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

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

Jessica Shi
MIT CSAIL
USA
jeshi@mit.edu

Shangdi Yu
MIT CSAIL
USA
shangdiy@mit.edu

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

Julian Shun
MIT CSAIL
USA
jshun@mit.edu

Abstract

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

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

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

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

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

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9146-7/22/07.
https://doi.org/10.1145/3490148.3538569

CCS Concepts

• **Theory of computation → Dynamic graph algorithms**; • **Computing methodologies → Shared memory algorithms**.

Keywords

parallel batch-dynamic algorithms, k-core decomposition, low out-degree orientation, maximal matching, k-clique counting, vertex coloring

ACM Reference Format:
Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, Julian Shun. 2022. Parallel Batch-Dynamic Algorithms for k-Core Decomposition and Related Graph Problems. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3490148.3538569

1 Introduction

Discovering the structure of large-scale networks is a fundamental problem for many areas of computing. One of the key challenges is to detect communities in which individuals (or vertices) have close ties with one another, and to understand how well-connected a particular individual is to the community. The well-connectedness of a vertex or a group of vertices is naturally captured by the concept of a k-core or, more generally, the k-core decomposition; hence, this particular problem and its variants have been widely studied in the machine learning [1, 24, 28], database [10, 14, 23, 46, 54], social network analysis and graph analytics [18, 19, 37, 39], computational biology [15, 40, 49, 52], and other communities [27, 39, 51, 59].

Given an undirected graph G, with n vertices and m edges, the k-core of the graph is the maximal subgraph $H \subseteq G$ such that the induced degree of every vertex in H is at least k. The k-core decomposition of the graph is defined as a partition of the graph into layers such that a vertex v is in layer k if it belongs to a k-core but not a $(k+1)$-core; this value is known as the *coreness* of the vertex and the coreness values induce a natural hierarchical clustering. Classic algorithms for k-core decomposition are inherently sequential. A well-known algorithm for finding the decomposition is to iteratively select and remove all vertices v with smallest degree from the graph until the graph is empty [53]. Unfortunately, the length of the sequential dependencies, or the *depth*, of such a process can be $\Omega(n)$ given a graph with n vertices. As k-core decomposition is a P-complete problem [3], it is unlikely to have a parallel algorithm with polylogarithmic depth. To obtain parallel methods with poly$(\log n)$ depth, we relax the condition of obtaining an *exact* decomposition to one of obtaining a close *approximate* decomposition.

Previous works studied approximate k-core decompositions as a way for obtaining faster and more scalable algorithms in larger graphs than in exact settings [13, 14, 24, 28, 63]. Approximate coreness values are useful for applications where existing methods are already approximate, such as diffusion protocols in epidemiological studies [15, 40, 49, 52], community detection and network centrality measures [21, 25, 32, 55, 66, 69], network visualization and modeling [1, 12, 68, 70], protein interactions [2, 5], and clustering [29, 44]. Furthermore, due to the rapidly changing nature of today's large networks, many recent studies have focused on the *dynamic* setting, where edges and vertices can be inserted and deleted and the k-core decomposition is computed in real time.

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

We introduce the necessary definitions in Section 2 before giving a technical overview of our results in Section 3. Section 4 presents our parallel level data structure and k-core decomposition algorithm in more detail. Section 5 presents experimental results. Section 6 gives an overview of our algorithms for several other problems.

2 Preliminaries

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

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

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

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

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

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

Model Definitions. We analyze the theoretical efficiency of our parallel algorithms in the *work-depth model*. The model is defined in terms of two complexity measures ***work*** and ***depth*** [16, 35]. The ***work*** is the total number of operations executed by the algorithm. The ***depth*** is the longest chain of sequential dependencies. We assume that concurrent reads and writes are supported in $O(1)$ work/depth. A ***work-efficient*** parallel algorithm is one with work that asymptotically matches the best-known sequential time complexity for the problem. We say that a bound holds ***with high probability (w.h.p.)*** if it holds with probability at least $1 - 1/n^c$ for any $c \geq 1$.

We use parallel primitives in our algorithms, which take as input a sequence A of length n, including: parallel ***reduce-add***, which returns the sum of the entries in A, and parallel ***filter***, which also takes as input a predicate function f, and returns the sequence B containing each element $a \in A$ where $f(a)$ is true, while preserving the same relative order as the order of elements in A. These primitives take $O(n)$ work and $O(\log n)$ depth [35]. We also use ***parallel hash tables*** that support insertions, deletions, and membership queries; they can perform n insertions or deletions in $O(n)$ work and $O(\log^* n)$ depth w.h.p. and n membership queries in $O(n)$ work and $O(1)$ depth w.h.p. [30].

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

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

All notations used are summarized in Table 1. Our data

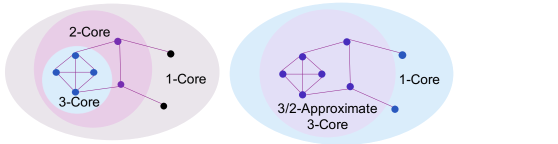

Figure 1: Exact k-core decomposition (left) and $(3/2)$-approximate k-core decomposition (right).

Symbol	Meaning
$G = (V, E)$	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
$\ell, \ell(v)$	a level (starting with level $\ell = 0$), current level of vertex v, resp.
V_ℓ, Z_ℓ	set of vertices in level ℓ, set of vertices in levels $\geq \ell$, resp.
g_i	set of levels in group i (starting with g_0)
$g(v), gn(\ell)$	*group number* of vertex v, index i where level $\ell \in g_i$, resp.
$k(v), \hat{k}(v)$	coreness of v, estimate of the coreness of v, resp.
$\mathrm{up}(v)$,$\mathrm{up}^*(v)$	*up-degree* of v, *up*-degree* of v, resp.
$\varepsilon, \lambda, \delta$	constants where $\varepsilon, \lambda, \delta > 0$

Table 1: Table of notations used in this paper.

structure also maintains a *low out-degree orientation*, which may be parameterized by a graph property known as the *arboricity*.

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

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

We define an $O(\alpha)$ ***out-degree orientation*** to be an acyclic orientation where all out-degrees are $O(\alpha)$. For an oriented graph, we call neighbors of vertex v connected by outgoing edges the ***out-neighbors*** of v and neighbors of v connected by incoming edges the ***in-neighbors*** of v.

3 Technical Overview

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

We first discuss k-core decomposition. A number of previous works [47, 50, 58, 71, 72] provided methods for maintaining the *exact* k-core decomposition under single edge updates in the sequential setting. Unfortunately, none of these works provide algorithms with provable polylogarithmic update time. The main bottleneck for obtaining *provably-efficient* methods is that a single edge update can cause *all* coreness values to change: consider a cycle with one edge removed as a simple example. Removing and adding the edge into this cycle, repeatedly, in succession, causes the coreness of all vertices to change by one with each update. In the parallel setting, a number of previous works [4, 26, 34, 36, 67] investigated batch-dynamic algorithms for exact k-core decomposition. Unfortunately, none of these works have poly$(\log n)$ depth and some even have $\Omega(n)$ depth.

This paper shows that we can surprisingly obtain a parallel batch-dynamic k-core decomposition algorithm with amortized time bounds that are independent of the number of vertices that *changed coreness* for *approximate* coreness. Such provable time bounds can be obtained by cleverly avoiding updating coreness values until enough error has accumulated; once such error has accumulated, we can charge the amount of time required to update the coreness to the number of updates that occurred. Doing so carefully allows a provable $O(\log^2 n)$ amortized work per update that is independent of the number of changed coreness values. A recent paper by Sun et al. [63] provides a *sequential* dynamic approximate k-core decomposition algorithm that takes $O(\log^2 n)$ amortized time per update. Their algorithm is a threshold peeling/elimination procedure that gives a $(2+\varepsilon)$-approximation bound. They also provide another sequential algorithm, which they call *round-indexing*, that performs faster in practice.[2] However, they do not provide formal runtime proofs for this algorithm. Their threshold peeling algorithm is inherently sequential since a vertex that changes thresholds can cause another to change their threshold (and coreness estimate), resulting in a long chain of sequential dependencies; such a situation results in polylogarithmic *amortized* depth, whereas efficient parallel algorithms require polylogarithmic depth w.h.p. in the *worst case*, which we obtain.

[1]d^+_{opt}, is equal to the *degeneracy*, d, of G, and is closely related to α: $d/2 \leq \alpha \leq d$.
[2]Our experiments compare against the round-indexing algorithm since it is faster than their thresholding peeling algorithm in practice.

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

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

To design our k-core decomposition algorithm, we formulate a *parallel level data structure (PLDS)* inspired by the sequential level data structures (LDS) of Bhattacharya et al. [7] and Henzinger et al. [33] to maintain a partition of the vertices satisfying specific degree properties in certain induced subgraphs. In the LDS, vertices are updated one at a time. One of our main technical insights is that we can update many vertices *simultaneously*, leading to high parallelism. Our k-core decomposition algorithm is work-efficient, matches the approximation factor of the best-known sequential dynamic approximate k-core decomposition algorithm of Sun et al. [63], while achieving polylogarithmic depth w.h.p.

Dynamic problems related to k-core decompositions have been recently studied in the theory community such as densest subgraph [7, 60] and low out-degree orientations [6, 11, 31, 33, 38, 41, 42, 62]; some of these works use the LDS. However, none of these previous works proved guarantees regarding the k-core decomposition that can be maintained via a LDS. Notably, we show via a new, intuitive proof that one can use the level of a vertex to estimate its coreness in the LDS of [33]. Unlike the proof in [63] for their dynamic algorithm, our proof does not require densest subgraphs nor any additional information besides the two invariants maintained by the structure.

Our main theoretical and practical technical contributions for k-core decomposition are three-fold: (1) we present a simple modification and a new $(2+\varepsilon)$-approximate coreness proof for the sequential level data structure of [7, 33] (which were not previously used for coreness values) using only the levels of the vertices—no such modification was known prior to this work since [63] requires an additional elimination/peeling/round-indexing procedure; (2) we provide the first parallel work-efficient batch-dynamic level data structure that takes $O(\log^2 n \log\log n)$ depth w.h.p., which we use to obtain a $(2+\epsilon)$-approximate batch-dynamic k-core decomposition algorithm; and (3) we provide multicore implementations of our new algorithm and demonstrate its practicality through extensive experimentation with state-of-the-art parallel and sequential algorithms.

The following theorems give our theoretical bounds.

Theorem 3.1 (Batch-Dynamic k-Core Decomposition). *Given $G = (V, E)$ where $n = |V|$ and batch of updates $\mathcal{B}$, our algorithm maintains*

[3]All bounds are w.h.p., except for the work of static k-core, $O(\alpha \log n)$-coloring, and maximal matching.
[4]$\widetilde{O}$ hides a factor of $O(\log\log n)$.
[5]We denote by α the *current* arboricity of the graph *after processing all updates including the most recent ones.*

$(2+\varepsilon)$-approximations of core values for all vertices (for any constant $\varepsilon > 0$) in $O(|\mathcal{B}|\log^2 n)$ amortized work and $O(\log^2 n \log\log n)$ depth w.h.p., using $O(n\log^2 n + m)$ space.

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

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

A consequence of Theorem 3.2 is the following corollary.

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

Using Theorem 3.2, we design a framework for parallel batch-dynamic algorithms on bounded-arboricity graphs (the framework is described in our full paper [48]), for batch of updates $\mathcal{B}$, which in addition to problem-specific techniques allows us to obtain a set of batch-dynamic algorithms for a variety of other fundamental graph problems including maximal matching, clique counting, and vertex coloring. The coloring algorithms are based heavily on the sequential algorithms of Henzinger et al. [33], but we present them as an application of our framework.

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

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

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

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

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

Our k-core, low out-degree orientation, and vertex coloring algorithms are work-efficient when compared to the best-known sequential, dynamic algorithms for the respective problems [7, 33, 63]. For maximal matching, our algorithm is work-efficient when $\alpha = \Omega(\log^2 n)$ when compared to the best-known sequential algorithm that is robust against adaptive adversaries [31, 56]; the extra work when $\alpha = o(\log^2 n)$ comes from the fact that our bounds are with respect to the *current* arboricity, compared to [31, 56] whose bounds are with respect to the *maximum* arboricity over the sequence of updates.

The best-known batch-dynamic algorithm for k-clique counting, by Dhulipala et al. [20], takes $O(|\mathcal{B}|m\alpha^{k-4})$ expected work and $O(\log^{k-2} n)$ depth w.h.p., using $O(m + |\mathcal{B}|)$ space. Compared with their algorithm, our algorithm uses less work when $m = \omega(\alpha^2\log^2 n)$. In many real-world networks, $\alpha << \sqrt{m}$ (see e.g., Table 3, for maximum k-core values, which upper bound α); thus, our result is more efficient in many cases at an additional multiplicative space cost of $O(\alpha^{k-2})$. We also obtain smaller depth for all $k > 4$. We provide further comparisons with the best-known sequential clique counting algorithm [22] in our full paper [48]. We describe more specific batch-dynamic challenges we face in designing the above algorithms in Section 6. The components of the PLDS used in each of the above results are summarized in Fig. 2.

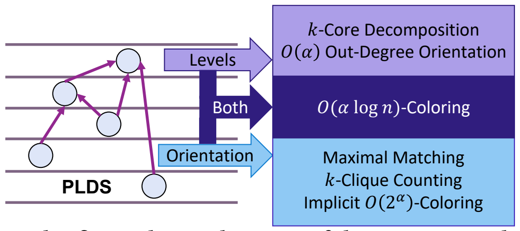
Levels
Both
Orientation
PLDS
k-Core Decomposition
$O(\alpha)$ Out-Degree Orientation
$O(\alpha \log n)$-Coloring
Maximal Matching
k-Clique Counting
Implicit $O(2^\alpha)$-Coloring

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

Finally, using ideas from our batch-dynamic k-core decomposition algorithm, we provide a new parallel static $(2+\varepsilon)$-approximate k-core decomposition algorithm. We compare this algorithm with the best-known parallel static exact algorithm of [18] which uses $O(m+n)$ expected work and $O(\rho\log m)$ depth w.h.p., where ρ is the *number of steps necessary to peel all vertices* (ρ could potentially be $\Omega(n)$). Hence, [18] does not guarantee poly$(\log n)$ depth.

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

Experimental Contributions. In addition to our theoretical contributions, we also provide optimized multicore implementations of our k-core decomposition algorithms. We compare the performance of our algorithms with state-of-the-art algorithms on a variety of real-world graphs using a 30-core machine with two-way hyper-threading. Our parallel static approximate k-core algorithm achieves a 2.8–3.9x speedup over the fastest parallel exact k-core algorithm [18] and achieves a 14.76–36.07x self-relative speedup.

We show that our parallel batch-dynamic k-core algorithm achieves up to 544.22× speedups over the state-of-the-art sequential dynamic approximate k-core algorithm of Sun et al. [63], while achieving comparable accuracy. We also achieve up to 114.52× speedups over the state-of-the-art parallel batch-dynamic exact k-core algorithm of Hua et al. [34], and up to 723.72× speedups against the state-of-the-art sequential exact k-core algorithm of Zhang and Yu [72]. Our batch-dynamic algorithm outperforms the best multicore static k-core algorithms by up to 121.76× on batch sizes that are less than 1/3 of the number of edges in the entire graph. We demonstrate that existing exact dynamic implementations are not efficient or scalable enough to handle

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

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

4 Batch-Dynamic k-Core Decomposition

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

4.1 Algorithm Overview

We present a *parallel level data structure (PLDS)* that maintains a $(2+\varepsilon)$-approximate k-core decomposition that is inspired by the class of sequential level data structures (LDS) of [7, 33]. Our algorithm achieves $O(\log^2 n)$ amortized work per update and $O(\log^2 n \log\log n)$ depth w.h.p. In our full paper [48], we also present a deterministic version of our algorithm that achieves the same work bound with $O(\log^3 n)$ depth. Our data structure can also handle batches of vertex insertions/deletions (described in our full paper [48]). Our data structure requires $O(\log^2 n)$ amortized work, which matches the $O(\log^2 n)$ amortized update time of [7, 33]. As in [33], our data structure can handle *changing arboricity* that is not known a priori. Such adaptivity is necessary to successfully maintain accurate approximations of coreness values.

The LDS and our PLDS consists of a partition of the vertices into $K = O(\log^2 n)$ ***levels***.[8] We provide a very high level overview of PLDS in this section. The levels are partitioned into equal-sized ***groups*** of consecutive levels. Updates are partitioned into insertions and deletions. Vertices move up and down levels depending on the type of edge update incident to the vertex. Rules governing the induced degrees of vertices to neighbors in different levels determine whether a vertex moves. Using information about the level of a vertex, we obtain a $(2+\varepsilon)$-approximation on the coreness of the vertex.

After every edge update, vertices update their levels depending on whether they satisfy two invariants. One invariant upper bounds the induced degree of each vertex v in the subgraph consisting of all vertices in the same or higher level. Vertices whose degree exceeds this bound move up one or more levels. We process the levels from smallest to largest level and move all vertices from the same level in parallel. The second invariant lower bounds the induced degree of each vertex v in the subgraph consisting of all vertices in the level below v, the level of v and all levels higher than the level of v. Vertices that violate this invariant calculate a ***desire-level*** or the closest level they can move to that satisfies this invariant. Then, vertices with the same desire-level are moved in parallel to that level. Finally, the coreness estimates of the vertices are computed based on the current level of each vertex. We obtain the low out-degree orientation by orienting edges from lower to higher levels (breaking ties by vertex index). Fig. 4 shows the invariants maintained by our algorithm; Figs. 5 and 6 show how our algorithm processes insertion and deletion updates. Together, they demonstrate an example run of our algorithm.

[8]When $m = o(n)$, we can also show that $O(\log^2 m)$ levels suffice.

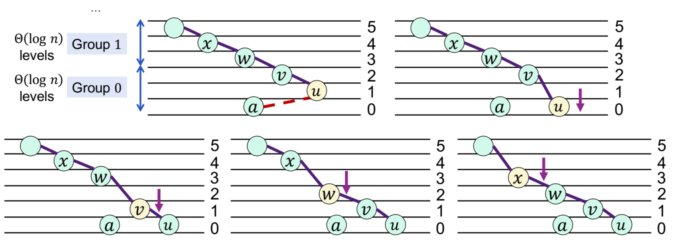

Figure 3: Example of a cascade of vertex movements caused by an edge deletion on u (shown by the dashed red line).

4.2 Sequential Level Data Structure (LDS)

The sequential level data structures (LDS) of [7, 33] maintains a low out-degree orientation under dynamic updates. Within their LDS, a vertex moves up or down levels one by one, where a vertex v (incident to an edge update) first checks whether an invariant is violated, and then may move up or down one level. Then, the vertex checks the invariants and repeats. Such movements may cause other vertices to move up or down levels. The LDS combined with our Section 4.5 directly gives an $O(\log^2 n)$ update time sequential, dynamic algorithm that outputs $(2+\varepsilon)$-approximate coreness values.

Unfortunately, such a procedure can be slow in practice. Specifically, a vertex that moves one level could cause a cascade of vertices to move one level. Then, if the vertex moves again, the same cascade of movements may occur. An example is shown in Fig. 3. Furthermore, any trivial parallelization of the LDS to support a batch of updates will run into race conditions and other issues, requiring the use of locks which blows up the runtime in practice.

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

4.3 Detailed PLDS Algorithm

As mentioned previously, the vertices of the input graph $G = (V, E)$ in our PLDS are partitioned across K ***levels***. For each level $\ell = 0, \dots, K-1$, let V_ℓ be the set of vertices that are currently assigned to level ℓ. Let Z_ℓ be the set of vertices in levels $\geq \ell$. Provided a constant $\delta > 0$, the levels are partitioned into ***groups*** $g_0, \dots, g_{\lceil \log_{(1+\delta)} n \rceil}$, where each group contains $4\lceil \log_{(1+\delta)} n \rceil$ consecutive levels. Each $\ell \in \left[i\lceil \log_{(1+\delta)} n \rceil, \dots, (i+1)\lceil \log_{(1+\delta)} n \rceil - 1 \right]$ is a level in group i. Our data structure consists of $K = O(\log^2 n)$ total levels. The PLDS satisfies the following invariants as introduced in [7, 33], which also govern how the data structure is maintained. The invariants assume a given constant $\delta > 0$ and a constant $\lambda > 0$.

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

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

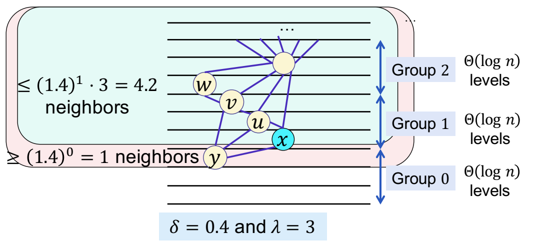

Figure 4: Example of invariants maintained by the PLDS for $\delta = 0.4$ and $\lambda = 3$. There are $\Theta(\log n)$ groups, each with $\Theta(\log n)$. Each vertex is in exactly one level of the structure and moves up and down by some movement rules. For example, vertex x (blue) is on level 3 and in group 1.

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

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

Vertices with no neighbors are placed in level 0. An example partitioning of vertices and maintained invariants is shown in Fig. 4. Let $\ell(v)$ be the level that v is currently on. We define the ***group number***, $g(v)$, of a vertex v to be the index i of the group g_i where $\ell(v) \in g_i$. Similarly, we define $gn(\ell) = i$ to be the group number for level ℓ where $\ell \in g_i$. We define the ***up-degree***, up(v), of a vertex v to be the number of its neighbors in $Z_{\ell(v)}$ (***up-neighbors***), and ***up*-degree***, up$^*(v)$, to be the number of its neighbors in $Z_{\ell(v)-1}$ (up*-***neighbors***). These two notions of induced degree correspond to the requirements of the two invariants of our data structure. We define neighbors w of v at levels $\ell(w) < \ell(v)$ to be the ***down-neighbors*** of v. Lastly, the ***desire-level*** 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 4.1 (Desire-level)**.** *The* desire-level, dl(v), *of vertex v is the level ℓ' that minimizes $|\ell(v) - \ell'|$, and where* up$^*(v) \geq (1+\delta)^{i'}$ *and* up$(v) \leq (2 + 3/\lambda)(1+\delta)^i$ *where $\ell' - 1 \in g_{i'}$, $\ell' \in g_i$, and $i' \leq i$. In other words, the desire-level of v is the closest level ℓ' to the current level of v, $\ell(v)$, where both Invariant 1 and Invariant 2 are satisfied.*

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

Data Structures. Each vertex v keeps track of its set of neighbors in two structures. U keeps track of the neighbors at v's level and above. We denote this set of v's neighbors by $U[v]$. L_v keeps track of neighbors of v for every level below $\ell(v)$—in particular, $L_v[j]$ contains the neighbors of v at level $j < \ell(v)$. We describe specific data structure implementation details in our full paper [48].

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

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

Fig. 5 shows an example of our entire insertion procedure described in Algorithm 2 for $\delta = 0.4$ and $\lambda = 3$. The red lines in the example represent the batch of edge insertions. Thus, in (a), the newly inserted edges are the edges (u, v), (u, x), and (x, w). We

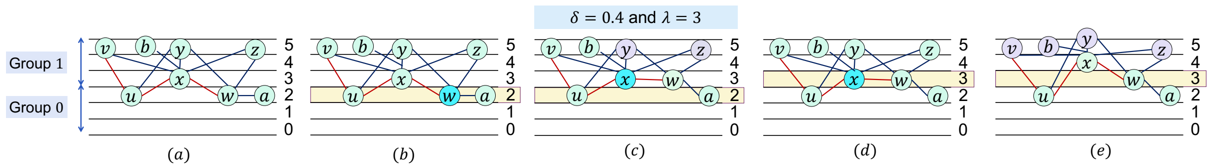

Figure 5: Example of RebalanceInsertions described in the text for $\delta = 0.4$ and $\lambda = 3$. The red lines represent the batch of edge insertions.

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

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

iterate from the bottommost level (level 0) to the topmost level (level $K-1$).

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

RebalanceDeletions($\mathcal{B}_{del}$). Unlike in LDS, deletions in PLDS are handled by moving each vertex at most once, directly to its final level (the vertex *does not move* again during this procedure). We show in the analysis that this guarantee is *crucial to obtaining low depth.* The pseudocode is shown in Algorithm 3. For each vertex v incident to an edge deletion, we check whether it violates Invariant 2

Algorithm 4 CalculateDesireLevel(v)

```
Input: A vertex $v$ that needs to move to a level $j < \ell(v)$.
Output: The desire-level $\text{dl}(v)$ of vertex $v$.
1: $d \leftarrow \text{up}^*(v), p \leftarrow 1, i \leftarrow 2$
2: while $d < (1+\delta)^{gn(\ell(v)-p)}$ and $\ell(v) - p > 0$ do
3:     $d \leftarrow d + \sum_{j=p}^{i-1} |L_v[\ell(v) - j - 1]|$
4:     if $d \ge (1+\delta)^{gn(\ell(v)-i)}$ then
5:         Binary search within levels $[\ell(v) - i + 1, \dots, \ell(v) - p]$ to find the closest level to $\ell(v)$ that satisfies Invariants 1 and 2; return this level.
6:     $p \leftarrow i, i \leftarrow \min(2 \cdot i, \ell(v))$.
7: return 0.
```

(Line 4). On Line 4, $gn(\ell(v) - 1)$ gives the group number i where $\ell(v) - 1 \in g_i$. If v violates Invariant 2, we calculate its desire-level, $\text{dl}(v)$, using CalculateDesireLevel (Line 5), described next. We iterate through the levels from $l = 0$ to $l = K - 1$ (Line 6). Then, in parallel for each vertex v whose desire-level is l, we move v to level l (Lines 7–8). We update the data structures of each v that moved and $w \in N(v)$ where $\ell(w) \ge l$ (Lines 9–21). Specifically, we need to update $U[v]$, $U[w]$, L_v, and L_w if v was originally an up-neighbor of w and becomes a down-neighbor or vice versa. Finally, we update the desire-level of neighbors of v that no longer satisfy Invariant 2 (Lines 22–23). We process all vertices that move and their neighbors in parallel.

Fig. 6 shows an example of Algorithm 3 for $\delta = 1$ and $\lambda = 3$. In (a), the newly deleted edges are (x, z) and (y, w). For each vertex adjacent to an edge deletion, we calculate its desire-level, or the closest level to its current level that satisfies Invariant 2. In (b), only x and z violate Invariant 2. The lower bound on the number of neighbors that must be at or above level 3 for x and level 4 for z is $(1 + \delta)^1 = 2$ since $\delta = 1$ and levels 3 and 4 are in group 1. (Recall that the lower bound is calculated with respect to the level *below* x and z.) We calculate that the desire-levels of x and z are both 3. The desire-levels of y and w are their current levels because they do not violate the invariant. Then, we iterate from the bottommost level (starting with level 0) to the topmost level (level $K - 1$). Level 3 is the first level where vertices want to move. Then, we move x and z to level 3 (shown in (c)). We only need to update the data structures of neighbors at or above x and z so we only update the structures of x, y, and z. Invariant 2 is no longer violated for x and z. In fact, our algorithm guarantees that each vertex *moves at most once.* We check whether any of x or z's up-neighbors violate Invariant 2. Indeed, y now violates the invariant. In (d), we recompute the desire-level of y and its desire-level is now 4. Then, we move y to level 4 in (e).

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

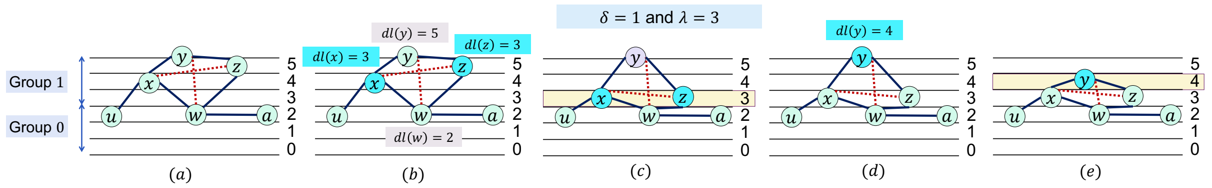

Figure 6: Example of RebalanceDeletions described in the text for $\delta = 1$ and $\lambda = 3$. The red dotted lines represent the batch of edge deletions.

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 iteratively double the number of levels from which we count neighbors until we find a level where Invariant 2 is satisfied (Line 6). On each iteration, we sum the number of neighbors (Line 3) in the range of levels using a parallel reduce. We continue until we find a level where Invariant 2 is satisfied. Let this level be ℓ' and the previous cutoff be ℓ_{prev}. Finally, we perform a binary search within the range $[\ell', \dots, \ell_{prev}]$ to find the *closest* level to $\ell(v)$ that satisfies Invariant 2 (Line 5).

4.4 Efficiency Analysis

We now analyze the work and depth of our PLDS. First, it is easy to show that there exists a level where both invariants are satisfied. This allows our PLDS to assign each vertex to a single level. Then, we make the following two observations that a batch of insertions never violates Invariant 2 and a batch of deletions never violates Invariant 1. This is true because deletions can never increase the up-degree of any vertex and insertions can never decrease the up*-degree of any vertex. All proofs are given in our full paper [48].

Observation 4.2 (Batch Insertions). *Given a batch of insertions, $\mathcal{B}_{ins}$, Invariant 2 is never violated while $\mathcal{B}_{ins}$ is applied.*

Observation 4.3 (Batch Deletions). *Given a batch of deletions, $\mathcal{B}_{del}$, Invariant 1 is never violated while $\mathcal{B}_{del}$ is applied.*

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

Lemma 4.4. *After processing level i in Algorithm 2, no vertex v in levels $\ell(v) \le i$ will violate Invariant 1. Furthermore, no vertex w on levels $\ell(w) > i$ will have* $\mathsf{dl}(w) \le i$.

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

Lemma 4.5. *After processing all vertices that move to level i in Algorithm 3, no vertex v needs to be moved to any level $j \le i$ in a future iteration of Line 6; i.e., no vertex v has* $\mathsf{dl}(v) \le i$ *after processing i.*

We describe the depth of our parallel data structures next. We maintain the list of neighbors using separate parallel hash tables for each vertex v. One hash table contains v's neighbors at the same or higher levels. Vertex v's neighbors in levels below $\ell(v)$ are placed in a separate hash table for each level. Parallel lookups into the hash tables require $O(1)$ depth w.h.p., and inserting and deleting elements within the tables require $O(\log^* n)$ depth w.h.p.

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

Using the above, we successfully prove that the depth of Algorithm 1 is $O(\log^2 n \log \log n)$ w.h.p. The extra space in addition to storing the graph is $O(n \log^2 n)$ because we must have $O(\log^2 n)$ size dynamic arrays for each vertex to track their neighbors at lower levels (i.e., the neighbors in L_v). We provide a set of linear-space data structures in the full paper [48] at the cost of increased depth. Our work bound uses potential functions similar to those in Section 4 of [7]. Our parallel algorithm serializes to a set of sequential steps that can be analyzed using these potential functions. We present the proof of the work bound in our full paper [48]. Together, we obtain the work, depth, and space bounds in Theorem 3.1.

4.5 Estimating the Coreness and Orientation

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

Definition 4.6 (Coreness Estimate). *The* coreness estimate $\hat{k}(v)$ *of vertex v is $(1+\delta)^{\max(\lfloor(\ell(v)+1)/4\lceil\log_{1+\delta} n\rceil\rfloor - 1, 0)}$, where each group has $4\lceil\log_{(1+\delta)} n\rceil$ levels.*

To see an example, consider vertex y in Fig. 6 (e). We estimate $\hat{k}(y) = 1$ since the highest level that is the last level of a group and is equal to or below level $\ell(y) = 4$ is level 2. Level 2 is part of group 0, and so our coreness estimate for y is $(1+\delta)^0 = 1$. This is a 2-approximation of its actual coreness of 2. Using Definition 4.6, we prove that our PLDS maintains a $(2 + 3/\lambda)(1+\delta)$-approximation of the coreness value of each vertex, for any constants $\lambda > 0$ and $\delta > 0$. Therefore, we obtain the following lemma giving the desired $(2+\varepsilon)$-approximation. Our experimental analysis shows that our theoretical bounds limit the maximum error of our experiments, although our average errors are much smaller. To get a maximum error bound of $(2+\varepsilon)$ for any $\varepsilon > 0$, we can set $\delta = \varepsilon/3$ and $\lambda = \frac{9}{\varepsilon} + 3$.

By Lemma 4.8, it suffices to return $\hat{k}(v)$ as the estimate of the coreness of v; this proves the approximation factor in Theorem 3.1.

Lemma 4.7. *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 $\hat{k}(v) \ge (1+\delta)^{g'}$. Otherwise, if $k(v) < \frac{(1+\delta)^{g'}}{(2+3/\lambda)(1+\delta)}$, then $\hat{k}(v) < (1+\delta)^{g'}$.*

The proof of Lemma 4.7 is in the full paper [48] and is an inductive proof using the PLDS invariants. We also show in the full paper that it implies Lemma 4.8.

Lemma 4.8. *The coreness estimate $\hat{k}(v)$ of a vertex v satisfies $\frac{k(v)}{(2+\varepsilon)} \leq \hat{k}(v) \leq (2+\varepsilon)k(v)$ for any constant $\varepsilon > 0$.*

For arbitrary batch sizes, getting better than a 2-approximation for coreness values is P-complete [3], and so there is unlikely to exist a polylogarithmic-depth algorithm with such guarantees.

$O(\alpha)$ **Out-Degree Orientation.** We orient all edges from vertices in lower levels to higher levels, breaking ties for vertices on the same level by using their indices. Such an orientation can be maintained dynamically in the same work and depth as our PLDS via a parallel hash table keyed by the edges and where the values give the orientation. The proof of Theorem 3.2 is given in our full paper [48].

5 Experimental Evaluation

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

Evaluated Algorithms. We evaluate two versions of our algorithm: ***PLDS***: an exact implementation of our theoretical algorithm and ***PLDSOpt***: a version with $\lceil \log_{1+\delta} n/50 \rceil$ levels per group. ***PLDS*** maintains the approximation guarantees given by Lemma 4.8, while ***PLDSOpt*** achieves better performance while maintaining slightly worse approximation bounds.

We compare our algorithms with the following *dynamic* implementations: ***Sun***: the sequential, approximate algorithm of Sun et al. [63], specifically their faster, round-indexing algorithm, which is publicly available [64]; ***Hua***: the parallel, exact algorithm of Hua et al. [34], kindly provided by the authors; ***Zhang***: the sequential, exact algorithm of Zhang and Yu [72], kindly provided by the authors; and ***LDS***: our implementation of the sequential, approximate algorithm of Henzinger et al. [33], but using our coreness approximation procedure in Section 4.5. All are state-of-the-art algorithms, outperforming previous algorithms in their respective categories.

We also implemented ***ApproxKCore***, our new static parallel approximate k-core decomposition algorithm (Theorem 3.8). We compared it with ***ExactKCore***, the state-of-the-art parallel, static, exact k-core algorithm of Dhulipala et al. [18].

Setup. We use `c2-standard-60` Google Cloud instances (3.1 GHz Intel Xeon Cascade Lake CPUs with a total of 30 cores with two-way hyper-threading, and 236 GiB RAM) and `m1-megamem-96` Google Cloud instances (2.0 GHz Intel Xeon Skylake CPUs with a total of 48 cores with two-way hyper-threading, and 1433.6 GB RAM). We use hyper-threading in our parallel experiments by default. Our programs are written in C++, use a work-stealing scheduler [8], and are compiled using g++ (version 7.5.0) with the `-O3` flag. We terminate experiments that take over 3 hours. PLDS and PLDSOpt finished within 3 hours for all experiments.

Datasets. We test our algorithms on 11 real-world undirected graphs from SNAP [45], the DIMACS Shortest Paths challenge road networks [17], and the Network Repository [57], namely ***dblp***, ***brain***, ***wiki***, ***orkut***, ***friendster***, ***stackoverflow***, ***usa***, ***ctr***, ***youtube***, and ***livejournal***. We also used ***twitter***, a symmetrized version of the Twitter network [43]. We remove duplicate edges, zero-degree vertices, and self-loops. Table 3 reflects the graph sizes *after* this removal, and gives the largest k-core values. Both *stackoverflow* and *wiki* are temporal networks; for these, we maintain the edge insertions and deletions in the temporal order from SNAP. *usa* and *ctr* are two high-diameter road networks and *brain* is a highly dense human brain network from NeuroData (https://neurodata.io/). All experiments are run on the `c2-standard-60` instances, except for *twitter* and *friendster*, which are run on the `m1-megamem-96` instances as they require more memory.

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

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

Ins/Del/Mix Experiments. Our experiments are run for *three different types of batched updates*, referred to by: (1) **Ins**: starting with an empty graph, *all* edges are inserted in multiple size $|\mathcal{B}|$ batches of insertion updates, (2) **Del**: starting with the original graph, *all* edges are deleted in multiple size $|\mathcal{B}|$ batches of deletion updates, and (3) **Mix**: starting with the initial graph minus a random set I of $|\mathcal{B}|/2$ edges, a set D of $|\mathcal{B}|/2$ random edges is chosen among the edges in the graph; then, a single size $|\mathcal{B}|$ mixed batch of updates with insertions I and deletions D is applied. For the temporal graphs, *stackoverflow* and *wiki*, the order of updates in the batches follows the order in SNAP [45]. For the rest, updates are generated by taking two random permutations of the edge list, one for **Ins** and one for **Del**. Batches are generated by taking regular intervals of the permuted lists. For **Mix**, I and D are chosen uniformly at random.

5.1 Accuracy vs. Running Time

We start by evaluating the empirical error ratio of the per-vertex core estimates given by our implementations (PLDSOpt, PLDS, LDS) and Sun on *dblp* and *livejournal*, using batches of size 10^5 and 10^6, respectively. Fig. 7 shows the average batch time (in seconds) against the average and maximum *per-vertex* core estimate error ratio. This error ratio is computed as $\max\left(\frac{\hat{k}(v)}{k(v)}, \frac{k(v)}{\hat{k}(v)}\right)$ for each vertex v (where $\hat{k}(v)$ is the core estimate and $k(v)$ is the exact core value). The average is the error ratio averaged across all vertices and the maximum is the maximum error. If the exact core number is 0, we ignore the vertex in our error ratio since our algorithm guarantees an estimate of 0; for vertices of non-zero degree, the lowest estimated core number is 1 for all implementations.

The parameters we use for PLDSOpt, PLDS, and LDS are all combinations of $\delta = \{0.2, 0.4, 0.8, 1.6, 3.2, 6.4\}$ and $\lambda = \{3, 6, 12, 24, 48, 96\}$. We call these *theoretically-efficient parameters*, since they maintain the work-efficiency of our algorithms. For Sun, we use all combinations of their parameters $\varepsilon_{sun} = \lambda_{sun} = \{0.2, 0.4, 0.8, 1.6, 3.2\}$, and $\alpha_{sun} = \{2(1 + 3\varepsilon_{sun})\}$. We also tested $\alpha_{sun} = \{1.1, 2, 3.2\}$, as done in Sun et al.'s work [63]. When $\alpha = 1.1$, the theoretical efficiency bounds by Sun et al. [63] no longer hold, but they yield better estimates empirically. We compare this heuristic setting to a similar one in our algorithms, where we replace $(2 + 3/\lambda)$ with 1.1 in our code (where our efficiency bounds no longer hold) for $\delta = \{0.4, 0.8, 1.6, 3.2\}$. We refer to these as the *heuristic parameters*.

Fig. 7 shows that, using theoretically-efficient parameters, our PLDSOpt, PLDS, and LDS implementations are faster than Sun, Zhang, and Hua, for parameters that give similar average and maximum per-vertex core estimate error ratios. Furthermore, besides PLDS, PLDSOpt *outperforms all other algorithms*, regardless of approximation factor and error. This set of experiments demonstrates the flexibility of our algorithm; one can achieve smaller error at the cost of slightly increased runtime. However, as the experiments demonstrate, PLDSOpt still outperforms all other algorithms even when the parameters are tuned to give small error; this performance gain is maintained for **Ins**, **Del**, and **Mix**. Greater speedups are achieved on *livejournal* compared to *dblp*. Such a result is expected since larger batches allow for greater parallelism.

Concretely, compared with Zhang, PLDSOpt achieves 7.19–147.59×, 19.70–58.41×, and 9.75–142.79× speedups on **Ins**, **Del**, and **Mix** batches, respectively. Compared with Hua, PLDSOpt achieves 2.49–33.95×, 6.81–24.51×, and 2.94–21.77× speedups. Against PLDS, PLDSOpt obtains 2.98–47.8×, 1.03–25.58×, and 1.5–76.94× speedups for **Ins**, **Del**, and **Mix**, respectively, on parameters that give similar approximations. Compared with Sun, on parameters that give similar theoretical guarantees and smaller empirical average error, PLDSOpt achieves 21.34–544.22×, 25.49–128.65×, and 19.04–248.36× speedups for **Ins**, **Del**, and **Mix**, respectively. Neither Zhang nor Hua guarantee polylogarithmic work. The peeling-based algorithm of Sun can have large depth and they do not provide a concrete bound on their amortized work for their faster, round-indexing implementation. Thus, the speedups we obtain over the benchmarks are due to the greater theoretical efficiency and because our algorithms are parallel.

Finally, PLDSOpt achieves average error in the ranges 1.26–2.13, 1.47–4.20, and 1.28–2.33 for **Ins**, **Del**, and **Mix**, respectively. PLDS gives comparable average errors in the ranges 1.27–4.22, 1.33–3.39, and 1.63–5.73, for **Ins**, **Del**, and **Mix**, respectively, while running slower than PLDSOpt for all parameters, despite the guarantee that the maximum error of PLDS is bounded by $(1 + \delta)(2 + 3/\lambda)$ (Lemma 4.8). Thus, our optimized version allows us to obtain good error bounds empirically while drastically improving performance.

5.2 Batch Size vs. Running Time

Fig. 8 shows the average per-batch running times for **Ins**, **Del**, and **Mix** on varying batch sizes for PLDSOpt, PLDS, Hua, LDS, and Zhang on *dblp* and *livejournal*. We do not run this experiment on Sun since their implementation does not have batching. Our experiments show that PLDSOpt is faster for all batch sizes except for the smallest **Del** and **Mix** batches.

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

5.3 Thread Count vs. Running Time

Fig. 9 shows the scalability of PLDSOpt, PLDS, and Hua with respect to their single-thread running times on *dblp* and *livejournal* using a batch size of 10^6. LDS, Sun, and Zhang are represented as horizontal lines since they are sequential. For **Ins**, **Del**, and **Mix** batches, PLDSOpt and PLDS achieve up to 30.28×, 32.02×, and 33.02×, and 26.46×, 25.33×, and 21.15×, self-relative speedup, respectively. Hua achieves up to a 3.6× self-relative speedup. We see that our PLDS algorithms achieve greater self-relative speedups than Hua. Also, with just 4 threads (available on a standard laptop), PLDSOpt already outperforms all other algorithms. Hua's algorithm performs DFS/BFS, which could lead to linear depth, potentially explaining the bottleneck to their scalability with more cores.

Gabert et al. [26] present a parallel batch-dynamic k-core decomposition algorithm but their code is proprietary. However, their algorithm appears slower and less scalable based on their paper's stated results. For example, their algorithm on 10^5 edges using 32 threads for the *livejournal* graph requires 4 seconds, while our algorithm on a batch of 10^6 edges using 30 threads (more edges and fewer threads) requires a *maximum* of 0.35 seconds. Also, they appear to exhibit a maximum of 8× self-relative speedup on *livejournal* while we exhibit 21.2× self-relative speedup on *livejournal*.

5.4 Results on Large Graphs

Fig. 10 shows the runtimes of PLDSOpt, PLDS, Hua, Sun, and Zhang compared with the static algorithms ExactKCore and ApproxKCore on additional graphs, using **Ins**, **Del**, and **Mix** batches, all of size 10^6. ExactKCore and ApproxKCore are run from scratch over the entire graph after every batch since they do not handle batch updates. PLDSOpt and PLDS finished for all graphs and experiments while all other algorithms timed out on **Ins** and **Del** batches for *twitter* and *friendster*. Zhang was able to finish on **Mix** because their indexing algorithm (used to create their data structures provided the initial graph without the mixed batch) was able to finish; since only one mixed batch is used to update the graph, the sum of the time needed for indexing plus the update time of one batch fell under the timeout. The same is true for ExactKCore and ApproxKCore. However, these algorithms were not able to finish for **Ins** and **Del** because the sum of the update times across all batches is too high.

PLDSOpt is faster than all other dynamic algorithms on all types of batches, except for PLDS on *ctr* and *usa*. We report concrete speedups for experiments which finished within the timeout. For **Ins**, it gets 10.01–229.71× speedups over Zhang, 6.20–58.66× speedups over Hua, 26.02–119.77× speedups over Sun, and 1.45–23.89× speedups over PLDS. For **Del**, it gets 30–176.48× speedups

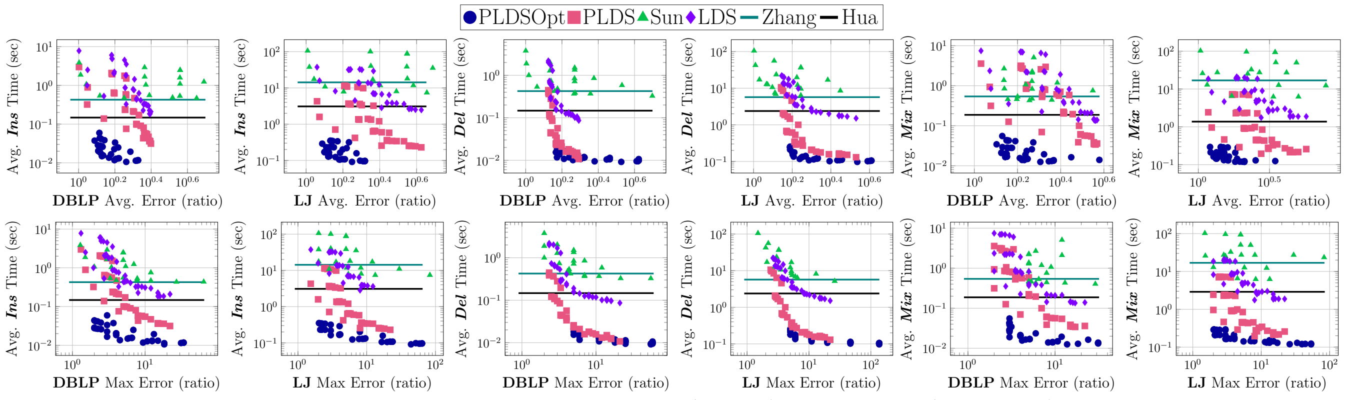

Figure 7: Comparison of the average per-batch time versus the average (top row) and maximum (bottom row) per-vertex core estimate error ratio of PLDSOpt, PLDS, Sun, and LDS, using varying parameters, on the *dblp* and *livejournal* graphs, with batch sizes 10^5 and 10^6, respectively. Experiments were run for Ins, Del, and Mix. The data uses theoretically-efficient parameters as well as the heuristic parameters where $(2 + 3/\lambda) = \alpha_{sun} = 1.1$. Runtimes for Hua and Zhang are shown as horizontal lines.

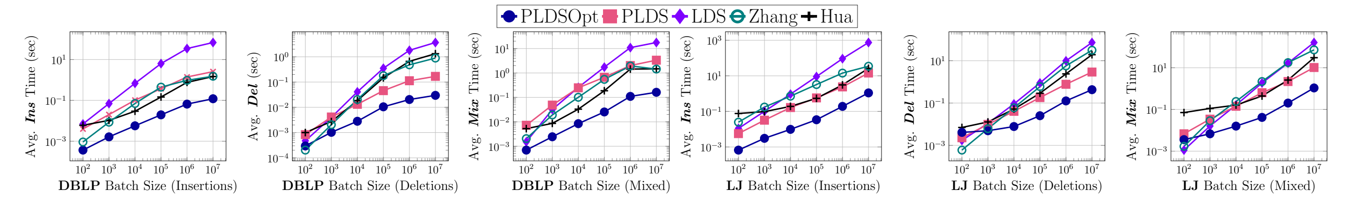

Figure 8: Average Ins, Del, and Mix per-batch running times on varying batch sizes for PLDSOpt, PLDS, LDS, Zhang, and Hua on *dblp* and *livejournal*.

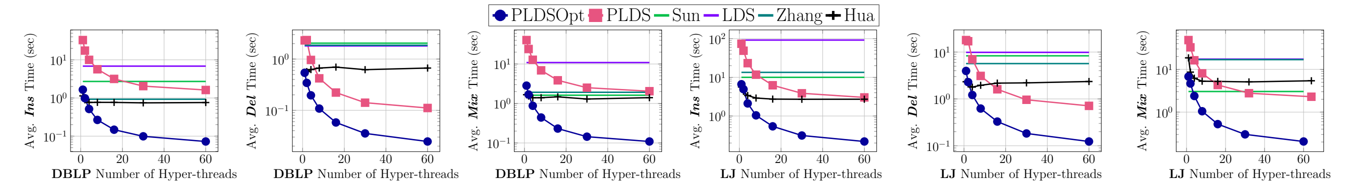

Figure 9: Parallel speedup of PLDSOpt, PLDS, and Hua, with respect to their single-threaded running times on *dblp* and *livejournal* on Ins, Del, and Mix batches of size 10^6 for all algorithms. The "60" on the x-axis indicates 30 cores with hyper-threading. LDS, Sun, and Zhang are shown as horizontal lines since they are sequential.

over Zhang, 15.79–52.36× speedups over Hua, 41.02–100.34× speedups over Sun, and 2.51–23.45× speedups over PLDS (except on *ctr* and *usa*). For **Mix**, it gets 17.54–723.72× speedups over Zhang, 11.34–91.95× over Hua, 6.95–35.59× speedups over Sun, and 2.81–18.68× speedups over PLDS (except on *ctr* and *usa*). These massive speedups over previous work demonstrate the utility of PLDSOpt not only on large graphs but also on smaller graphs. Notably, our PLDSOpt and PLDS algorithms perform not only well on dense networks but also on very sparse road networks. For *ctr* and *usa*, PLDS performs better than PLDSOpt, achieving up to a 1.09× speedup on **Del** and 1.12× speedup on **Mix**.

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

5.5 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. We tested the errors for $\delta = 0.4$ and $\lambda = 3$. According to our theoretical proofs, the maximum error (for PLDS) should be $(2 + 3/3)(1 + 0.4) = 4.2$. We confirm that the maximum empirical error for PLDS falls under this constraint. PLDSOpt achieves an average error of 1.24–2.37 compared to errors of 1.26–3.48 for PLDS, 1.01–4.17 for ApproxKCore, and 1.03–3.23 for Sun. PLDSOpt gets a maximum error of 3–6 compared to 2–4.19 for PLDS, 3–5 for ApproxKCore, and 3–5.99 for Sun. We conclude that our error bounds match those of the current best-known algorithms and are sufficiently small to be of use for many applications.

6 Additional Parallel Algorithms

Here, we provide an overview of the techniques and challenges for our other algorithms. Due to space constraints, we provide detailed

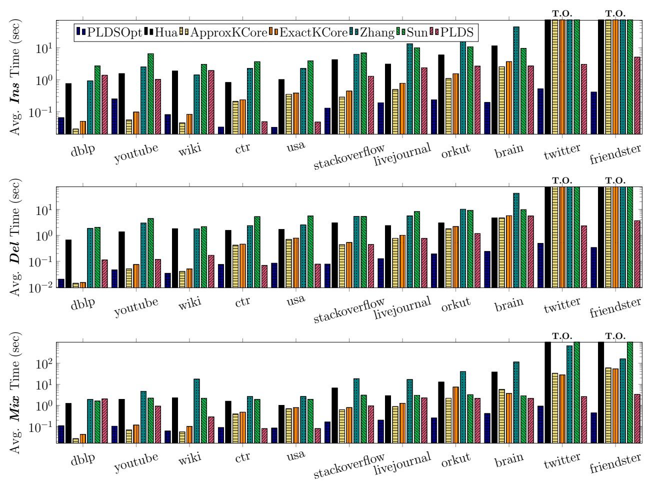

Figure 10: Average per-batch running times for PLDSOpt, Hua, PLDS, Sun, Zhang, ApproxKCore, and ExactKCore, on *dblp, youtube, wiki, ctr, usa, stackoverflow, livejournal, orkut, brain, twitter,* and *friendster* with batches of size 10^6 (and approximation settings $\delta = 0.4$ and $\lambda = 3$ for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

descriptions and proofs of each of these algorithms in our full paper [48]. The bounds for these algorithms are shown in Table 2.

6.1 Static k-Core Decomposition

We present a parallel, approximate, static k-core decomposition algorithm based on the parallel, exact algorithm of Dhulipala et al. [18]. Their algorithm is a bucketing-based algorithm where vertices are partitioned into buckets according to degree; vertices are then peeled and the partitioning may change as the vertices' degrees change. Although their algorithm is work-efficient, the depth could be linear. We show that combining their bucketing-based algorithm with our PLDS results in a parallel static $(2 + \varepsilon)$-approximate k-core decomposition algorithm with $O(m + n)$ expected work and $(\log^3 n)$ depth w.h.p.. Our experiments (Section 5) also demonstrate improvements over [18] on real-world networks.

6.2 Framework for Batch-Dynamic Graph Algorithms from Low Out-Degree Orientations

In this section, we introduce a framework that we will use in all of our batch-dynamic algorithms that use our batch-dynamic low out-degree orientation algorithm (Section 4.5). Our framework assumes three different methods for each of the problems (maximal matching, k-clique counting, and vertex coloring) that we solve. Specifically, these three methods handle batches of insertions and deletions separately and are instantiated in our full paper [48]; let BatchFlips, BatchInsert, and BatchDelete denote these three methods.

We assume for simplicity that all updates in the batch $\mathcal{B}$ are *unique*, which means that no edge deletion occurs on an inserted edge in the batch and vice versa. Furthermore, we assume that the updates are *valid*, meaning that if an edge insertion (u, v) is in $\mathcal{B}$, then (u, v) does not exist in the graph, and if an edge deletion (w, x) is in $\mathcal{B}$, then edge (w, x) exists in the graph. Such assumptions are only *simplifying* assumptions because it is easy to perform preprocessing on $\mathcal{B}$ in $O(|\mathcal{B}| \log n)$ work and $O(\log n)$ depth to ensure that these assumptions are satisfied. In fact, our implementations in Section 5 do perform this preprocessing on the input batches. To find all unique updates, we perform a parallel sort in $O(|\mathcal{B}| \log n)$ work and $O(\log n)$ depth [9, 19, 35]; we first sort on the edge and then the timestamp of the update. Then, we perform a parallel filter in $O(|\mathcal{B}|)$ work and $O(1)$ depth [9, 19, 35] where we keep each edge with the latest timestamp. Then, we perform another parallel filter to keep only edge insertions of nonexistent edges and edge deletions of edges that exist in the graph. This preprocessing ensures $\mathcal{B}$ follows our simplifying assumptions and do not exceed the complexity bounds of our PLDS, and hence, we assume all input batches contain unique and valid updates. The work and depth for preprocessing are subsumed by the bounds for the algorithms.

Algorithm 5 GraphProblemUpdate$(G, \mathcal{B})$

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

Detailed Framework The pseudocode for our framework is shown in Algorithm 5. We first update the PLDS by calling the update procedure (Algorithm 1) on the batch of updates in Line 1. Afterwards, we call our low out-degree orientation algorithm to obtain the set of edges that were flipped, placed in set A (Line 2). Then, we take the batch of updates $\mathcal{B}$ and split the batch into a batch of insertions, $\mathcal{B}_{ins}$, and a batch of deletions, $\mathcal{B}_{del}$ (Line 3). We call BatchFlips (Line 4) on the set of flipped edges A, which processes the edge flips accordingly for each problem. Finally, we call the problem specific functions BatchDelete and BatchInsert (Lines 5 and 6) on $\mathcal{B}_{del}$ and $\mathcal{B}_{ins}$, respectively; we first call BatchDelete and then BatchInsert.

Analysis By Corollary 3.3, our low out-degree orientation algorithm gives a $O(\alpha)$ out-degree orientation. Furthermore, the amortized work of the algorithm indicates that $O(|\mathcal{B}| \log^2 n)$ amortized flips occur with each batch $\mathcal{B}$. Suppose that BatchFlips(A) takes $O\left(|A| W_{flips}(\alpha)\right)$ work and $O\left(D_{flips}\right)$ depth; BatchInsert$(\mathcal{B}_{ins})$ takes $O\left(|\mathcal{B}_{ins}| W_{ins}(\alpha)\right)$ work and $O\left(D_{ins}\right)$ depth, and BatchDelete$\left(\mathcal{B}_{del}\right)$ takes $O\left(|\mathcal{B}_{del}| W_{del}(\alpha)\right)$ work and $O\left(D_{del}\right)$ depth; and the update methods require $O(S)$ space in total. Then, we show the following theorem about our framework. The proof is provided in our full paper [48].

Theorem 6.1. *Algorithm 5 takes*

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

amortized work and

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

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

6.3 Maximal Matching

The best-known sequential algorithm of Neiman and Solomon [56] (against an adaptive adversary) maintains the set of unmatched in-neighbors of each vertex. When a vertex becomes unmatched due to an edge deletion, it matches itself to an unmatched in-neighbor if possible. If none are unmatched, it checks all of its out-neighbors to see if any are unmatched. Unfortunately, such a simple algorithm does not work when given batches of updates since *many* vertices can become unmatched and we must match them with potentially the same set of in-neighbors. Because in-degree is unbounded, we cannot use a static algorithm on the induced subgraph of all unmatched in-neighbors, as this subgraph is too large.

Our algorithm instead first runs a static maximal matching algorithm on the induced subgraph consisting of all newly unmatched vertices and their unmatched out-neighbors. Because the out-degree is bounded, we can afford this. Then, for any vertex that remains unmatched, we progressively try to match it to an in-neighbor by doubling the number of in-neighbors we query with each attempt and running the static algorithm on the induced subgraph of the queried in-neighbors and the unmatched vertices. Our proof ensures that work-efficiency is maintained by this doubling procedure.

6.4 k-Clique Counting

The best-known batch-dynamic algorithm of Dhulipala et al. [20] uses a static parallel k-clique counting algorithm [61] to enumerate smaller cliques which are intersected with the update batch to produce k-cliques in $O(|\mathcal{B}|m\alpha^{k-4})$ expected work and $O(\log^{k-2} n)$ depth w.h.p., using $O(m + |\mathcal{B}|)$ space. Our algorithm, in contrast, dynamically maintains small cliques in memory, without ever using any static algorithms. Then, new updates are used to *complete* and turn partial structures into k-cliques. Specifically, we maintain for each *potential k-clique*, C, the *largest incomplete clique without a source*, C', where $C' \subset C$. When C' becomes a clique, then C becomes a k-clique (and deletions are treated symmetrically).

The main challenge is in maintaining these *partial* structures efficiently; provided edge updates, we cannot afford to naively update all structures formed by in-neighbors since the in-degree is unbounded. By keeping an intricate count of different structures in parallel hash tables, we are able to achieve better work than [20] when $m = \omega(\alpha^2 \log^2 n)$; such a condition holds for many graphs, e.g., the road networks and our densest graphs, *brain*, *orkut*, and *friendster*, as shown in Table 3.

6.5 Vertex Coloring

Henzinger et al. [33] present two currently best-known sequential, dynamic algorithms for vertex coloring, with complexity bounds in terms of α. Their first algorithm is robust against oblivious adversaries and uses the LDS to maintain disjoint color palettes for each level. Each vertex has a color from its level's palette. This invariant is maintained under edge insertions that cause conflicts and vertex level changes. In the batch-dynamic setting, the main challenge is that several vertices may be choosing a color simultaneously from the same palette of non-conflicting colors. We show that we can maintain work-efficiency in expectation and low depth w.h.p. by simply repeatedly choosing a non-conflicting color uniformly at random (even if multiple use the same palette) until all relevant vertices have chosen non-conflicting colors.

Henzinger et al. [33] also give an *implicit* vertex coloring algorithm. An implicit coloring structure is maintained after updates and provides valid colors for *queried* vertices. Their implicit coloring algorithm maintains a set of forests on the outgoing edges of vertices in an orientation of the input graph. Their original algorithm prevents cycles from forming in the forests because edges are added sequentially. However, this is challenging to parallelize, as it involves parallel cycle detection and then splitting the cycles across multiple trees. Instead, we present a simpler version of their algorithm that is easier to parallelize, provided an *acyclic* low out-degree orientation (which PLDS maintains), while guaranteeing the same properties. Our simpler algorithm is conducive to batch updates, where we make use of parallel Euler trees [65].

Acknowledgements

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

References

[1] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large Scale Networks Fingerprinting and Visualization Using the K-Core Decomposition. In *International Conference on Neural Information Processing Systems*.

[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 *ACM International Conference on Distributed and Event-Based Systems*. 161–168.

[5] Gary D. Bader and Christopher WV Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics* 4, 1 (Jan. 2003), 2.

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

[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)*. 173–182.

[8] 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)*.

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

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

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

[12] 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.

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

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

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

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

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

[18] 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.

[19] 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)*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[34] 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.

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

[36] 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.

[37] 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.

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

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

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

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

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

[43] 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.

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

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

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

[47] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Transactions on Knowledge and Data Engineering* 26, 10 (2014), 2453–2465.

[48] 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. https://arxiv.org/abs/2106.03824

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

[50] Qi Luo, Dongxiao Yu, Zhipeng Cai, Xuemin Lin, and Xiuzhen Cheng. 2021. Hypercore Maintenance in Dynamic Hypergraphs. In *IEEE ICDE*. 2051–2056.

[51] Qi Luo, Dongxiao Yu, Feng Li, Zhenhao Dou, Zhipeng Cai, Jiguo Yu, and Xiuzhen Cheng. 2019. Distributed Core Decomposition in Probabilistic Graphs. In *Computational Data and Social Networks*. 16–32.

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

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

[54] Sourav Medya, Tianyi Ma, Arlei Silva, and Ambuj Singh. 2020. A Game Theoretic Approach For K-Core Minimization. In *19th International Conference on Autonomous Agents and MultiAgent Systems*. 1922–1924.

[55] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. 2015. Scalable Large Near-Clique Detection in Large-Scale Networks via Sampling. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 815–824.

[56] Ofer Neiman and Shay Solomon. 2015. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. on Alg. (TALG)* 12, 1 (2015), 1–15.

[57] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. 4292–4293. http://networkrepository.com

[58] Ahmet Erdem Sariyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal* 25, 3 (2016), 425–447.

[59] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for K-Core Decomposition. *Proc. VLDB Endow.* 6, 6 (April 2013), 433–444.

[60] Saurabh Sawlani and Junxing Wang. 2020. Near-Optimal Fully Dynamic Densest Subgraph. In *ACM SIGACT Symposium on Theory of Computing*. 181–193.

[61] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Parallel Clique Counting and Peeling Algorithms. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*. 135–146.

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

[63] 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).

[64] Bintao Sun, T-H. Hubert Chan, and Mauro Sozio. 2020. *Fully Dynamic Approximate k-Core Decomposition in Hypergraphs*. https://github.com/btsun/DynHyperCoreDecomp

[65] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. 2019. Batch-Parallel Euler Tour Trees. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*. 92–106.

[66] Kai Wang, Xin Cao, Xuemin Lin, Wenjie Zhang, and Lu Qin. 2018. Efficient Computing of Radius-Bounded k-Cores. In *IEEE ICDE*. 233–244.

[67] Na Wang, Dongxiao Yu, Hai Jin, Chen Qian, Xia Xie, and Qiang-Sheng Hua. 2017. Parallel Algorithm for Core Maintenance in Dynamic Graphs. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2366–2371.

[68] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (Jan. 2015), 181–213.

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

[70] 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.

[71] Y. Zhang, J. Yu, Y. Zhang, and L. Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *IEEE ICDE*. 337–348.

[72] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and Efficiency of Truss Maintenance in Evolving Graphs. In *ACM SIGMOD International Conference on Management of Data*. 1024–1041.