Parallel k-Core Decomposition with Batched Updates and Asynchronous Reads

Quanquan C. Liu
Yale University
USA
quanquan.liu@yale.edu

Julian Shun
MIT CSAIL
USA
jshun@mit.edu

Igor Zablotchi
Mysten Labs
Switzerland
igor@mystenlabs.com

Abstract

Maintaining a dynamic k-core decomposition is an important problem that identifies dense subgraphs in dynamically changing graphs. Recent work by Liu et al. [SPAA 2022] presents a parallel batch-dynamic algorithm for maintaining an approximate k-core decomposition. In their solution, both reads and updates need to be batched, and therefore each type of operation can incur high latency waiting for the other type to finish. To tackle most real-world workloads, which are dominated by reads, this paper presents a novel hybrid concurrent-parallel dynamic k-core data structure where asynchronous reads can proceed concurrently with batches of updates, leading to significantly lower read latencies. Our approach is based on tracking causal dependencies between updates, so that causally related groups of updates appear atomic to concurrent readers. Our data structure guarantees linearizability and liveness for both reads and updates, and maintains the same approximation guarantees as prior work. Our experimental evaluation on a 30-core machine shows that our approach reduces read latency by orders of magnitude compared to the batch-dynamic algorithm, up to a $(4.05 \cdot 10^5)$-factor. Compared to an unsynchronized (non-linearizable) baseline, our read latency overhead is only up to a 3.21-factor greater, while improving accuracy of coreness estimates by up to a factor of 52.7.

CCS Concepts: • **Theory of computation** → **Concurrent algorithms**; **Dynamic graph algorithms**; • **Computing methodologies** → **Parallel algorithms**.

Keywords: parallelism, concurrency, k-core decomposition

1 Introduction

The discovery of underlying structure in large-scale networks poses a fundamental challenge in various computing domains. One crucial aspect involves identifying communities within the network where individuals or vertices share strong connections, as well as understanding the level of connectivity of each individual to their respective community. The notion of a k-core, or more generally, k-core decomposition, effectively captures the well-connectedness of a vertex or group of vertices. Consequently, this problem and its variations have received extensive attention across machine learning [8, 33, 40], database [17, 21, 32, 54, 63], social network analysis, graph analytics [25, 26, 49, 50], computational biology [22, 51, 59, 61], and other relevant communities [39, 50, 60, 67].

Given an undirected graph G with n vertices and m edges, the k-core of the graph represents the largest subgraph $H \subseteq G$ in which every vertex in H has a degree of at least k. The k-core decomposition of the graph refers to a partition of the graph into layers, where a vertex v is placed in layer k if it belongs to a k-core but not a $(k+1)$-core. This layering process assigns a *coreness* value to each vertex based on the largest k-core that it belongs to, leading to a natural hierarchical clustering.

Traditional algorithms that give exact solutions to k-core decomposition inherently follow a sequential approach [62]. In fact, k-core decomposition is known to be a P-complete problem [11], so efficient parallel algorithms that solve it exactly are unlikely to exist. To overcome this limit, we focus on achieving a close approximate decomposition, which provides utility in areas where existing methods focus mostly on approximations, such as epidemiology [22, 51, 59, 61], community detection and network centrality measures [30, 34, 42, 64, 72, 76], network visualization and modeling [8, 19, 75, 77], protein interactions [7, 13], and clustering [41, 53].

Current emphasis has also been on addressing the *dynamic* nature of large networks. Networks undergo frequent updates which require real-time k-core computations for various applications. Significant progress has been made on dynamic k-core algorithms in both sequential [55, 56, 68, 70, 74, 78] and parallel settings [12, 46, 48] to achieve fast, practical solutions.

Recent work by Liu et al. has studied k-core decomposition in the parallel *batch-dynamic* setting, where operations proceed in batches and there is global synchronization between different batches [57]. Each batch consists of exactly one type of operation—reads, insertions, or deletions. However, a key

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03.
https://doi.org/10.1145/3627535.3638508

challenge arises: querying the system state has high latency, as reads cannot safely proceed concurrently with update batches. Unsynchronized reads, concurrent with updates, may not only lead to hard-to-interpret non-linearizable results, but can also break the approximation bounds of the k-core algorithm (in fact, the error could be unbounded, as we show later). Thus, reads in current parallel batch-dynamic algorithms must either wait for updates to finish, or be performed synchronously as part of the batch, both adding latency. This is problematic for applications that require low read latency. Examples include social networks and search engines: these need to be very responsive on the dominant user-facing read path [18, 20], while prioritizing throughput on the update path.

In this paper, we address this gap by proposing a novel k-core algorithm in which reading a vertex's coreness can proceed asynchronously and concurrently with (batches of) updates and with other reads. We achieve this by tracking causal dependencies between updates and reads. We show that such dependencies can be tracked efficiently, without locking, and without sacrificing the performance of updates.

Our algorithm, similar to previous work, relies on the Level Data Structure (LDS) approach. The core idea behind the LDS approach is that the k-core decomposition of a graph can be represented as a sequence of levels. These levels are organized into groups, where vertices within each group share the same coreness (within the approximation factor). The LDS serves as a data structure that maintains the levels of all vertices, gets updated when the graph undergoes edge insertions or removals, and facilitates queries regarding vertex coreness.

The main challenge in designing our algorithm is achieving atomic reads that can proceed concurrently with batches of updates while incurring low overhead. In brief, this challenge arises because reads might need to be atomic with respect with, and thus synchronize with, a potentially large number of concurrent updates. This might seem at first counter-intuitive.

At first glance, it may seem as though a read of vertex v only needs to be synchronize with updates to edges incident to v. However, the situation is more intricate: an update, say an insertion of edge e, may not only cause changes in the levels of vertices incident to e, but can also trigger a chain effect of vertices moving levels inside the LDS. All of these level changes are causally dependent on the initial update and therefore must appear to reads to take place atomically. Furthermore, it is possible for vertex level changes to collectively result from multiple edge updates, necessitating that all of these updates appear atomic to reads.

We aim for lock-free reads. Lock-freedom has the benefit of guaranteeing that the system always makes progress, even if some processes are slow, but it comes with the challenge of precluding simple solutions based on locking. We also aim for our updates to complete in a finite number of steps. Due to technical reasons which we explain in Section 2, our updates cannot be said to be lock-free, and so we use the term *live* instead.

To overcome these challenges, we propose a solution that involves tracking causal dependencies through Directed Acyclic Graphs (DAGs) of operation descriptors. In essence, this works as follows. During each update batch, each vertex v that needs to change levels in the LDS is associated with an operation descriptor containing information about which vertices that moved earlier in the batch caused v to also have to move. This creates a DAG of operation descriptors. Readers that encounter a vertex v with an active descriptor need to first establish whether v, and the transitive closure of v's causal dependencies (as tracked by the DAG), are still in the process of being updated. If they are, the read must return the old level of v, since the new, final level might not be known yet. Otherwise, if the update process is complete, the read operation can safely return the new level.

We call our data structure the ***concurrent parallel level data structure (CPLDS)***. We implement our data structure in C++ using the GBBS [27] and ParlayLib [16] libraries and conduct an experimental evaluation of our algorithm on a 30-core machine. Our evaluation shows that, compared to the batch-dynamic algorithm of Liu et al. [57], adding asynchronous reads only increases the update time by a factor of at most 1.48, while decreasing the read latency by a factor of up to $4.05 \cdot 10^5$. We also compare to an unsynchronized (non-linearizable) baseline, and show that our read latency is only up to 3.21x slower, while returning coreness estimates that are up to 52.7x more accurate.

2 Preliminaries

We study undirected and unweighted graphs in this paper, and use n to denote the number of vertices and m to denote the number of edges in a graph. We assume each vertex is represented by a unique integer in $[0, \ldots, n-1]$. We study the k-core decomposition problem, which is defined below.

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

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

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

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

In the parallel *batch-dynamic* setting, algorithms process operations in batches, with each batch consisting of exactly one type of operation—reads, edge insertions, or edge deletions.[1] In this paper, we study a hybrid setting, where reads are asynchronous and can execute at any time, while updates are batched and executed together periodically. This solves the latency issue for read operations, which are the dominant type of operation in most workloads, e.g., in social networks [18, 20].

In theory, it would be desirable to make updates asychronous as well, but it is much more challenging to do so while guaranteeing linearizability. We leave this to future work. Below, we introduce our model more formally.

We consider a set of P processes that communicate through standard shared-memory primitives. The processes coordinate to maintain the graph G and G's associated CPLDS data structure by serving incoming operations. Operations on the CPLDS can be either reads or updates. A read operation takes an input node and returns its coreness estimate in the CPLDS. An update operation can be either an edge insertion or an edge deletion. It adds or removes an input edge e to/from G and updates the (levels of vertices in the) CPLDS accordingly.

The set of processes can be partitioned into a set of update processes, which only perform updates, and a set of read processes, which only perform reads. Updates are performed in batches by the update processes. We assume in this paper that each batch consists either of only insertions or only deletions (in practice, batches contain a mix of insertions and deletions, which are separated into insertion and deletion sub-batches during pre-processing). The updates in each batch are executed collectively and in parallel by the updating processes. The steps required to execute all updates in a batch are pooled together for efficient parallel execution. In other words, it is not the case that each update is executed by a single process; instead, all update processes collectively execute each batch. Reads are performed by the read processes asynchronously and concurrently to batches of updates. In contrast to updates, reads are not executed in batches, but individually. Each read is performed by a single process from beginning to end. Such process separation may be employed by applications with different flows for reads and updates, e.g., in which reads access data directly, while updates modify several internal data structures.

Our timing assumptions are as follows: (1) update processes are synchronous, meaning that their computation and communication delays are bounded by a known constant, and (2) read processes are asynchronous, meaning that they can be arbitrarily delayed, without any upper bound on the delay. We do not consider process failures in this work.

[1]We focus on edge updates for simplicity, but most batch-dynamic solutions can be modified to support vertex updates as well.

In terms of safety, our algorithms satisfy *linearizability* (also called atomicity). Essentially, linearizability requires that each operation (read or update) appears to take effect instantaneously at a moment in time that falls between that operation's invocation and response.

In terms of liveness, our algorithms guarantee that reads are *lock-free*: if reads are invoked infinitely often, then some operation in the system terminates in a finite number of steps, infinitely often [44]. Furthermore, our algorithms guarantee that each update terminates in a finite number of steps. However, since our updates are executed on synchronous processes that do not fail, they cannot be said to be lock-free, so we instead say that updates are *live*.

3 Background

This section presents background information on the sequential and parallel level data structures that our approach is based on.

3.1 Level Data Structure (LDS)

The sequential level data structure of Bhattacharya et al. [15] and Henzinger et al. [43] combined with the proof given by Liu et al. [57] maintains a $(2+\varepsilon)$-approximate coreness value for each vertex in the graph for any constant $\varepsilon > 0$.

The LDS partitions the vertices of G into $K = O(\log^2 n)$ *levels*, $0, \ldots, K-1$. The levels are partitioned into equal-sized *groups* of contiguous levels. There are $O(\log n)$ groups and each group g_i has $O(\log n)$ levels. We denote the level of a vertex v by $\ell(v)$.

Whenever an edge is inserted into or removed from the graph, one or more vertices may change their level, and thus the LDS must also be updated. This proceeds as follows. After each edge update, vertices update their levels based on whether or not they satisfy two invariants (these invariants are explained below). If a vertex v violates one of the invariants, it must move up or down one level in the LDS, and then re-check the invariants; we repeat this process for every vertex v until all vertices satisfy both invariants.

It is important to note that each time a vertex changes levels, this may cause other vertices to violate one of the invariants and thus have to move as well. Thus, every vertex level change may potentially trigger a cascading effect of other vertices changing levels.

LDS Invariants. The first invariant upper bounds the induced degree of a vertex v in the subgraph of all vertices at v's level or above. If a vertex v violates the first invariant, v must move up (at least) one level. The second invariant lower bounds the induced degree of a vertex v in the subgraph consisting of the level below v, the level of v, and all levels above v. If a vertex v violates the second invariant, it must move down (at least) one level. It is important to note that inserting more edges into the graph may only cause

vertices to violate the first invariant, but not the second; similarly, deleting edges from the graph may only cause vertices to violate the second invariant, but not the first.

We now give the invariants in more technical detail. For each level $\ell = 0, \ldots, K-1$, let V_ℓ be the set of vertices currently in level ℓ. Let Z_l be the set of vertices in levels greater or equal to ℓ. Let $\delta > 0$ and $\lambda > 0$ be two constants. Let $g_0, ..., g_{\lceil \log_{(1+\delta)} n \rceil}$ be the groups into which the K levels are partitioned.

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

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

3.2 Parallel LDS (PLDS)

The Parallel LDS (PLDS) algorithm of Liu et al. [57] is a parallel batch-dynamic LDS algorithm. It improves upon the original LDS algorithm by observing that (1) in many cases, vertices can be updated in parallel (instead of sequentially) and (2) if the vertices are updated in a carefully chosen order, the number of times a given vertex needs to be processed can be significantly reduced.

In the PLDS algorithm, updates arrive in batches. During the execution of a batch, updates are partitioned into insertions and deletions; thus each batch has an insertion phase and a deletion phase.

During the insertion phase, levels are visited in increasing order (starting with level 0). The vertices in each level are checked in parallel against Invariant 1 and moved up one level if necessary. The algorithm ensures that each level needs to be visited at most once during the insertion phase: after vertices move up from level ℓ, no future step in the current batch moves a vertex up from level ℓ. Note that a vertex can move up many levels, one level at a time.

During the deletion phase, each vertex that violates Invariant 2 computes its ***desire level***, which is the highest level below its current level where it satisfies Invariant 2. Levels are visited in increasing order, and when processing level ℓ, all vertices with a desire level of ℓ move there. Their neighbors at higher levels will then recompute their desire levels. The algorithm ensures that a vertex will never need to move again once it is moved to its desire level, and that no vertices will want to move to a level $\leq \ell$ after processing level ℓ.

Coreness Approximation. The $(2+\epsilon)$-approximate coreness $\hat{k}(v)$ of a vertex v is computed as in Definition 3.1.

Definition 3.1 (Coreness Estimate)**.** The *coreness estimate* $\hat{k}(v)$ of vertex v is $(1+\delta)^{\max(\lfloor (\ell(v)+1)/4\lceil \log_{1+\delta} n \rceil \rfloor - 1, 0)}$, where each group has $4\lceil \log_{(1+\delta)} n \rceil$ levels.

The following lemma by Liu et al. [57] proves the $(2+\epsilon)$-approximation for coreness values.

Lemma 3.2. *Let $\hat{k}(v)$ be the coreness estimate and $k(v)$ be the coreness of v, respectively. If $k(v) > (2+3/\lambda)(1+\delta)^{g'}$, then $\hat{k}(v) \geq (1+\delta)^{g'}$. Otherwise, if $k(v) < \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$, then $\hat{k}(v) < (1+\delta)^{g'}$.*

4 Algorithm Overview

To ensure linearizability, a basic challenge that our algorithm needs to solve is to avoid returning intermediate values: a read of some vertex v's level, that is concurrent with an update to the level of v, should either return v's pre-update level (its old level), or v's post-update level (its new level), but not any intermediate level between the old and new levels.

A first and naive version of our algorithm that addresses this challenge is as follows: we use *operation descriptors* to synchronize between updates and reads.[2] If a vertex v has an active operation descriptor, this signals to concurrent reads that v is in the process of changing levels in the CPLDS. Essentially, if a read of v finds that v is *marked* with an active descriptor, the read must return the old level of v, before v started changing levels in the current batch. This is because the final level of v might not yet be known, and returning an intermediate level for v (in between its old and new levels) would violate linearizability. Thus, v's operation descriptor records the old level of v.

However, this first algorithm does not solve another challenge required by linearizability: avoiding new-old inversions among causally dependent vertices. Consider two vertices u and v, such that u's level change (which is triggered by an update) causes v to now violate one of the LDS invariants and to also have to change levels. In any sequential execution, the update that moves u also moves v, so no read can observe the old level of v after some read has already observed the new level of u, or vice-versa. However, our first algorithm allows such new-old inversions in concurrent executions: if u is marked but v is not yet (or no longer) marked, then a pair of reads might return the new level of v (since v is not marked) and then the old level of u (since u is marked).

Therefore, it is not sufficient for a read of v to synchronize with level changes of v alone. Such a read must also synchronize with level changes of v's causally dependent vertices. In fact, it must synchronize with the entire transitive closure of vertices that may have caused v to move or which v may have caused to move. As in the LDS and PLDS algorithms, in our algorithm it is possible for updates to create dependency chains among vertices: an update causes a node v to change levels, which causes one or more of v's neighbors to violate the invariants and have to change levels, which may cause their neighbors in turn to change levels, and so on. We represent these causal dependencies as a Directed Acyclic Graph (DAG): in such a DAG, there is an edge $v \rightarrow u$

[2] Note that updates do not synchronize with each other through the operation descriptors; instead, they are synchronized as part of the batch-dynamic parallel execution.

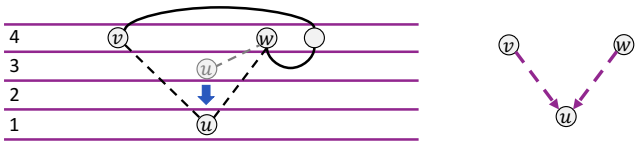


Figure 1. A PLDS and a dependency DAG in which v's and w's level changes are indirectly caused by the level change of u. In any sequential execution, the operation that causes the level of u to change also changes the levels of v and w. Thus, it is impossible in any sequential execution for a read to return the old level of u, v, or w after another read has already returned the new level of one of these vertices. To ensure linearizability, our algorithm must therefore guarantee that level changes to vertices in the same DAG appear to take effect atomically to concurrent readers.

if u's level change caused v to also have to change level. If v has no such outgoing edge, we call v a *root* (this occurs if v moves only as a direct result of an edge update, as opposed to moving as a result of one of its neighbors in G moving).

The set of vertices that move during a batch can thus be partitioned into dependency DAGs. To avoid new-old inversions, our algorithm must ensure that the level changes of all vertices within a DAG appear to concurrent readers to take effect atomically; we call this the *DAG atomicity rule*. An example is shown in Fig. 1.

We enforce the DAG atomicity rule by maintaining the invariant that each DAG has a single root, and rely on an atomic operation on this single root to linearize the level changes of all vertices in the DAG. To ensure that each DAG has a single root, we do the following: whenever a DAG has more than one root, we deterministically pick one of them as the sole root, and make the others point to the sole root.

Even though the dependency graph is a DAG, in our algorithm we do not need to materialize the entire DAG (i.e., store all of the dependencies). In fact, we only require that we can reach the root of a DAG from any vertex in the DAG. Thus, it is sufficient to store a single *parent* for each vertex in the DAG. Whenever we create an operation descriptor for some vertex v (we say that v becomes *marked*), we include in the descriptor a pointer to v's parent in the DAG. By traversing these parent pointers we will reach the root from any vertex in a finite number of steps. Therefore, we only materialize a subtree of each DAG. However, we continue using the DAG terminology in this paper.

We now describe the high-level changes our CPLDS data structure introduces with respect to PLDS:

1. When a vertex v becomes marked during a batch of updates, we create an operation descriptor for v and populate it with v's old (pre-update) level and parent.
2. At the end of each batch, we unmark all marked nodes by deleting all operation descriptors. We first unmark the root of each DAG, and then unmark all non-root vertices.

Algorithm 1. Data structures and global variables

```
1  struct Descriptor:
2      // a pointer to this node's parent in the dependency DAG
3      int parent
4      // this node's level before the current batch of updates
5      int old_level

7  // global variables
8  Descriptor desc_array[num_vertices]
9  int batch_number = 0 // incremented at the start of every batch
```

3. A read of vertex v examines v's operation descriptor (if any): if v is marked and its root is also marked, the read returns the coreness estimate using v's old level (as recorded in v's descriptor); otherwise, the read returns the coreness estimate using v's current level, which we call its ***live level***.

In the next section, we describe our algorithm in more technical detail.

5 Detailed Algorithm

5.1 Data Structures and Global State

Algorithm 1 shows the `Descriptor` data structure; it may be in one of two states at any given time. If the `Descriptor` has the special value `UNMARKED`, then we say that v and its descriptor are *unmarked*, which means that v is not currently in the process of changing levels in the CPLDS. Otherwise, we say that v and its descriptor are *marked*, and thus v is in the process of changing its level. A marked descriptor has two fields: `parent` and `old_level`. The `parent` field contains the index of v's parent node, or the special value `I_AM_ROOT` if v has no parent because v is the root of its DAG.

We maintain a global array `desc_array` of `Descriptors`, one per vertex in the graph, for the lifetime of the program. As part of our global state, we also maintain a variable `batch_number`, which is incremented at the start of each batch.

5.2 Updates

Our update algorithm executes each batch $\mathcal{B}$ as follows; we show an example in Fig. 2. First, we insert into, or delete from, G all of the edges in $\mathcal{B}$. Then, we traverse the CPLDS level by level and update the levels of the vertices impacted by the edge updates of $\mathcal{B}$. Whenever we detect that a vertex violates one of the invariants, we mark it as described below, and move it up or down one or more levels in the CPLDS. This is done in parallel for all vertices on a given level in the CPLDS. After we have done this for every level in the CPLDS, we finalize the batch by unmarking all marked vertices (described below).

Marking. Whenever a node v becomes marked, we call the `mark` function (shown in Algorithm 2) and pass in v's index in `desc_array`, as well as an array containing the indices of v's *triggers*. A vertex u is a trigger for v if u may have contributed to v becoming marked during the current batch. In the case of insertions, the set of triggers contains all marked neighbors

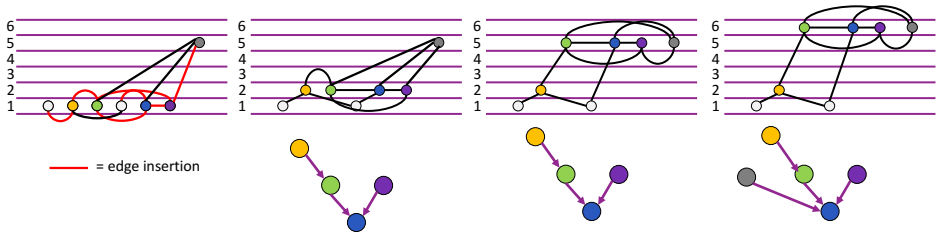


Figure 2. The insertion batch is shown in red. The batch causes the yellow, green, blue, and purple vertices to move up one level with the created dependency DAG shown below. Then, the green, blue and purple vertices continue moving up the levels. Finally, the green, blue, and purple vertices cause the gray vertex to move up a level. Since the green, blue, and purple vertices are all in the same dependency DAG, the gray vertex points to the root (the blue vertex).

Algorithm 2. Update algorithm: marking and unmarking

```
1  mark(int v, int triggers[]):
2      desc = new Descriptor
3      desc.old_level = LDS.get_level(v)
4      marked_batch_neighbors = [w for (v,w) in the batch B and w
           ↪ is marked]
5      for w in (marked_batch_neighbors + triggers):
6          union(v,w)
7      desc_array[v] = desc

9  // this is called at end of batch
10 unmark_all():
11     // unmark all roots
12     parfor all nodes v such that desc_array[v] != UNMARKED
           and desc_array[v].root == I_AM_ROOT :
13         desc_array[v] = UNMARKED
14     // unmark all other marked nodes
15     parfor all nodes v such that desc_array[v] != UNMARKED :
16         desc_array[v] = UNMARKED
```

of v at the same level or higher level as v in the CPLDS. (A vertex which was at a lower level than v earlier in the batch but moved higher than v could become a trigger later.) In the case of deletions, the set of triggers contains all marked neighbors of v at any level lower than $\ell(v) - 1$'s level.

In the `mark` function, we first create a new descriptor for v and populate its `old_level` field with v's current level, before v moves (Lines 2–3). We then determine the set of DAGs into which v will be merged. These are: (1) the set of DAGs of v's triggers and (2) the set of DAGs of v's *marked batch neighbors* (Line 4). A vertex w is a marked batch neighbor of v if the edge (v, w) is updated during $\mathcal{B}$ and w is already marked when we mark v. We merge v into its marked batch neighbors' DAGs to ensure that no updated edge has its endpoints in different DAGs—this is necessary for correctness (see Section 6).

Next, we merge the DAGs determined in the previous steps and add v to the merged DAG (Lines 5–6). Care must be taken here regarding synchronization, as multiple threads that are marking vertices in parallel might merge overlapping sets of DAGs at the same time. In fact, this step is very similar to the *union* operation in concurrent union-find implementations [6, 28, 45, 47]. For conciseness, we reuse the union implementation described in [47] and implemented in [28], and denote it as `union` (Line 6).

Unmarking. Unmarking, shown in Algorithm 2, is done by overwriting the contents of a vertex v's descriptor with the special `UNMARKED` value. We first unmark all DAG roots (Lines 12–13), and then unmark all other nodes (Lines 15–16).

By unmarking root descriptors first, we maintain the following invariant: for each DAG, the root descriptor is marked before non-root descriptors in the same DAG are marked, and is unmarked before non-root descriptors in the same DAG are unmarked.

Optimization: Path Compression. In our algorithm, we do not need to materialize DAGs fully; instead, each vertex v points directly to the root of its DAG *as it was at the moment when v was added to the DAG*. However, due to our DAG merging mechanism in Algorithm 2, it is possible for the path from v to the true root of v's DAG to become more than one hop long. This is both unnecessary and inefficient, as traversing several hops to reach the root may impact performance. Therefore, as an optimization, when doing reads or updates, we perform *path compression* when traversing the path from a vertex to its root: if this path is longer than one hop, at the end of the traversal, we overwrite v's `parent` field, as well as the `parent` field of all of v's ancestors that we traversed, to point to the root. This optimization is a standard optimization in union-find algorithms and is done in the union-find implementation that we use [28].

5.3 Reads

We start with Algorithm 3, which contains the helper function `check_DAG`. This function takes a vertex v's descriptor D and determines whether D is part of a marked DAG. The basic logic of `check_DAG` is as follows: we traverse D's DAG until we reach the root: if the root is marked, return `MARKED`; otherwise return `UNMARKED`. We also perform path compression for reads, and thus this is the same logic as the *find* operation in union-find algorithms (not shown in the pseudocode). However, instead of traversing to the root every time, we implement the following optimization which enables us to return early from `check_DAG` in some cases. If we encounter any unmarked descriptor along the way, including D itself, we can return `UNMARKED` immediately, without continuing to the root. This is due to the invariant described above: if any non-root descriptor in a DAG is unmarked, it must be the case that the DAG's root has also been unmarked.

Algorithm 3. check_DAG helper function

```
1  // returns whether the DAG that includes desc is marked or
       ↪ unmarked
2  check_DAG(Descriptor desc):
3      // if v's descriptor is marked we can return directly
4      if (desc == UNMARKED):
5          return UNMARKED

7      // otherwise, traverse to the root of v's DAG
8      while (desc.parent != I_AM_ROOT):
9          desc = desc.parent
10         // if we encounter an unmarked descriptor on the path to
               ↪ the root, we can return directly
11         if (desc == UNMARKED):
12             return UNMARKED

14     // return whether the root is MARKED or UNMARKED
15     if (desc == UNMARKED):
16         return UNMARKED
17     return MARKED
```

Algorithm 4. Read algorithm

```
1  // returns the level of the vertex with index v
2  read(int v):
3  retry:
4      b1 = batch_number
5      l1 = LDS.get_level(v)
6      desc = desc_array[v]
7      status = check_DAG(desc)
8      l2 = LDS.get_level(v)
9      b2 = batch_number
10     if (b1 != b2):
11         goto retry
12     else if status == MARKED:
13         return coreness estimate using desc.old_level
14     else: // status was UNMARKED
15         if (l1 == l2):
16             return coreness estimate using l1
17         else:
18             goto retry
```

Path compression is done on the path up to the unmarked node that we find.

We now describe the main read algorithm, whose pseudocode is in Algorithm 4. Essentially, the logic of a read of vertex v is as follows: (1) read v's live level and descriptor (Lines 5–6); (2) determine if v's root is marked (Line 7); (3) if it is, then return v's old level from its descriptor (Line 13); otherwise, return v's live level from step (1) (Line 16). However, we require additional logic to ensure linearizability.

First, we "sandwich" steps (1) and (2) above between two reads of the batch number (Lines 4 and 9). We repeat steps (1) and (2) until the two batch numbers match, meaning that the steps occurred within the same batch. Otherwise, the read logic might observe a mix of states from different batches and thus return non-linearizable results.

Furthermore, we sandwich step (2) in between two reads of the v's live level (Lines 5 and 8); in case v is unmarked (and thus the read returns the live level), these two reads must match. If we only performed one such read of the live level, this would enable a scenario in which the read returns an intermediate level of v, in between v's old and new levels, which would not be linearizable.

6 Correctness

We prove the linearizability and liveness of our algorithm in the full version of our paper [58]. In short, we prove

Theorem 6.1. *Our algorithm is linearizable, and live: updates terminate in a finite number of steps and reads are lock-free.*

6.1 Approximation Guarantees

The level that a reader uses to compute the coreness estimate will correspond to the level of the vertex during some point in time in between update batches. This is because when a reader returns a coreness estimate, it never sees an intermediate level of the vertex (it uses the level either at the beginning of a batch or at the end of it). Therefore, when compared to the true coreness value of the vertex at a point in time between two consecutive update batches, we maintain the $(2+\epsilon)$-approximation guarantee as in the algorithm by Liu et al. [57].

Note that using unsynchronized reads can return coreness values of vertices using intermediate levels within a batch, and the error can be unbounded with respect to the true coreness values at both the beginning and the end of the batch. For example, consider a batch of insertions that causes a vertex v to move up from group g to group $g+i$, for $i = O(\log_{1+\delta} n)$ (there are $\log_{1+\delta} n$ groups in the level data structure). An unsynchronized read can see the vertex v in any group in $[g, \ldots, g+i]$. In the worst case, we return the coreness estimate of v at group $g + i/2$. According to Definition 3.1, this will increase the error by a multiplicative factor of $(1+\delta)^{i/2} = O(\sqrt{n})$ relative to the guarantee in Lemma 3.2, no matter whether we compare to the ground truth at the beginning or at the end of the batch.

7 Experimental Evaluation

In this section, we implement our algorithm and test it against various baselines to determine the latency, throughput, and accuracy of our reads and updates. We implement our algorithms on top of the parallel level data structure (PLDS) in Liu et al. [57] which uses the Graph Based Benchmark Suite (GBBS) [27]. Our results show that our algorithms decreases the latency of reads compared to synchronous implementations by up to *five orders of magnitude*.

Evaluated Algorithms. We compare our CPLDS against two baseline algorithms that we also implement. First, we compare our CPLDS against a synchronous implementation (SyncReads) where all reads must wait until all updates are performed in the batch before the reads can be performed. We also compare against a non-synchronous version (NonSync) of our algorithm where reads can be done at *any time* in the batch. This algorithm is not linearizable. We obtain *orders-of-magnitude improvements* on the accuracy of our reads against the non-linearizable (NonSync) implementation and on the latency against the synchronous (SyncReads) algorithm.

Experimental Setup. We use a `c2-standard-60` Google

Graph Dataset	Num. Vertices	Num. Edges	Largest value of k
dblp	317,080	1,049,866	113
brain	784,262	267,844,669	1200
wiki	1,094,018	2,787,967	124
youtube (yt)	1,138,499	2,990,443	51
stackoverflow (so)	2,584,164	28,183,518	198
livejournal (lj)	4,846,609	42,851,237	372
orkut	3,072,441	117,185,083	253
ctr	14,081,816	16,933,413	3
usa	23,947,347	28,854,312	3
twitter	41,652,230	1,202,513,046	2488

Table 1. Graph sizes and largest values of k for k-core decomposition.

Cloud instance (3.1 GHz Intel Xeon Cascade Lake CPUs with a total of 30 cores with two-way hyper-threading, and 236 GiB RAM) and an `m1-megamem-96` Google Cloud instance (2.0 GHz Intel Xeon Skylake CPUs with a total of 48 cores with two-way hyper-threading, and 1433.6 GB RAM). We do not use hyper-threading in our experiments as we found it not to improve performance. Our programs are written in C++, use a work-stealing scheduler [16], and are compiled using `g++` (version 7.5.0) with the `-O3` flag. We terminate experiments that take over 2 hours.

We test our algorithms on batches of ***insertions*** and ***deletions***. Unless specified otherwise, all experiments are conducted on batches of 10^6 edges. We run each experiment for 11 trials, and we compute the mean and maximum results for each experiment.

Datasets. We use datasets from the Stanford Network Analysis Project (SNAP), the Network Respository, and the DIMACS Shortest Paths challenge, specifically, the datasets used by Liu et al. [57] in their evaluation: com-DBLP (*dblp*), com-LiveJournal (*lj*), com-Orkut (*orkut*), com-Youtube (*yt*), wiki-talk (*wiki*), sx-stackoverflow (*so*), twitter (*twitter*) [52], human-Jung2015-M87113878 (*brain*), full USA (*usa*), and central USA (*ctr*). Graph characteristics are given in Table 1.

Implementation Details. All of our code is publicly available.[3] We make use of the optimization feature given in the original PLDS code with the `-opt` flag set to 20. This optimization feature speeds up the code but degrades its approximation error. We set the parameters $\delta = 0.2$ and $\lambda = 9$. The theoretical approximation factor using these parameters is 2.8 (i.e., $\varepsilon = 0.8$). Our experiments demonstrate we never exceed the maximum approximation factor obtained by the original PLDS implementation for each dataset. We test our implementations on combinations of different numbers of reader and update threads. Each thread is on a separate core with no other reader or update threads. We test combinations of 1, 2, 4, 8, and 15 reader and update threads. **Latency.** First, we measured the latency of reads using all three implementations on all of the graphs. For all algorithms, each read thread continuously generates reads of vertices chosen uniformly at random for the duration of the batch. Reads for CPLDS are implemented and performed according to our algorithms. NonSync performs reads immediately by looking at the current level of the vertex. Each read thread in SyncReads maintains an array of reads in the order that they are generated during each update batch and performs the reads, in order, at the end of the batch.

For each implementation and graph, we obtain the average, 99-th percentile latency, and 99.99-th percentile latency across all reads and all trials. The results are shown in Fig. 3. We see that against SyncReads, our CPLDS algorithm achieves up to *five orders of magnitude* smaller latency for both insertions and deletions for the average, 99-th percentile and 99.99-th percentile latencies. This is because in SyncReads, reads that arrive must wait until the end of the batch before they can execute. Compared to NonSync, reads are at most 3.21x slower in CPLDS, but are linearizable.

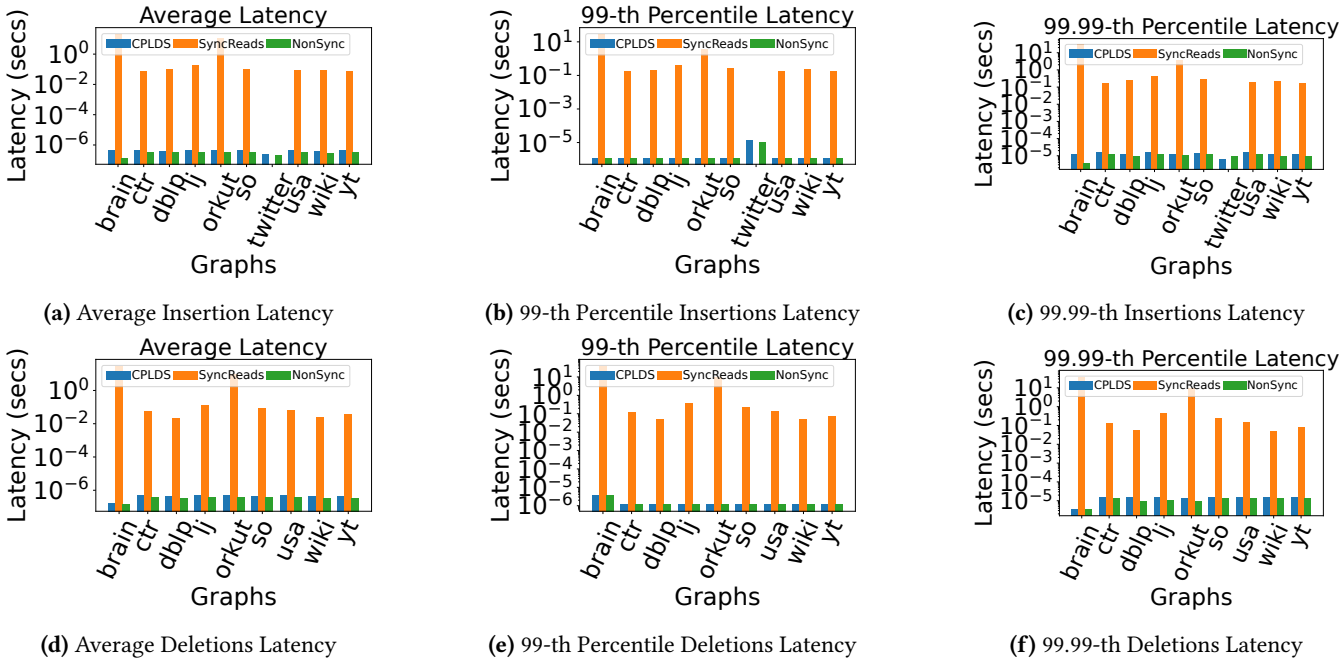
Batch Size vs. Latency. Fig. 4 shows the latency of reads across multiple insertion batch sizes for all three implementations. Specifically, we show the average, 99-th percentile, and 99.99-th percentile latencies for *dblp* and *lj*. For *yt*, the average latency is 1.12–1.38 factor larger for CPLDS than NonSync but is *at least seven orders of magnitude smaller* than SyncReads. For the 99-th percentile latency on *dblp*, CPLDS and NonSync exhibit the same latency and CPLDS exhibits smaller latency than SyncReads by up to seven orders of magnitude. Finally, for the 99.99-th percentile latency on *dblp*, CPLDS exhibits larger latency than NonSync by up to a factor of 3.98, but exhibits up to five orders of magnitude smaller latency than SyncReads.

For *dblp*, the average latency is 1–1.70 factor larger for CPLDS than NonSync but is *at least five orders of magnitude smaller* than SyncReads. For the 99-th percentile on *dblp*, CPLDS and NonSync exhibit the same latency and CPLDS exhibits smaller latency than SyncReads by up to six orders of magnitude. Finally, for the 99.99-th percentile on *dblp*, CPLDS exhibits larger latency than NonSync by up to a factor of 1.88, but exhibits up to five orders of magnitude smaller latency than SyncReads. Deletions follow a similar trend: for *dblp*, the average, 99-th percentile and 99.99-th percentile latencies for CPLDS are up to 1.84, 1.0, and 1.66 factors, respectively, larger than NonSync. Compared to SyncReads, CPLDS exhibits up to six orders of magnitude smaller lantencies on *dblp* and up to seven orders of magnitude smaller latencies on *yt*. For *yt*, the average, 99-th percentile, and 99.99-th percentile latencies for CPLDS are up to 1.44, 1.0, and 2.33 factors, respectively, larger than NonSync.

We found that deletions follow a similar trend.

Update Time. Fig. 5 shows the average and maximum update times throughout all of our trials on all graphs. We see that NonSync requires the least amount of update time, although our algorithm is at most 1.48x slower for both insertions and deletions. The reason that SyncReads requires

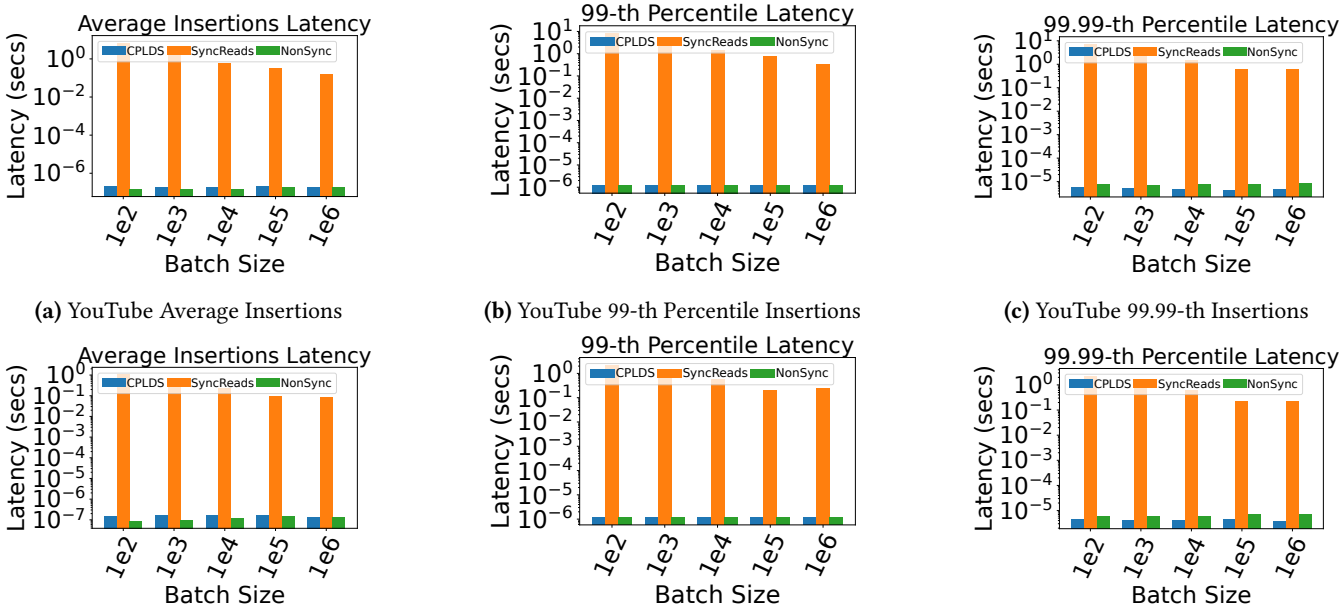
[3] https://github.com/qqliu/batch-dynamic-kcore-decomposition/tree/master/gbbs/benchmarks/EdgeOrientation/ConcurrentPLDS



(a) Average Insertion Latency **(b)** 99-th Percentile Insertions Latency **(c)** 99.99-th Insertions Latency

(d) Average Deletions Latency **(e)** 99-th Percentile Deletions Latency **(f)** 99.99-th Deletions Latency

Figure 3. Comparison of the average, 99-th percentile, and 99.99-th percentile read latencies of the implementations under batches of insertions or deletions. The y-axis is in log-scale. Twitter times out for SyncReads and we do not show their results.



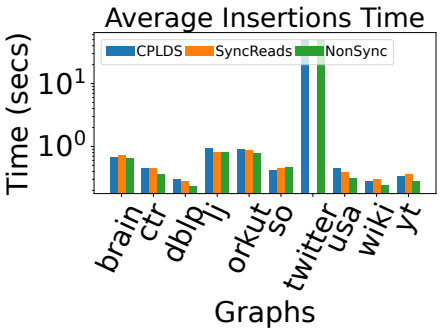
(a) YouTube Average Insertions **(b)** YouTube 99-th Percentile Insertions **(c)** YouTube 99.99-th Insertions

(d) DBLP Average Insertions **(e)** DBLP 99-th Percentile Insertions **(f)** DBLP 99.99-th Insertions

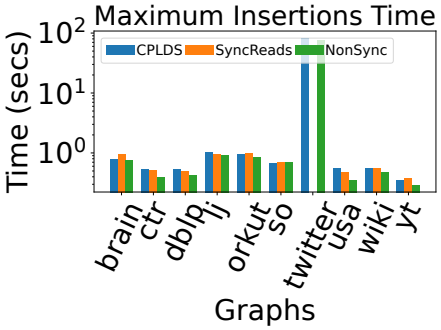
Figure 4. Comparison of the latencies over different insertion batch sizes using 15 update threads and 15 read threads. The y-axis is in log-scale. We tested on *yt* and *dblp*.

more time sometimes (up to 1.85 factor worse) than the other methods is due to the fact that reads occur synchronously and must factor into the update time (since updates are blocked and cannot be performed until all synchronous reads finish). We see that for most graphs, NonSync results in the lowest update time because the updates methods did not change compared to the previous synchronous PLDS implementation of [57].

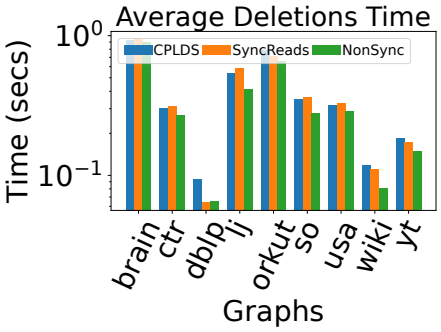
Approximation Factors. Fig. 6 shows the average and maximum approximation factors of our algorithm versus the baselines. We see that the maximum approximation factors



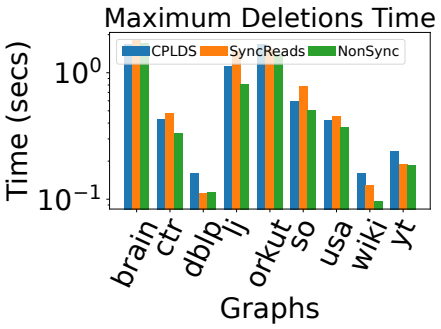
(a) Average Insertions Batch Update Time



(b) Maximum Insertions Batch Update Time

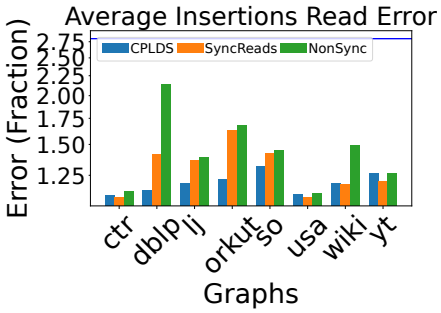


(c) Average Deletions Batch Update Time

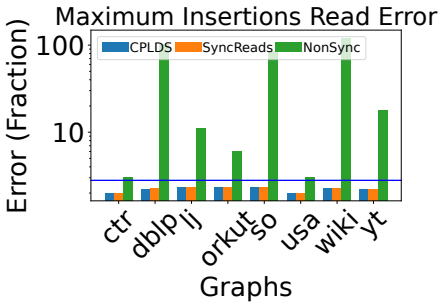


(d) Maximum Deletions Batch Update Time

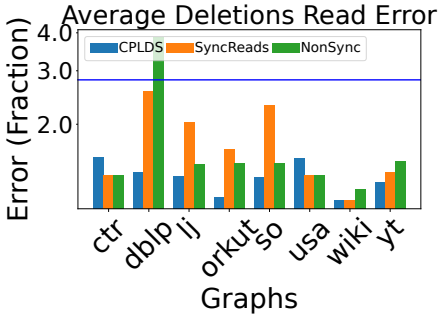
Figure 5. Comparison of the average and maximum batch update time over all batches and trials using 15 update threads and 15 read threads. The y-axis is in log-scale. Twitter times out for SyncReads and we do not show their results.



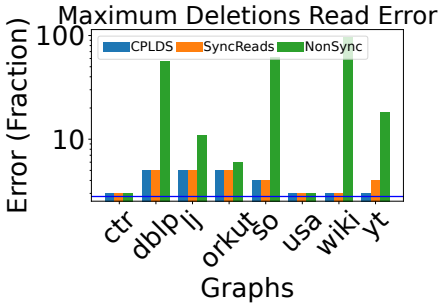
(a) Average Insertions Read Error



(b) Maximum Insertions Read Error



(c) Average Deletions Read Error



(d) Maximum Deletions Read Error

Figure 6. Comparison of the average and maximum errors over all reads and all trials using 15 update threads and 15 read threads. The y-axis is in log-scale. The blue line shows the theoretical maximum error of 2.8. The deletion errors sometimes exceed 2.8 due to the optimizations in our data structure.

for CPLDS are upper bounded by 2.8, the theoretical maximum bound for insertion, and by the maximum approximation factors returned by SyncReads for deletions. The deletion errors for CPLDS and SyncReads exceed 2.8 due to the optimizations in our data structure, as described earlier. For CPLDS, because of our theoretical approximation guarantees, our reads are guaranteed to be linearizable to either the beginning of the batch or the end of the batch. Since it is difficult to know whether the read linearized to the beginning or the end of the batch, we take the minimum of the two errors.

We see that our average error for CPLDS is sometimes slightly larger than the average error for SyncReads, by a factor of at most 1.15. Such a small factor is likely due to the variance in our selections of reads. For NonSync, we return the minimum approximation factor between the beginning and the end of the batch. We see that the maximum errors for NonSync are up to 52.7x worse than CPLDS because the a read can occur while the vertex is in the middle of moving levels. Thus, the vertex can be stuck in a "middle" level whose core number is far from the approximate coreness estimate at the beginning or end of the batch.

Scalability of Read and Write Throughputs. We test the scalability of our read throughputs as we increase the number of reader threads while maintaining 15 writer threads. We also test our write throughput. We record the average throughput across all batches and all trials for the *dblp* and *lj* graphs. For CPLDS and NonSync reads and writes, the average throughput is computed as the total number of reads or writes divided by the total write time over *all* batches. For SyncReads reads and writes, the duration of time in the denominator is the total *read* plus *write* time over all batches, respectively. For the read scalability of SyncReads, we compute the throughput analytically: we divide the total number of reads performed by CPLDS by half of the sum of the update time and the minimum read time of any thread (on average, a read operation will come in the middle of this interval). The minimum read time of any thread is computed by multiplying the minimum observed latency of reads (performed by NonSync) times the total number of reads divided by the number of threads. This analytical computation upper bounds the read throughput of SyncReads. For both graphs, we test on the number of reader threads from $\{1, 2, 4, 8, 15\}$.

In addition to read throughputs, we also test the scalability of our write throughputs as we increase the number of writer threads while maintaining 15 reader threads. For *dblp*, we test on the number of writer threads from $\{1, 2, 4, 8, 15\}$. For *lj*, due to the high running times on smaller number of writer threads, we only test on $\{8, 15\}$.

The results are shown in Fig. 7. We see that NonSync has the greatest read throughput for most graphs due to the fact that it does not requiring synchronization mechanisms for individual reads (i.e., the dependency DAG), while CPLDS has the worst read throughputs. Because we are upper bounding the read throughput of SyncReads, sometimes SyncReads has greater throughput than NonSync (by a small margin). NonSync has slightly higher read throughput by factors of up to 2.21x than CPLDS since reads in NonSync do not have to traverse the dependency DAG. On the other hand, either SyncReads or NonSync have the greatest writer throughput. CPLDS sometimes has the worst write throughput and is sometimes between SyncReads and NonSync, specifically, with write throughput within a factor of 7 of the maximum throughput of either SyncReads and NonSync. Such an ordering of the throughputs is expected as NonSync has the smallest total time (consisting only of write time) while SyncReads also has additional time resulting from reads and CPLDS requires additional time to maintain the DAGs.

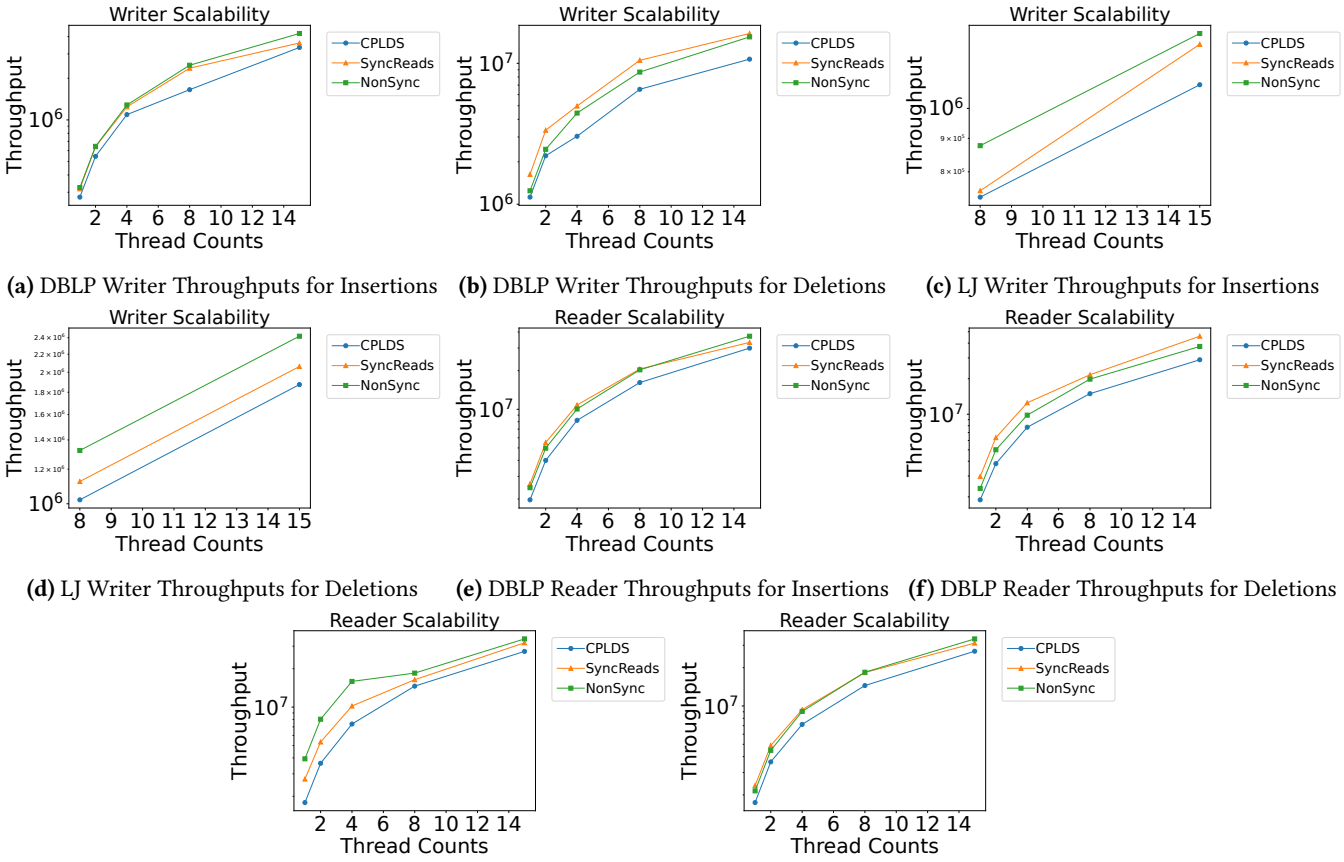
8 Related Work

Parallel batch-dynamic graph algorithms. There has been work on parallel batch-dynamic k-core decomposition, both in the exact [12, 38, 46, 48, 73] and approximate [57] settings. The approximate algorithm of Liu et al. [57] has been shown to significantly outperform the exact algorithms. Similar to our paper, these works maintain a k-core decomposition of a graph, or an approximation thereof, under batches of edge updates. Unlike our work, they do not propose a way to query coreness values concurrently with updates. Parallel batch-dynamic algorithms have been designed for a number of other graph problems [1, 2, 9, 10, 29, 36, 66, 69, 71].

Concurrency on graphs. Fedorov et al. [35] propose a concurrent algorithm for dynamic connectivity, which requires maintaining the connected components of a graph under dynamic edge insertions and deletions. Their algorithm supports single-writer multi-reader concurrency, like our algorithm. If fine-grained locking is applied, their algorithm can handle writers in disjoint components. Nathan et al. [65] propose a non-stop streaming data analysis model, in which updates and reads can proceed concurrently. However, the results of their algorithms are not necessarily linearizable.

Dhulipala et al. [24, 27] design compressed fully-functional trees that support single-writer multi-reader operations on graphs. Unlike our work where the results of reads can reflect the most recent updates, their work only supports concurrent reads on static snapshots of graphs.

Concurrency from parallel batch-dynamic data structures. Aksenov et al. [5] propose *parallel combining*, which implements a concurrent data structure from a parallel batch-dynamic one by synchronizing operations into batches executed by a "combiner." Of particular relevance is their read-optimized version, which performs updates sequentially and reads in parallel. They apply their idea to a dynamic connectivity algorithm. Agrawal et al. [4] propose a similar idea, where a scheduler implicitly batches concurrent accesses to a data structure, executing one batch at a time. Like our paper, both works enable concurrency from batch-dynamic data



(a) DBLP Writer Throughputs for Insertions **(b)** DBLP Writer Throughputs for Deletions **(c)** LJ Writer Throughputs for Insertions

(d) LJ Writer Throughputs for Deletions **(e)** DBLP Reader Throughputs for Insertions **(f)** DBLP Reader Throughputs for Deletions

(g) LJ Reader Throughputs for Insertions **(h)** LJ Reader Throughputs for Deletions

Figure 7. Comparison of the average throughput over all batches and trials using different numbers of update threads and reader threads on the *dblp* and *lj* graphs. The *y*-axis is in log-scale. For the writer throughput experiments, we fix the number of reader threads to 15, and for the reader throughput experiments, we fix the number of writer threads to 15.

structures but, unlike our paper, they do not allow asynchronous reads concurrent with update batches, and therefore cannot guarantee low latency for reads.

Concurrency techniques. Some of our techniques are similar to previous methods in concurrent programming. Operation descriptors, like the ones we use to synchronize reads and updates, are a classic technique for lock-free algorithms [14, 31, 37]. Our sandwiched reads are reminiscent of the clean double collect method used by Afek et al. [3] in their atomic snapshot algorithm. Finally, the epsilon trick has been used before to space out linearization points that would otherwise (incorrectly) occur at the same time [23].

9 Conclusion

We present a novel approximate k-core decomposition algorithm that supports parallel batch-dynamic updates and asynchronous concurrent reads. We ensure linearizability by efficiently tracking causal dependencies between operations using a lightweight dependency DAG design. Our experimental evaluation demonstrates that the high throughput of parallel batch-dynamic updates is preserved, while asynchronous reads attain ultra-low latency and accuracy similar to that of the previous synchronous algorithm. For future work, we are interested in supporting asynchronous updates in our data structure. We are also interested in using our data structure for other closely related graph problems, such as low out-degree orientation, maximal matching, k-clique counting, vertex coloring, and densest subgraph.

Acknowledgments

We thank Rachid Guerraoui, Maurice Herlihy, and Siddhartha Jayanti for helpful discussions. A large portion of this work was completed while Q.C. Liu was a postdoctoral scholar at Northwestern Univeristy and an Apple Research Fellow at the Simons Institute at UC Berkeley. Part of this work was completed while I. Zablotchi was a postdoctoral fellow at MIT CSAIL, where he was supported by SNSF Early Postdoc.Mobility Fellowship P2ELP2_195126. J. Shun was supported DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, Google Research Scholar Award, cloud computing credits from Google-MIT, and FinTech@CSAIL Initiative.

References

[1] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. 2019. Parallel Batch-Dynamic Graph Connectivity. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 381–392.

[2] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, and Sam Westrick. 2020. Parallel Batch-Dynamic Trees via Change Propagation. In *Annual European Symposium on Algorithms*, Vol. 173. 2:1–2:23.

[3] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic Snapshots of Shared Memory. *Journal of the ACM (JACM)* 40, 4 (1993), 873–890.

[4] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. 2014. Provably Good Scheduling for Parallel Programs That Use Data Structures through Implicit Batching. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. 84–95.

[5] Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. 2018. Parallel Combining: Benefits of Explicit Synchronization. In *International Conference on Principles of Distributed Systems (OPODIS)*, Vol. 125. 11:1–11:16.

[6] Dan Alistarh, Alexander Fedorov, and Nikita Koval. 2019. In Search of the Fastest Concurrent Union-Find Algorithm. In *23rd International Conference on Principles of Distributed Systems*, Vol. 153. 15:1–15:16.

[7] Md. Altaf-Ul-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 Bioinform.* 7 (2006), 207.

[8] 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 *Advances in Neural Information Processing Systems*. 41–50.

[9] Daniel Anderson and Guy E. Blelloch. 2023. Deterministic and Work-Efficient Parallel Batch-Dynamic Trees in Low Span. *CoRR* abs/2306.08786 (2023), 20 pages. https://doi.org/10.48550/arXiv.2306.08786

[10] Daniel Anderson, Guy E. Blelloch, Anubhav Baweja, and Umut A. Acar. 2021. Efficient Parallel Self-Adjusting Computation. In *33rd ACM Symposium on Parallelism in Algorithms and Architectures*. 59–70.

[11] Richard Anderson and Ernst W Mayr. 1984. *A P-complete problem and approximations to it.* Technical Report. Stanford University.

[12] Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. 2016. Distributed k-core decomposition and maintenance in large dynamic graphs. In *ACM International Conference on Distributed and Event-based Systems (DEBS)*. 161–168.

[13] Gary D. Bader and Christopher W. V. Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinform.* 4 (2003), 2.

[14] Greg Barnes. 1993. A Method for Implementing Lock-Free Shared-Data Structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 261–270.

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

[16] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. Brief Announcement: ParlayLib – A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symp. on Parallel Alg. (SPAA)*. 507–509.

[17] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 1316–1325.

[18] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference (ATC)*, Andrew Birrell and Emin Gün Sirer (Eds.). 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

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

[20] Audrey Cheng, Xiao Shi, Aaron N. Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. 2022. TAOBench: An End-to-End Benchmark for Social Networking Workloads. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1965–1977. https://doi.org/10.14778/3538598.3538616

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

[22] Martino Ciaperoni, Edoardo Galimberti, Francesco Bonchi, Ciro Cattuto, Francesco Gullo, and Alain Barrat. 2020. Relevance of temporal cores for epidemic spread in temporal networks. *Scientific Reports* 10, 1 (2020), 12529.

[23] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The Inherent Cost of Remembering Consistently. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 259–269.

[24] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: supporting parallel and compressed purely-functional collections. In *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 108–121.

[25] 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.

[26] 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)*. 393–404.

[27] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 918–934.

[28] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms. *Proc. VLDB Endow.* 14, 4 (Dec 2020), 653–667.

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

[30] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2009. Extraction and classification of dense implicit communities in the Web graph. *ACM Trans. Web* 3, 2 (2009), 7:1–7:36.

[31] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing (PODC)*. 131–140.

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

[33] Hossein Esfandiari, Silvio Lattanzi, and Vahab S. Mirrokni. 2018. Parallel and Streaming Algorithms for K-Core Decomposition. In *International Conference on Machine Learning (ICML) (Proceedings of Machine*

Learning Research, Vol. 80). 1396–1405.

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

[35] Alexander Fedorov, Nikita Koval, and Dan Alistarh. 2021. A Scalable Concurrent Algorithm for Dynamic Connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 208–220.

[36] Paolo Ferragina and Fabrizio Luccio. 1994. Batch Dynamic Algorithms for Two Graph Problems. In *International PARLE Conference on Parallel Architectures and Languages Europe*, Vol. 817. 713–724.

[37] Keir Fraser. 2004. *Practical lock-freedom*. Ph. D. Dissertation. University of Cambridge, UK.

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

[39] Edoardo Galimberti, Francesco Bonchi, Francesco Gullo, and Tommaso Lanciano. 2020. Core Decomposition in Multilayer Networks: Theory, Algorithms, and Applications. *ACM Trans. Knowl. Discov. Data* 14, 1 (2020), 11:1–11:40.

[40] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrovic. 2019. Improved Parallel Algorithms for Density-Based Network Clustering. In *International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 97)*. 2201–2210.

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

[42] John Healy, Jeannette C. M. Janssen, Evangelos E. Milios, and William Aiello. 2006. Characterization of Graphs Using Degree Cores. In *International Workshop on Algorithms and Models for the Web-Graph (WAW)*, Vol. 4936. 137–148.

[43] Monika Henzinger, Stefan Neumann, and Andreas Wiese. 2020. Explicit and Implicit Dynamic Coloring of Graphs with Bounded Arboricity. *CoRR* abs/2002.10142 (2020), 18 pages.

[44] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc.

[45] Changwan Hong, Laxman Dhulipala, and Julian Shun. 2020. Exploring the Design Space of Static and Incremental Graph Connectivity Algorithms on GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 55–69.

[46] Qiang-Sheng Hua, Yuliang Shi, Dongxiao Yu, Hai Jin, Jiguo Yu, Zhipeng Cai, Xiuzhen Cheng, and Hanhua Chen. 2020. Faster Parallel Core Maintenance Algorithms in Dynamic Graphs. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2020), 1287–1300.

[47] Siddhartha V. Jayanti and Robert E. Tarjan. 2021. Concurrent disjoint set union. *Distributed Computing* 34, 6 (2021), 413–436.

[48] Hai Jin, Na Wang, Dongxiao Yu, Qiang-Sheng Hua, Xuanhua Shi, and Xia 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.

[49] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-Core Decomposition on Multicore Platforms. In *IEEE International Parallel and Distributed Processing Symposium Workshops, (IPDPS)*. 1482–1491.

[50] Wissam Khaouid, Marina Barsky, S. Venkatesh, and Alex Thomo. 2015. K-Core Decomposition of Large Networks on a Single PC. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.

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

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

[53] Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu C. Aggarwal. 2010. A Survey of Algorithms for Dense Subgraph Discovery. In *Managing and Mining Graph Data*. Advances in Database Systems, Vol. 40. 303–336.

[54] Conggai Li, Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2019. Efficient Progressive Minimum k-core Search. *Proceedings of the VLDB Endowment* 13, 3 (2019), 362–375.

[55] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2453–2465.

[56] Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. 2021. Hierarchical Core Maintenance on Large Dynamic Graphs. *Proceedings of the VLDB Endowment* 14, 5 (2021), 757–770.

[57] Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2022. Parallel Batch-Dynamic Algorithms for k-Core Decomposition and Related Graph Problems. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 191–204.

[58] Quanquan C. Liu, Julian Shun, and Igor Zablotchi. 2024. Parallel k-Core Decomposition with Batched Updates and Asynchronous Reads. arXiv:2401.08015 [cs.DC]

[59] Ying Liu, Ming Tang, Tao Zhou, and Younghae Do. 2015. Core-like groups result in invalidation of identifying super-spreader by k-shell decomposition. *Scientific Reports* 5, 1 (2015), 9602.

[60] Qi Luo, Dongxiao Yu, Feng Li, Zhenhao Dou, Zhipeng Cai, Jiguo Yu, and Xiuzhen Cheng. 2019. Distributed Core Decomposition in Probabilistic Graphs. In *International Conference on Computational Data and Social Networks (CSoNet)*, Vol. 11917. 16–32.

[61] Fragkiskos D. Malliaros, Maria-Evgenia G. Rossi, and Michalis Vazirgiannis. 2016. Locating influential nodes in complex networks. *Scientific Reports* 6, 1 (2016), 19307.

[62] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427.

[63] Sourav Medya, Tianyi Ma, Arlei Silva, and Ambuj K. Singh. 2020. A Game Theoretic Approach For k-Core Minimization. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 1922–1924.

[64] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos E. 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 (KDD)*. 815–824.

[65] Eisha Nathan, E. Jason Riedy, Anita Zakrzewska, and Chunxing Yin. 2017. A New Direction for Streaming Graph Analysis. In *IEEE International Conference on Cluster Computing (CLUSTER)*. 645–646.

[66] Shaunak Pawagi and Owen Kaser. 1993. Optimal parallel algorithms for multiple updates of minimum spanning trees. *Algorithmica* 9, 4 (1993), 357–381.

[67] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for k-core Decomposition. *Proceedings of the VLDB Endowment* 6, 6 (2013), 433–444.

[68] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *Proceedings of the VLDB Endowment* 25, 3 (2016), 425–447.

[69] X. Shen and W. Liang. 1993. A parallel algorithm for multiple edge updates of minimum spanning trees. In *Proceedings Seventh International Parallel Processing Symposium*. 310–317.

[70] 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 (2020), 39:1–39:21.

[71] Tom Tseng, Laxman Dhulipala, and Julian Shun. 2022. Parallel Batch-Dynamic Minimum Spanning Forest and the Efficiency of Dynamic

Agglomerative Graph Clustering. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 233–245.

[72] Kai Wang, Xin Cao, Xuemin Lin, Wenjie Zhang, and Lu Qin. 2018. Efficient Computing of Radius-Bounded k-Cores. In *IEEE International Conference on Data Engineering (ICDE)*. 233–244.

[73] Na Wang, Dongxiao Yu, Hai Jin, Chen Qian, Xia Xie, and Qiang-Sheng Hua. 2017. Parallel Algorithm for Core Maintenance in Dynamic Graphs. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2366–2371.

[74] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2019. I/O Efficient Core Graph Decomposition: Application to Degeneracy Ordering. *IEEE Trans. Knowl. Data Eng.* 31, 1 (2019), 75–90.

[75] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.* 42, 1 (2015), 181–213.

[76] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2017. When Engagement Meets Similarity: Efficient (k, r)-Core Computation on Social Networks. *Proceedings of the VLDB Endowment* 10, 10 (2017), 998–1009.

[77] Haohua Zhang, Hai Zhao, Wei Cai, Jie Liu, and Wanlei Zhou. 2010. Using the k-core decomposition to analyze the static structure of large-scale software systems. *J. Supercomput.* 53, 2 (2010), 352–369.

[78] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *IEEE International Conference on Data Engineering (ICDE)*. 337–348.

A Artifact Appendix

A.1 Setup and Experiment Script

Our experiments use code from the Graph Based Benchmark Suite (GBBS) which can be installed from this Github link: https://github.com/qqliu/batch-dynamic-kcore-decomposition. GBBS is most easily installed and run on Ubuntu 20.04 LTS, but can be installed easily on any Ubuntu machine. We have provided an instance with pre-installed software on which you can run experiments if you provide us with a public key.

First, run `setup.sh` within the main `batch-dynamic-kcore-decomposition/` directory by typing `sh setup.sh` into the command line. The following are the setup instructions that are run by `setup.sh`:

1. If you do not have make, run `sudo apt install make`.
2. If you do not have g++, run `sudo apt-get update`, then `sudo apt-get install g++`.
3. Run `git submodule update --init --recursive` to obtain subpackages from inside the GBBS directory.
4. All scripts for running code is included under the `/batch-dynamic-kcore-decomposition/gbbs/scripts` directory.
5. The relevant scripts are: `cplds_approx_kcore_setup.txt`, `cplds_test_approx_kcore.py`, and `cplds_read_approx_kcore_results.py`.

Experiment Machine Setup Our experiments require machines with 30 cores. Specifically, we tested our experiments on machines with the following specifications. We use a `c2-standard-60` Google Cloud instance (3.1 GHz Intel Xeon Cascade Lake CPUs with a total of 30 cores with two-way hyper-threading, and 236 GiB RAM) and an `m1-megamem-96` Google Cloud instance (2.0 GHz Intel Xeon Skylake CPUs with a total of 48 cores with two-way hyper-threading, and 1433.6 GB RAM). We do not use hyper-threading in our experiments. Our programs are written in C++, use a work-stealing scheduler [16], and are compiled using `g++` (version 7.5.0) with the `-O3` flag. We terminate experiments that take over 2 hours to finish.

Experiment Script We have prepared an experimental script for you to run to reproduce the results for *all* experiments for insertions on three of our tested graphs. We chose these experiments in order for our suite of experiments to complete within a reasonable time limit. All of our experiments in the script can be completed in a total of 15 minutes. The experimental script is included in /batch-dynamic-kcore-decomposition/gbbs/scripts/cplds_experiments and can be run by typing `sh run_experiments.sh` into the terminal. The program outputs into the terminal, the results of all experiments with the corresponding labels.

A.2 Step-by-Step Instructions

All of our experiments can be performed using our general purpose script given in the README file under the `gbbs/benchmarks/EdgeOrientation/ConcurrentPLDS` directory.