, achieving a maximum speedup of 23.45x against PLDS, 19.64x against ApproxKCore, and 52.36x against Hua (on all graphs that did not timeout). These results are similar to those obtained for insertions, although the slight speedup could be due to less levels causing deletions to achieve greater speedup than insertions. For *com-youtube* and *com-orkut*, ApproxKCore is up to 1.4x faster than ExactKCore on average per batch. Additionally, for these graphs, ApproxKCore is up to 2.26x faster than PLDS on average per batch, for the same reason as discussed in the main paper (10^6 is a relatively large batch size for these graphs, and so it is faster to re-run our static algorithm compared to our batch-dynamic algorithm). For the larger graphs (*twitter* and *com-friendster*), PLDS is orders of magnitude faster, as ApproxKCore times out.

Accuracy and Space Usage. Table 4 shows a slight increase in the max error for PLDSOpt, with a max error of 6. This is to be expected theoretically; recall the theoretical upper bound on the error is 4.2 (followed by PLDS). PLDS and ApproxKCore do not show noticeable differences in error. Space are somewhat better for deletions than insertions for PLDS. PLDSOpt gets a maximum space usage of 1.07x and 1.69x for *com-dblp* and *com-lj* over Hua, and PLDS a 48.50x and 21.15x factor, respectively, over Hua.

6 Conclusion

We presented a work-efficient parallel batch-dynamic level data structure that gives a $(2+\delta)$-approximate k-core decomposition. Our approach also gave a static approximate k-core algorithm that is to the best of our knowledge the first work-efficient algorithm with polylogarithmic depth for this problem. We studied shared-memory implementations of all of our algorithms and confirmed the practical applicability of our approach. An interesting open problem is to design a batch-dynamic algorithm that is space-efficient (linear space), without incurring additional costs in depth.

A Additional Data Structure Implementations

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

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 partitioned by their levels j where $j < \ell(v)$. Specifically, we maintain a separate dynamic array for each level from level 0 to level $\ell(v)-1$ storing the neighbors of v 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.

The space and depth required to use these data structures is provided in Appendix B.0.2.

$O(n+m)$ 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 $m = o(n \log^2 n)$, 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 at the j'th non-empty level (a non-empty level is one where v has neighbors at that level) that is less than $\ell(v)$. 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 in which 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 precisely the number of elements equal to $2m$.

The space and depth required to use these data structures is provided in Appendix B.0.2.

B Potential Argument for Work Bound

Our work bound uses the potential functions presented in Section 4 of [7]. 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 B.1 (Lemma 3.4 of Main Paper). *For a batch of $\mathcal{B} < m$ updates, Algorithm 1 returns a PLDS that maintains Invariant 1 and Invariant 2 in $O(\mathcal{B} \log^2 m)$ amortized work and $O(\log^2 m \log\log m)$ depth whp, using $O(n \log^2 m + m)$ space.*

B.0.1 Proof of Work Bound Unlike the algorithm presented in [7, 25], 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 [7, 25], 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 movement up or down levels using the potential function argument of [7]. Note that this potential function also gives us the amortized work per edge update since there exists a corresponding set of sequential updates that cannot do less work than the set of parallel updates.

Charging the Cost of Moving Levels The strategy behind our potential function is 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. 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 [7], for some constant $\lambda > 3$:

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

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

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

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 $2\lambda K$. 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 $\text{up}^*(v) < (1+\varepsilon)^{gn(\ell(v)-1)}$. It is moved to a level $\mathsf{dl}(v)$ where its up*-degree is at least $(1 + \varepsilon)^{gn(\mathsf{dl}(v)-1)}$ and its up-degree is at most $(2 + 3/\lambda)\,(1 + \varepsilon)^{gn(\mathsf{dl}(v))}$ (or it is moved to level 0).

The potential before the move is at least

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

since we only move a vertex to a lower level if $\text{up}^*(v) < (1 + \varepsilon)^{gn(\ell(v)-1)}$ 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 $|N(v, Z_{\mathsf{dl}(v)})| < (1 + \varepsilon)^{gn(\mathsf{dl}(v))}$ (otherwise, we could've moved v to level $\mathsf{dl}(v) + 1$). Then, substituting $(1 + \varepsilon)^{gn(i)}$ for all levels $i \geq \mathsf{dl}(v)$ into $\Phi(v)$ allows us to obtain $\sum_{i=\mathsf{dl}(v)}^{\ell(v)-1} (\lambda + 3)(1 + \varepsilon)^{gn(i)}$. 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,

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

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

We need to account for two potential increases: the increase in Ψ and the increase in Φ from neighbors of v. We first consider the increase in Ψ. The potential increase in $\Psi(u, v)$ for every edge (u, v) where $\ell(u) \geq \mathsf{dl}(v)$ is at most $2(\ell(v) - \mathsf{dl}(v))(1+\varepsilon)^{gn(\mathsf{dl}(v))}$, since v has up-degree at most $(1+\varepsilon)^{gn(\mathsf{dl}(v))}$ at level $\mathsf{dl}(v)$ (otherwise, we can increase its $\mathsf{dl}(v)$) and each of these edge's potential gain is upper bounded by 2 for every level in $[\mathsf{dl}(v), \ell(v) - 1]$.

Furthermore, we need to account for the increase in potential of every neighbor whose edge is flipped by the move. The total increase in Φ is at most $\lambda(\ell(v)-\mathsf{dl}(v))(1+\varepsilon)^{gn(\mathsf{dl}(v))}$ for every neighbor on levels $> \mathsf{dl}(v) + 1$, since we move v to the *highest* level that satisfies the invariants and $N(v, \mathsf{dl}(v)) < (1+\varepsilon)^{gn(\mathsf{dl}(v))}$. Decreasing the degree of each neighbor by one for each of $N(v, \mathsf{dl}(v)) < (1+\varepsilon)^{gn(\mathsf{dl}(v))}$ results in the total increase in Φ.

Then, in total, the potential decrease is at least

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

which is enough to pay for the at most $(1+\varepsilon)^{gn(\mathsf{dl}(v))}$ 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 $|N(v, \mathsf{dl}(v)))| < (1+\varepsilon)^{gn(\mathsf{dl}(v))}$. Hence, our number of edge flips is also bounded by $(1+\varepsilon)^{gn(\mathsf{dl}(v))}$.

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 $\mathsf{up}(v) > (2+3/\lambda)(1+\varepsilon)^{gn(i)}$ before the move. Before and after the move, $\Phi(v) = 0$, since in these cases $\mathsf{up}^*(v) > (2+3/\lambda)(1+\varepsilon)^{gn(i-1)}$ and $\mathsf{up}^*(v) > (2+3/\lambda)(1+\varepsilon)^{gn(i)}$, 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 $\Psi(v, w)$.

Let Z_i be 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 B.2. *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 B.3. *For a batch of $\mathcal{B} < m$ updates, Algorithm 1 requires $O(\mathcal{B}\log^2 m)$ amortized work with high probability. The required space is $O(n\log^2 m + m)$ 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 $O(\log^2 m)$ amortized work per edge update to the structure.

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 to 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 $O(\log^2 m)$. □

B.0.2 Overall Work and Depth Bounds Our deterministic and space-efficient structures also give the following corollary using our above work-bound arguments.

Corollary B.4. *There exists a set of data structures where Algorithm 1 requires $O(\log^2 m)$ amortized work per update, deterministically, using $O(m+n)$ space.*

Using Corollary 3.12 and Lemma B.3, we obtain Lemma B.1. Similarly, combining Lemma 3.14 and Corollary B.4 and Corollary 3.13 and Corollary B.4, we obtain the following two corollaries.

Corollary B.5. *For a batch of $\mathcal{B} < m$ updates, Algorithm 1 returns a PLDS that maintains Invariant 1 and Invariant 2 in $O(\mathcal{B}\log^2 m)$ amortized work and $O(\log^3 m)$ worst-case depth, using $O(n\log^2 m + m)$ space.*

Corollary B.6. *For a batch of $\mathcal{B} < m$ updates, Algorithm 1 returns a PLDS that maintains Invariant 1 and Invariant 2 in $O(\mathcal{B}\log^2 m)$ amortized work and $O(\log^4 m)$ worst-case depth, using $O(n+m)$ space.*

C 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.

Because we have $O(\log^2 m)$ levels in our data structure, we rebuild the data structure once we have made $m/2$ edge updates (including edge updates from edges incident to deleted vertices). Rebuilding the data structure requires $O(m\log^2 n)$ total work which we can amortize to the $m/2$ edge updates to obtain $O(\log^2 n)$ amortized work *whp*. Running Algorithm 2 and Algorithm 3 on the entire set of $O(m)$ edges requires $O(\text{poly}\log n)$ depth *whp*depending on the specific set of data structures we use.

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

References

[1] 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 *Proceedings of the 18th International Conference on Neural Information Processing Systems*.

[2] 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.

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

[4] Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. 2016. Distributed K-Core Decomposition and Maintenance in Large Dynamic Graphs. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*. 161–168.

[5] Leonid Barenboim and Michael Elkin. 2010. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. *Distributed Comput.* 22, 5-6 (2010), 363–379.

[6] Vladimir Batagelj and Matjaž Zaveršnik. 2011. Fast algorithms for determining (generalized) core groups in social networks. *Advances in Data Analysis and Classification* 5, 2 (2011), 129–145.

[7] 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)*. 173âĂŞ182.

[8] Kshipra Bhawalkar, Jon Kleinberg, Kevin Lewi, Tim Roughgarden, and Aneesh Sharma. 2015. Preventing Unraveling in Social Networks: The Anchored k-Core Problem. *SIAM Journal on Discrete Mathematics* 29, 3 (2015), 1452–1475.

[9] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

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

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

[12] N. S. Dasari, R. Desh, and M. Zubair. 2014. ParK: An efficient algorithm for k-core decomposition on multicore processors. In *IEEE International Conference on Big Data*. 9–16.

[13] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. 2008. *Implementation Challenge for Shortest Paths.* Springer US, Boston, MA, 395–398. https://doi.org/10.1007/978-0-387-30162-4_181

[14] 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.

[15] 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)*.

[16] B. Elser and A. Montresor. 2013. An evaluation study of BigData frameworks for graph processing. In *IEEE International Conference on Big Data*. 60–67.

[17] David Eppstein, Maarten Löffler, and Darren Strash. 2010. Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time. In *Algorithms and Computation*. 403–414.

[18] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. 2018. Parallel and Streaming Algorithms for K-Core Decomposition. In *Proceedings of the 35th International Conference on Machine Learning*. 1397–1406.

[19] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrović. 2019. Improved Parallel Algorithms for Density-Based Network Clustering. In *Proceedings of the 36th International Conference on Machine Learning*. 2201–2210.

[20] J. Gil, Y. Matias, and U. Vishkin. 1991. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 698–710.

[21] Michael T. Goodrich and Paweł Pszona. 2011. External-Memory Network Analysis Algorithms for Naturally Sparse Graphs. In *European Symposium on Algorithms*. 664âĂŞ676.

[22] 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.

[23] 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.

[24] John Healy, Jeannette Janssen, Evangelos Milios, and William Aiello. 2007. *Characterization of Graphs Using Degree Cores.* 137–148.

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

[26] 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.

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

[28] 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.

[29] 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.

[30] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media? 591–600.

[31] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. (2014).

[32] 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.

[33] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at What COST?. In *USENIX Conference on Hot Topics in Operating Systems (HotOS)*.

[34] Amir Mehrafsa, Sean Chester, and Alex Thomo. 2020. Vectorising K-Core Decomposition for GPU Acceleration. In *International Conference on Scientific and Statistical Database Management*.

[35] 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*. 815âĂŞ824.

[36] A. Montresor, F. De Pellegrini, and D. Miorandi. 2013. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems* 24, 2 (2013), 288–300.

[37] K. Pechlivanidou, D. Katsaros, and L. Tassiulas. 2014. MapReduce-Based Distributed K-Shell Decomposition for Online Social Networks. In *IEEE World Congress on Services*. 30–37.

[38] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com

[39] Ahmet Erdem SarÄśyüce, 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.

[40] 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.

[41] Shay Solomon and Nicole Wein. 2020. Improved Dynamic Graph Coloring. *ACM Trans. Algorithms* 16, 3, Article 41 (June 2020).

[42] 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).

[43] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *PVLDB* 13, 2 (2019), 211–225.

[44] A. Tripathy, F. Hohman, D. H. Chau, and O. Green. 2018. Scalable K-Core Decomposition for Static Graphs Using a Dynamic Graph Data Structure. In *IEEE International Conference on Big Data (Big Data)*. 1134–1141.

[45] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-Efficient and Practical Parallel DBSCAN. In *ACM SIGMOD*.

[46] 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.

[47] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *IEEE 33rd International Conference on Data Engineering (ICDE)*. 337–348.