Parallel Batch-Dynamic k-Clique Counting

Laxman Dhulipala
MIT CSAIL
laxman@mit.edu

Quanquan C. Liu
MIT CSAIL
quanquan@mit.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Shangdi Yu
MIT CSAIL
shangdiy@mit.edu

Abstract

In this paper, we study new batch-dynamic algorithms for the k-clique counting problem, which are dynamic algorithms where the updates are *batches* of edge insertions and deletions. We study this problem in the parallel setting, where the goal is to obtain algorithms with low (polylogarithmic) depth. Our first result is a new parallel batch-dynamic triangle counting algorithm with $O(\Delta\sqrt{\Delta+m})$ amortized work and $O(\log^*(\Delta+m))$ depth with high probability, and $O(\Delta+m)$ space for a batch of Δ edge insertions or deletions. Our second result is an algebraic algorithm based on parallel fast matrix multiplication. Assuming that a parallel fast matrix multiplication algorithm exists with parallel matrix multiplication constant ω_p, the same algorithm solves dynamic k-clique counting with $O\left(\min\left(\Delta m^{\frac{(2k-1)\omega_p}{3(\omega_p+1)}}, (\Delta+m)^{\frac{2(k+1)\omega_p}{3(\omega_p+1)}}\right)\right)$ amortized work and $O(\log(\Delta+m))$ depth with high probability, and $O\left((\Delta+m)^{\frac{2(k+1)\omega_p}{3(\omega_p+1)}}\right)$ space. Using a recently developed parallel k-clique counting algorithm, we also obtain a simple batch-dynamic algorithm for k-clique counting on graphs with arboricity α running in $O(\Delta(m+\Delta)\alpha^{k-4})$ expected work and $O(\log^{k-2} n)$ depth with high probability, and $O(m+\Delta)$ space. Finally, we present a multicore CPU implementation of our parallel batch-dynamic triangle counting algorithm. On a 72-core machine with two-way hyper-threading, our implementation achieves 36.54–74.73x parallel speedup, and in certain cases achieves significant speedups over existing parallel algorithms for the problem, which are not theoretically-efficient.

1 Introduction

Subgraph counting algorithms are fundamental graph analysis tools, with numerous applications in network classification in domains including social network analysis and bioinformatics. A particularly important type of subgraph for these applications is the triangle, or 3-clique—three vertices that are all mutually connected [New03]. Counting the number of triangles is a basic and fundamental task that is used in numerous social and network science measurements [Gra77, WS98].

In this paper, we study the triangle counting problem and its generalization to higher cliques from the perspective of dynamic algorithms. A k-clique consists of k vertices and all $\binom{k}{2}$ possible edges among them (for applications of k-cliques, see, e.g., [HR05]). As many real-world graphs change rapidly in real-time, it is crucial to design dynamic algorithms that efficiently maintain k-cliques upon updates, since the cost of re-computation from scratch can be prohibitive. Furthermore, due to the fact that dynamic updates can occur at a rapid rate in practice, it is increasingly important to design ***batch-dynamic*** algorithms which can take arbitrarily large batches of updates (edge insertions or deletions) as their input. Finally, since the batches, and corresponding update complexity can be large, it is also desirable to use parallelism to speed-up maintenance and design algorithms that map to modern parallel architectures.

Due to the broad applicability of k-clique counting in practice and the fact that k-clique counting is a fundamental theoretical problem of its own right, there has been a large body of prior work on the problem. Theoretically, the fastest static algorithm for arbitrary graphs uses fast matrix multiplication, and counts 3ℓ cliques in $O(n^{\ell\omega})$ time where ω is the matrix multiplication exponent [NP85]. Considerable effort has also been devoted to efficient combinatorial algorithms. Chiba and Nishizeki [CN85] show how to compute k-cliques in $O(\alpha^{k-2}m)$ work, where m is the number of edges in the graph and α is the arboricity of the graph. This algorithm was recently parallelized by Danisch et al. [DBS18a] (although not in polylogarithmic depth). Worst-case optimal join algorithms can perform k-clique counting in $O(m^{k/2})$ work as a special case [NPRR18, ALT+17]. Alon, Yuster, and Zwick [AYZ97] design an algorithm for triangle counting in the sequential model, based on fast matrix multiplication. Eisenbrand and Grandoni [EG04] then extend this result to k-clique counting based on fast matrix multiplication. Vassilevska designs a space-efficient combinatorial algorithm for k-clique counting [Vas09]. Finocchi et al. give clique counting algorithms for MapReduce [FFF15]. Jain and Seshadri provide probabilistic algorithms for estimating clique counts [JS17]. The k-clique problem is also a classical problem in parameterized-complexity, and is known to be $W[1]$-complete [DF95].

The problem of maintaining k-cliques under dynamic updates began more recently. Eppstein et al. [ES09, EGST12]

design sequential dynamic algorithms for maintaining size-3 subgraphs in $O(h)$ amortized time and $O(mh)$ space and size-4 subgraphs in $O(h^2)$ amortized time and $O(mh^2)$ space, where h is the h-index of the graph ($h = O(\sqrt{m})$). Ammar et al. extend the worst-case optimal join algorithms to the parallel and dynamic setting [AMSJ18]. However, their update time is not better than the static worst-case optimal join algorithm. Recently, Kara et al. [KNN+19] present a sequential dynamic algorithm for maintaining triangles in $O(\sqrt{m})$ amortized time and $O(m)$ space. Dvorak and Tuma [DT13] present a dynamic algorithm that maintains k-cliques as a special case in $O(\alpha^{k-2} \log n)$ amortized time and $O(\alpha^{k-2}m)$ space by using low out-degree orientations for graphs with arboricity α.

Designing Parallel Batch-Dynamic Algorithms. Traditional dynamic algorithms receive and apply updates one at a time. However, in the ***parallel batch-dynamic*** setting, the algorithm receives *batches of updates* one after the other, where each batch contains a mix of edge insertions and deletions. Unlike traditional dynamic algorithms, a parallel batch-dynamic algorithm can apply *all* of the updates together, and also take advantage of parallelism while processing the batch. We note that the edges inside of a batch may also be ordered (e.g., by a timestamp). If there are duplicate edge insertions within a batch, or an insertion of an edge followed by its deletion, a batch-dynamic algorithm can easily remove such redundant or nullifying updates.

The key challenge is to design the algorithm so that updates can be processed in parallel while ensuring low work and depth bounds. The only existing parallel batch-dynamic algorithms for k-clique counting are triangle counting algorithms by Ediger et al. [EJRB10] and Makkar et al. [MBG17], which take linear work per update in the worst case. The algorithms in this paper make use of efficient data structures such as parallel hash tables, which let us perform parallel batches of edge insertions and deletions with better work and (polylogarithmic) depth bounds. To the best of our knowledge, no prior work has designed dynamic algorithms for the problem that support parallel batch updates with non-trivial theoretical guarantees.

Theoretically-efficient parallel dynamic (and batch-dynamic) algorithms have been designed for a variety of other graph problems, including minimum spanning tree [KPR18, FL94, DF94], Euler tour trees [TDB19], connectivity [STTW18, AABD19, FL94], tree contraction [RT94, AAW17], and depth-first search [Kha17]. Very recently, parallel dynamic algorithms were also designed for the Massively Parallel Computation (MPC) setting [ILMP19, DDK+20].

Summary of Our Contributions. In this paper, we design parallel algorithms in the batch-dynamic setting, where the algorithm receives a batch of $\Delta \geq 1$ edge updates that can be processed in parallel. Our focus is on parallel batch-dynamic algorithms that admit strong theoretical bounds on their work and have polylogarithmic depth with high probability. Note that although our work bounds may be amortized, our depth will be polylogarithmic with high probability, leading to efficient RNC algorithms. As a special case of our results, we obtain algorithms for parallelizing single updates ($\Delta = 1$). We first design a parallel batch-dynamic triangle counting algorithm based on the sequential algorithm of Kara et al. [KNN+19]. For triangle counting, we obtain an algorithm that takes $O(\Delta\sqrt{\Delta + m})$ amortized work and $O(\log^*(\Delta + m))$ depth w.h.p.[1] assuming a fetch-and-add instruction that runs in $O(1)$ work and depth, and runs in $O(\Delta + m)$ space. The work of our parallel algorithm matches that of the sequential algorithm of performing one update at a time (i.e., it is work-efficient), and we can perform all updates in parallel with low depth.

We then present a new parallel batch-dynamic algorithm based on fast matrix multiplication. Using the best currently known parallel matrix multiplication [Wil12, LG14], our algorithm dynamically maintains the number of k-cliques in $O\left(\min\left(\Delta m^{0.469k-0.235}, (\Delta + m)^{0.469k+0.469}\right)\right)$ amortized work w.h.p. per batch of Δ updates where m is defined as the maximum number of edges in the graph before and after all updates in the batch are applied. Our approach is based on the algorithm of [AYZ97, EG04, NP85], and maintains triples of $k/3$-cliques that together form k-cliques. The depth is $O(\log(\Delta + m))$ w.h.p. and the space is $O\left((\Delta + m)^{0.469k+0.469}\right)$. Our results also imply an amortized time bound of $O\left(m^{0.469k-0.235}\right)$ per update for dense graphs in the sequential setting. Of potential independent interest, we present the first proof of logarithmic depth in the parallelization of any tensor-based fast matrix multiplication algorithms. We also give a simple batch-dynamic k-clique listing algorithm, based on enumerating smaller cliques and intersecting them with edges in the batch. The algorithm runs in $O(\Delta(m + \Delta)\alpha^{k-4})$ expected work, $O(\log^{k-2} n)$ depth w.h.p., and $O(m + \Delta)$ space.

Finally, we implement our new parallel batch-dynamic triangle counting algorithm for multicore CPUs, and present some experimental results on large graphs and with varying batch sizes using a 72-core machine with two-way hyper-threading. We found our parallel implementation to be much faster than the multicore implementation of Ediger et al. [EJRB10]. We also developed an optimized multicore implementation of the GPU algorithm by Makkar et al. [MBG17]. We found that our new algorithm is up to an order of magnitude faster than our CPU implementation of the Makkar et al. algorithm, and our new algorithm achieves 36.54–74.73x parallel speedup on 72 cores with hyper-threading. Our code is publicly available at `https://github.com/ParAlg/gbbs`.

[1] We use "with high probability" (w.h.p.) to mean with probability at least $1 - 1/n^c$ for any constant $c > 0$.

2 Preliminaries

Given an undirected graph $G = (V, E)$ with n vertices and m edges, and an integer k, a ***k-clique*** is defined as a set of k vertices $v_1, \ldots, v_k$ such that for all $i \neq j$, $(v_i, v_j) \in E$. The ***k-clique count*** is the total number of k-cliques in the graph. The ***dynamic k-clique problem*** maintains the number of k-cliques in the graph upon edge insertions and deletions, given individually or in a batch. The ***arboricity*** α of a graph is the minimum number of forests that the edges can be partitioned into and its value is between $\Omega(1)$ and $O(\sqrt{m})$ [CN85].

In this paper, we analyze algorithms in the work-depth model, where the ***work*** of an algorithm is defined to be the total number of operations done, and the ***depth*** is defined to be the longest sequential dependence in the computation (or the computation time given an infinite number of processors) [Jaj92]. Our algorithms can run in the PRAM model or the fork-join model with arbitrary forking. We use the concurrent-read concurrent-write (CRCW) model, where reads and writes to a memory location can happen concurrently. We assume either that concurrent writes are resolved arbitrarily, or are reduced together (i.e., fetch-and-add PRAM).

We use the following primitives throughout the paper. ***Approximate compaction*** takes a set of m objects in the range $[1, n]$ and allocates them unique IDs in the range $[1, O(m)]$. The primitive is useful for filtering (i.e., removing) out a set of obsolete elements from an array of size n, and mapping the remaining m elements to a sparse array of size $O(m)$. Approximate compaction can be implemented in $O(n)$ work and $O(\log^* n)$ depth w.h.p. [GMV91]. We also use a ***parallel hash table*** which supports n operations (insertions, deletions) in $O(n)$ work and $O(\log^* n)$ depth w.h.p., and n lookup operations in $O(n)$ work and $O(1)$ depth [GMV91].

Our algorithms in this paper make use of the widely used ***atomic-add*** instruction. An atomic-add instruction takes a memory location and atomically increments the value stored at the location. In this paper, we assume that the atomic-add instruction can be implemented in $O(1)$ work and depth. Our algorithms can also be implemented in a model without atomic-add in the same work, a multiplicative $O(\log n)$ factor increase in the depth, and space proportional to the number of atomic-adds done in parallel.

3 Parallel Batch-Dynamic Triangle Counting

In this section, we present our parallel batch-dynamic triangle counting algorithm, which is based on the $O(m)$ space and $O(\sqrt{m})$ amortized update, sequential, dynamic algorithm of Kara et al. [KNN+19]. Theorem 3.1 summarizes the guarantees of our algorithm.

THEOREM 3.1. *There exists a parallel batch-dynamic triangle counting algorithm that requires $O(\Delta(\sqrt{\Delta + m}))$ amortized work and $O(\log^*(\Delta + m))$ depth with high probability, and $O(\Delta + m)$ space for a batch of Δ edge updates.*

Our algorithm is work-efficient and achieves a significantly lower depth for a batch of updates than applying the updates one at a time using the sequential algorithm of [KNN+19]. We provide a detailed description of the fully dynamic sequential algorithm of [KNN+19] in the full version of our paper [DLSY20] for reference,[2] and a brief high-level overview of that algorithm in this section.

3.1 Sequential Algorithm Overview

Given a graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, let $M = 2m + 1$, $t_1 = \sqrt{M}/2$, and $t_2 = 3\sqrt{M}/2$. We classify a vertex as ***low-degree*** if its degree is at most t_1 and ***high-degree*** if its degree is at least t_2. Vertices with degree in between t_1 and t_2 can be classified either way.

Data Structures. The algorithm partitions the edges into four edge-stores $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ based on a degree-based partitioning of the vertices. $\mathcal{HH}$ stores all of the edges (u, v), where both u and v are high-degree. $\mathcal{HL}$ stores edges (u, v), where u is high-degree and v is low-degree. $\mathcal{LH}$ stores the edges (u, v), where u is low-degree and v is high-degree. Finally, $\mathcal{LL}$ stores edges (u, v), where both u and v are low-degree.

The algorithm also maintains a wedge-store $\mathcal{T}$ (a wedge is a triple of distinct vertices (x, y, z) where both (x, y) and (y, z) are edges in E). For each pair of high-degree vertices u and v, the wedge-store $\mathcal{T}$ stores the number of wedges (u, w, v), where w is a low-degree vertex. $\mathcal{T}$ has the property that given an edge insertion (resp. deletion) (u, v) where both u and v are high-degree vertices, it returns the number of wedges (u, w, v), where w is low-degree, that u and v are part of in $O(1)$ expected time. $\mathcal{T}$ is implemented via a hash table indexed by pairs of high-degree vertices that stores the number of wedges for each pair.

Finally, we have an array containing the degrees of each vertex, $\mathcal{D}$.

Initialization. Given a graph with m edges, the algorithm first initializes the triangle count C using a static triangle counting algorithm in $O(\alpha m) = O(m^{3/2})$ work and $O(m)$ space [Lat08]. The $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ tables are created by scanning all edges in the input graph and inserting them into the appropriate hash tables. $\mathcal{T}$ can be initialized by iterating over edges (u, w) in $\mathcal{HL}$ and for each w, iterating over all edges (w, v) in $\mathcal{LH}$ to find pairs of high-degree vertices u and v, and then incrementing $\mathcal{T}(u, v)$.

The Kara et al. Algorithm [KNN+19]. Given an edge insertion (u, v) (deletions are handled similarly, and for simplicity assume that the edge does not already exist in G), the update algorithm must identify all tuples (u, w, v) where (u, w) and (v, w) already exist in G, since such triples correspond to new triangles formed by the edge insertion.

[2] Kara et al. [KNN+19] described their algorithm for counting directed 3-cycles in relational databases, where each triangle edge is drawn from a different relation, and we simplified it for the case of undirected graphs.

The algorithm proceeds by considering how a triangle's edges can reside in the data structures. For example, if all of u, v, and w are high-degree, then the algorithm will enumerate these triangles by checking $\mathcal{HH}$ and finding all neighbors w of u that are also high-degree (there are at most $O(\sqrt{m})$ such neighbors), checking if the (v, w) edge exists in constant time. On the other hand, if u is low-degree, then checking its $O(\sqrt{m})$ many neighbors suffices to enumerate all new triangles. The interesting case is if both u and v are high-degree, but w is low-degree, since there can be much more than $O(\sqrt{m})$ such w's. This case is handled using $\mathcal{T}$, which stores for a given (u, v) edge in $\mathcal{HH}$ all w such that (w, u) and (w, v) both exist in $\mathcal{LH}$.

Finally, the algorithm updates the data structures, first inserting the new edge into the appropriate edge-store. The algorithm updates $\mathcal{T}$ as follows. If u and v are both low-degree or both high-degree, then no update is needed to $\mathcal{T}$. Otherwise, without loss of generality suppose u is low-degree and v is high-degree. Then, the algorithm enumerates all high-degree vertices w that are neighbors of u and increments the value of (v, w) in $\mathcal{T}$.

3.2 Parallel Batch-Dynamic Update Algorithm

We present a high-level overview of our parallel algorithm in this section, and a more detailed description in Section 3.3. We consider batches of Δ edge insertions and/or deletions. Let $\texttt{insert}(u, v)$ represent the update corresponding to inserting an edge between vertices u and v, and $\texttt{delete}(u, v)$ represent deleting the edge between u and v. We first preprocess the batch to account for updates that *nullify* each other. For example, an $\texttt{insert}(u, v)$ update followed chronologically by a $\texttt{delete}(u, v)$ update nullify each other because the (u, v) edge that is inserted is immediately deleted, resulting in no change to the graph. To process the batch consisting of nullifying updates, we claim that the only update that is not nullifying for any pair of vertices is the chronologically last update in the batch for that edge. Since all updates contain a timestamp, to account for nullifying updates we first find all updates on the same edge by hashing the updates by the edge that it is being performed on. Then, we run the parallel maximum-finding algorithm given in [Vis10] on each set of updates for each edge in parallel. This maximum-finding algorithm then returns the update with the largest timestamp (the most recent update) from the set of updates for each edge. This set of returned updates then composes a batch of non-nullifying updates.

Before we go into the details of our parallel batch-dynamic triangle counting algorithm, we first describe some challenges that must be solved in using Kara et al. [KNN+19] for the parallel batch-dynamic setting.

Challenges. Because Kara et al. [KNN+19] only considers one update at a time in their algorithm, they do not deal with cases where a set of two or more updates creates a new triangle. Since, in our setting, we must account for batches of multiple updates, we encounter the following set of challenges:

(**1**) We must be able to efficient find new triangles that are created via two or more edge insertions.

(**2**) We must be able to handle insertions and deletions simultaneously meaning that a triangle with one inserted edge and one deleted edge should not be counted as a new triangle.

(**3**) We must account for over-counting of triangles due to multiple updates occurring simultaneously.

For the rest of this section, we assume that $\Delta \leq m$, as otherwise we can re-initialize our data structure using the static parallel triangle-counting algorithm [ST15][3] to get the count in $O(\Delta^{3/2})$ work, $O(\log^* \Delta)$ depth, and $O(\Delta)$ space (assuming atomic-add), which is within the bounds of Theorem 3.1.

Parallel Initialization. Given a graph with m edges, we initialize the triangle count C using a static parallel triangle counting algorithm in $O(\alpha m) = O(m^{3/2})$ work, $O(\log^* m)$ depth, and $O(m)$ space [ST15], using atomic-add. We initialize $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ by scanning the edges in parallel and inserting them into the appropriate parallel hash tables. We initialize the degree array $\mathcal{D}$ by scanning the vertices. Both steps take $O(m)$ work and $O(\log^* m)$ depth w.h.p. $\mathcal{T}$ can be initialized by iterating over edges (u, w) in $\mathcal{HL}$ in parallel and for each w, iterating over all edges (w, v) in $\mathcal{LH}$ in parallel to find pairs of high-degree vertices u and v, and then incrementing $\mathcal{T}(u, v)$. The number of entries in $\mathcal{HL}$ is $O(m)$ and each w has $O(\sqrt{m})$ neighbors in $\mathcal{LH}$, giving a total of $O(m^{3/2})$ work and $O(\log^* m)$ depth w.h.p. for the hash table insertions. The amortized work per edge update is $O(\sqrt{m})$.

Data Structure Modifications. We now describe additional information that is stored in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, $\mathcal{LL}$, and $\mathcal{T}$, which is used by the batch-dynamic update algorithm:

(**1**) Every edge stored in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ stores an associated state, indicating whether it is an ***old edge***, a ***new insertion*** or a ***new deletion***, which correspond to the values of 0, 1, and 2, respectively.

(**2**) $\mathcal{T}(u, v)$ stores a tuple with 5 values instead of a single value for each index (u, v). Specifically, a 5-tuple entry of $\mathcal{T}(u, v) = (t_1^{(u,v)}, t_2^{(u,v)}, t_3^{(u,v)}, t_4^{(u,v)}, t_5^{(u,v)})$ represents the following:

- $t_1^{(u,v)}$ represents the number of wedges with endpoints u and v that include only old edges.
- $t_2^{(u,v)}$ and $t_3^{(u,v)}$ represent the number of wedges with endpoints u and v containing one or two newly inserted edges, respectively.
- $t_4^{(u,v)}$ and $t_5^{(u,v)}$ represent the number of wedges with

[3] The hashing-based version of the algorithm given in [ST15] can be modified to obtain the stated bounds if it does not do ranking and when using the $O(\log^* n)$ depth w.h.p. parallel hash table and uses atomic-add.

endpoints u and v containing one or two newly deleted edges, respectively. In other words, they are wedges that do not exist anymore due to one or two edge deletions.

Algorithm Overview. We first remove updates in the batch that either insert edges already in the graph or delete edges not in the graph by using approximate compaction to filter. Next, we update the tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ with the new edge insertions. Recall that we must update the tables with both (u, v) and (v, u) (and similarly when we update these tables with edge deletions). We also mark these edges as newly inserted. Next, we update $\mathcal{D}$ with the new degrees of all vertices due to edge insertions. Since the degrees of some vertices have now increased, for low-degree vertices whose degree exceeds t_2, in parallel, we promote them to high-degree vertices, which involves updating the tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, $\mathcal{LL}$, and $\mathcal{T}$. Next, we update the tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ with new edge deletions, and mark these edges as newly deleted. We then call the procedures `update_table_insertions` and `update_table_deletions`, which update the wedge-table $\mathcal{T}$ based on all new insertions and all new deletions, respectively. At this point, our auxiliary data structures contain both new triangles formed by edge insertions, and triangles deleted due to edge deletions.

For each update in the batch, we then determine the number of new triangles that are created by counting different types of triangles that the edge appears in (based on the number of other updates forming the triangle). We then aggregate these per-update counts to update the overall triangle count.

Now that the count is updated, the remaining steps of the algorithm handle unmarking the edges and restoring the data structures so that they can be used by the next batch. We unmark all newly inserted edges from the tables, and delete all edges marked as deletes in this batch. Finally, we handle updating $\mathcal{T}$, the wedge-table for all insertions and deletions of edges incident to low-degree vertices. The last steps in our algorithm are to update the degrees in response to the newly inserted edges and the now truly deleted edges. Then, since the degrees of some high-degree vertices may drop below t_1 (and vice versa), we convert them to low-degree vertices and update the tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, $\mathcal{LL}$, and $\mathcal{T}$ (and vice versa). This step is called ***minor rebalancing***. Finally, if the number of edges in the graph becomes less than $M/4$ or greater than M we reset the values of M, t_1, and t_2, and re-initialize all of the data structures. This step is called ***major rebalancing***.

Algorithm Description. A simplified version of our algorithm is shown below. The following COUNT-TRIANGLE procedure takes as input a batch of Δ updates $\mathcal{B}$ and returns the count of the updated number of triangles in the graph (assuming the initialization process has already been run on the input graph and all associated data structures are up-to-date).

Algorithm 1 Simplified parallel batch-dynamic triangle counting algorithm.

1: **function** COUNT-TRIANGLES($\mathcal{B}$)
2: **parfor** `insert`$(u, v) \in \mathcal{B}$ **do**
3: Update and label edges (u, v) and (v, u) in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ as inserted edges.
4: **parfor** `delete`$(u, v) \in \mathcal{B}$ **do**
5: Update and label edges (u, v) and (v, u) in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ as deleted edges.
6: **parfor** `insert`$(u, v) \in \mathcal{B}$ or `delete`$(u, v) \in \mathcal{B}$ **do**
7: Update $\mathcal{T}$ with (u, v). $\mathcal{T}$ records the number of wedges that have 0, 1, or 2 edge updates.
8: **parfor** `insert`$(u, v) \in \mathcal{B}$ or `delete`$(u, v) \in \mathcal{B}$ **do**
9: Count the number of new triangles and deleted triangles incident to edge (u, v), and account for duplicates.
10: Rebalance data structures if necessary.

Small Example Batch Updates. Here we provide a small example of processing a batch of updates. We assume that no rebalancing occurs. Suppose we have a batch of updates containing an edge insertion (u, v) with timestamp 3, an edge deletion (w, x) with timestamp 1, and an edge deletion (u, v) with timestamp 2. Since the edge insertion (u, v) has the later timestamp, it is the update that remains. After removing nullifying updates, the two updates that remain are insertion of (u, v) and deletion of (w, x). The algorithm first looks in $\mathcal{D}$ to find the degrees of u, v, w, and x in parallel. Suppose u, v, and w are high-degree and x is low-degree. We need to first update our data structures with the new edge updates. To update the data structure, we first update the edge table $\mathcal{HH}$ with (u, v) marked as an edge insertion. Then, we update the edge tables $\mathcal{HL}$ and $\mathcal{LH}$ with (w, x) as an edge deletion. Finally, we update the counts of wedges in $\mathcal{T}$ with (w, x)'s deletion. Specifically, for each of x's neighbors y in $\mathcal{LH}$, we update $\mathcal{T}(w, y)$ by incrementing $t_4^{(w,y)}$ (since (x, y) is not a new update).

After updating the data structures, we can count the changes to the total number of triangles in the graph. All of the following actions can be performed in parallel. Suppose that u comes lexicographically before v. We count the number of neighbors of u in $\mathcal{HH}$ and this will be the number of new triangles containing three high-degree vertices. To avoid overcounting, we do not count the number of high-degree neighbors of v. Since we are counting the number of triangles containing updates, we also do not count the number of high-degree neighbors of w since (w, x) cannot be part of any new triangles containing three high-degree vertices. Then, in parallel, we count the number of neighbors of x in $\mathcal{LL}$ and $\mathcal{LH}$; this is the number of deleted triangles containing one and two high-degree vertices, respectively. We use $\mathcal{T}$ to count the number of triangles containing one low-degree vertex and

(u, v). To count the number of inserted triangles containing (u, v) and a low-degree vertex, we look up $t_1^{(u,v)}$ in $\mathcal{T}$ and add it to our final triangle count; all other stored count values for (u, v) in $\mathcal{T}$ are 0 since there are no other new updates incident to u or v.

3.3 Parallel Batch-Dynamic Triangle Counting Detailed Algorithm

The detailed pseudocode of our parallel batch-dynamic triangle counting algorithm are shown below. Recall that the update procedure for a set of $\Delta \leq m$ non-nullifying updates is as follows (the subroutines used in the following steps are described afterward).

Algorithm 2 Detailed parallel batch-dynamic triangle counting procedure.

(1) Remove updates that insert edges already in the graph or delete edges not in the graph as well as nullifying updates using approximate compaction.

(2) Update tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ with the new edge insertions using $\texttt{insert}(u, v)$ and $\texttt{insert}(v, u)$. Mark these edges as newly inserted by running $\texttt{mark\_inserted\_edges}(\mathcal{B})$ on the batch of updates $\mathcal{B}$.

(3) Update tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ with new edge deletions using $\texttt{delete}(u, v)$ and $\texttt{delete}(v, u)$. Mark these edges as newly deleted using $\texttt{mark\_deleted\_edges}(\mathcal{B})$ on $\mathcal{B}$.

(4) Call $\texttt{update\_table\_insertions}(\mathcal{B})$ for the set $\mathcal{B}$ of all edge insertions $\texttt{insert}(u, w)$, where either u or w is low-degree and the other is high-degree.

(5) Call $\texttt{update\_table\_deletions}(\mathcal{B})$ for the set $\mathcal{B}$ of all edge deletions $\texttt{delete}(u, w)$ where either u or w is low-degree and the other is high-degree.

(6) For each update in the batch, determine the number of new triangles that are created by counting 6 values. Count the values using a 6-tuple, $(c_1, c_2, c_3, c_4, c_5, c_6)$ based on the number of other updates contained in a triangle:

(a) For each edge insertion $\texttt{insert}(u, v)$ resulting in a triangle containing only one newly inserted edge (and no deleted edges), increment c_1 by $\texttt{count\_triangles}(1, 0, \texttt{insert}(\texttt{u}, \texttt{v}))$.

(b) For each edge insertion $\texttt{insert}(u, v)$ resulting in a triangle containing two newly inserted edges (and no deleted edges), increment c_2 by $\texttt{count\_triangles}(2, 0, \texttt{insert}(\texttt{u}, \texttt{v}))$.

(c) For each edge insertion $\texttt{insert}(u, v)$ resulting in a triangle containing three newly inserted edges, increment c_3 by $\texttt{count\_triangles}(3, 0, \texttt{insert}(\texttt{u}, \texttt{v}))$.

(d) For each edge deletion $\texttt{delete}(u, v)$ resulting in a deleted triangle with one newly deleted edge, increment c_4 by $\texttt{count\_triangles}(0, 1, \texttt{delete}(\texttt{u}, \texttt{v}))$.

(e) For each edge deletion $\texttt{delete}(u, v)$ resulting in a deleted triangle with two newly deleted edges, increment c_5 by $\texttt{count\_triangles}(0, 2, \texttt{delete}(\texttt{u}, \texttt{v}))$.

(f) For each edge deletion $\texttt{delete}(u, v)$ resulting in a deleted triangle with three newly deleted edges, increment c_6 by $\texttt{count\_triangles}(0, 3, \texttt{delete}(\texttt{u}, \texttt{v}))$.

Let C be the previous count of the number of triangles. Update C to be $C + c_1 + (1/2)c_2 + (1/3)c_3 - c_4 - (1/2)c_5 - (1/3)c_6$, which becomes the new count.

(7) Scan through updates again. For each update, if the value stored in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and/or $\mathcal{LL}$ is 2 (a deleted edge), remove this edge. If stored value is 1 (an inserted edge), change the value to 0. For all updates where the endpoints are both high-degree or both low-degree, we are done. For each update (u, w) where either u or w is low-degree (assume without loss of generality that w is) and the other is high-degree, look for all high-degree neighbors v of w and update $\mathcal{T}(u, v)$ by summing all c_1, c_2,, and c_3 of the tuple and subtracting c_4 and c_5.

(8) Update $\mathcal{D}$ with the new degrees.

(9) Perform minor rebalancing for all vertices v that exceed t_2 in degree or fall under t_1 in parallel using $\texttt{minor\_rebalance}(\texttt{v})$. This makes a formerly low-degree vertex high-degree (and vice versa) and updates relevant structures.

(10) Perform major rebalancing if necessary (i.e., the total number of edges in the graph is less than $M/4$ or greater than M). Major rebalancing re-initializes all structures.

Procedure $\texttt{mark\_inserted\_edges}(\mathcal{B})$**.** We scan through each of the $\texttt{insert}(u, v)$ updates in $\mathcal{B}$ and mark (u, v) and (v, u) as newly inserted edges in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and/or $\mathcal{LL}$ by storing a value of 1 associated with the edge.

Procedure $\texttt{mark\_deleted\_edges}(\mathcal{B})$**.** Because we removed all nullifying updates before $\mathcal{B}$ is passed into the procedure, none of the deletion updates in $\mathcal{B}$ should delete newly inserted edges. For all edge deletions $\texttt{delete}(u, v)$, we change the values stored under (u, v) and (v, u) from 0 to 2 in the tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and/or $\mathcal{LL}$.

Procedure $\texttt{update\_table\_insertions}(\mathcal{B})$**.** For each $(u, w) \in \mathcal{B}$, assume without loss of generality that w is the low-degree vertex and do the following. We first find all of w's neighbors, v, in $\mathcal{LH}$ in parallel. Then, we determine for each neighbor v if (w, v) is new (marked as 1). If the edge (w, v) is not new, then increment the second value stored in the tuple with index $\mathcal{T}(u, v)$. If (w, v) is newly inserted, then increment the third value stored in $\mathcal{T}(u, v)$. The first, fourth, and fifth values stored in $\mathcal{T}(u, v)$ do not change in this step. The first, second, and third values count the number of edge insertions contained in the wedge keyed by (u, v). The first value counts all wedges with endpoints u and v that do not

contain any edge update, the second count the number of wedges containing one edge insertion, and the third counts the number of wedges containing two edge insertions. Then, intuitively, the first, second, and third values will tell us later for edge insertion (u, v) between two high-degree vertices whether newly created triangles containing (u, v) have one (the only update being (u, v)), two, or three, respectively, new edge insertions from the batch update. We do not update the edge insertion counts of wedges which contain a mix of edge insertion updates and edge deletion updates.

Procedure `update_table_deletions`($\mathcal{B}$)**.** For each $(u, w) \in \mathcal{B}$, assume without loss of generality that w is the low-degree vertex and do the following. We first find all of w's neighbors, v, in $\mathcal{LH}$ in parallel. Then, we determine for each neighbor v if (w, v) is a newly deleted edge (marked as 2). If (w, v) is not a newly deleted edge, increment the fourth value in the tuple stored in $\mathcal{T}(u, v)$ and decrement the first value. Otherwise, if (w, v) is a newly deleted edge, increment the fifth value of $\mathcal{T}(u, v)$ and decrement the first value. The second and third values in $\mathcal{T}(u, v)$ do not change in this step. For any key (u, v), the first, fourth, and fifth values gives the number of wedges with endpoints u and v that contain zero, one, or two edge deletions, respectively. Intuitively, the first, fourth, and fifth values tell us later whether newly deleted triangles have one (where the only edge deletion is (u, v)), two, or three, respectively, new edge deletions from the batch update.

Procedure `count_triangles`(`i`, `d`, `update`)**.** This procedure returns the number of triangles containing the update `insert`(u, v) or `delete`(u, v) and exactly i newly inserted edges or exactly d newly deleted edges (the update itself counts as one newly inserted edge or one newly deleted edge). If at least one of u or v is low-degree, we search in the tables, $\mathcal{LH}$, and $\mathcal{LL}$ for neighbors of the low-degree vertex and the number of marked edges per triangle: edges marked as 1 for insertion updates and edges marked as 2 for deletion updates. If both u and v are high-degree, we first look through all high-degree vertices using $\mathcal{HH}$ to see if any form a triangle with both high-degree endpoints u and v of the update. This allows us to find all newly updated triangles containing only high-degree vertices. Then, we confirm the existence of a triangle for each neighbor found in the tables by checking for the third edge in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, or $\mathcal{LL}$. We return only the counts containing the correct number of updates of the correct type. To avoid double counting for each update we do the following. Suppose all vertices are ordered lexicographically in some order. For any edge which contains two high-degree or two low-degree vertices, we search in $\mathcal{LL}$, $\mathcal{HH}$, and $\mathcal{LH}$ for exactly one of the two endpoints, the one that is lexicographically smaller.

Then, we return a tuple in $\mathcal{T}(u, v)$ based on the values of i and d to determine the count of triangles containing u and v and one low-degree vertex:

- Return the first value $t_1^{(u,v)}$ if either $i = 1$ or $d = 1$.
- Return the second value $t_2^{(u,v)}$ if $i = 2$.
- Return the third value $t_3^{(u,v)}$ if $i = 3$.
- Return the fourth value $t_4^{(u,v)}$ if $d = 2$.
- Return the fifth value $t_5^{(u,v)}$ if $d = 3$.

Note that we ignore all triangles that include more than one insertion update *and* more than one deletion update.

Procedure `minor_rebalance`(`u`)**.** This procedure performs a minor rebalance when either the degree of u decreases below t_1 or increases above t_2. We move all edges in $\mathcal{HH}$ and $\mathcal{HL}$ to $\mathcal{LH}$ and $\mathcal{LL}$ and vice versa. We also update $\mathcal{T}$ with new pairs of vertices that became high-degree and delete pairs that are no longer both high-degree.

3.4 Analysis

We prove the correctness of our algorithm in the following theorem. The proof is based on accounting for the contributions of an edge to each triangle that it participates in based on the number of other updated edges found in the triangle.

THEOREM 3.2. *Our parallel batch-dynamic algorithm maintains the number of triangles in the graph.*

Proof. All triangles containing at least one low-degree vertex can be found either in $\mathcal{T}$ or by searching through $\mathcal{LH}$ and $\mathcal{LL}$. All triangles containing all high-degree vertices can be found by searching $\mathcal{HH}$. Suppose that an edge update `insert`(u, v) (resp. `delete`(u, v)) is part of $I_{(u,v)}$ (resp. $D_{(u,v)}$) triangles. We need to add or subtract from the total count of triangles $I_{(u,v)}$ or $D_{(u,v)}$, respectively. However, some of the triangles will be counted twice or three times if they contain more than one edge update. By dividing each triangle count by the number of updated edges they contain, each triangle is counted exactly once for the total count C. ∎

Overall Bound. We now prove that our parallel batch-dynamic algorithm runs in $O(\Delta\sqrt{\Delta + m})$ work, $O(\log^*(\Delta + m))$ depth, and uses $O(\Delta + m)$ space. Henceforth, we assume that our algorithm uses the atomic-add instruction (see Section 2). Removing nullifying updates takes $O(\Delta)$ total work, $O(\log^* \Delta)$ depth w.h.p., and $O(\Delta)$ space for hashing and the find-maximum procedure outlined in Section 3.2. In step **(1)**, we perform table lookups for the updates into $\mathcal{D}$ and in $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, or $\mathcal{LL}$, followed by approximate compaction to filter. The hash table lookups take $O(\Delta)$ work and $O(\log^* m)$ depth with high probability and $O(m)$ space. Approximate compaction [GMV91] takes $O(\Delta)$ work, $O(\log^* \Delta)$ depth, and $O(\Delta)$ space. Steps **(2)**, **(3)**, and **(8)** perform hash table insertions and updates on the batch of $O(\Delta)$ edges, which takes $O(\Delta)$ amortized work and $O(\log^* m)$ depth with high probability.

The next lemma shows that updating the tables based on the edges in the update (steps **(4)** and **(5)**) can be done

in $O(\Delta\sqrt{m})$ work and $O(\log^* m)$ depth w.h.p., and $O(m)$ space.

LEMMA 3.1. `update_table_insertions`($\mathcal{B}$) *and* `update_table_deletions`($\mathcal{B}$) *on a batch $\mathcal{B}$ of size Δ takes $O(\Delta\sqrt{m})$ work and $O(\log^*(\Delta+m))$ depth w.h.p., and $O(\Delta+m)$ space.*

Proof. For each w, we find all of its high-degree neighbors in $\mathcal{LH}$ and perform the increment or decrement in the corresponding entry in $\mathcal{T}$ in parallel (at this point, the vertices are still classified based on their original degrees). The total number of new neighbors gained across all vertices is $O(\Delta)$ since there are Δ updates. Therefore, across all updates, this takes $O(\Delta\sqrt{m}+\Delta)$ work and $O(\log^*(\Delta+m))$ depth w.h.p. due to hash table lookup and updates. Then, for all high-degree neighbors found, we perform the increments or decrements on the corresponding entries in $\mathcal{T}$ in parallel, taking the same bounds. All vertices can be processed in parallel, giving a total of $O(\Delta\sqrt{m}+\Delta)$ work and $O(\log^*(\Delta+m))$ depth w.h.p. ∎

The next lemma bounds the complexity of updating the triangle count in step **(6)**.

LEMMA 3.2. *Updating the triangle count takes $O(\Delta\sqrt{m})$ work and $O(\log^*(\Delta+m))$ depth w.h.p., and $O(\Delta+m)$ space.*

Proof. We initialize $c_1,\ldots,c_6$ to 0. For each edge update in $\mathcal{B}$ where both endpoints are high-degree, we perform lookups in $\mathcal{T}$ and $\mathcal{HH}$ for the relevant values in parallel and increment the appropriate c_i. Finding all triangles containing the edge update and containing only high-degree vertices takes $O(\Delta\sqrt{m})$ work and $O(\log^*(\Delta+m))$ depth w.h.p. This is because there are $O(\sqrt{m})$ high-degree vertices in total, and for each we check whether it appears in the $\mathcal{HH}$ table for both endpoints of each update. Performing lookups in $\mathcal{T}$ takes $O(\Delta)$ work and $O(\log^*(\Delta+m))$ depth w.h.p.

For each update containing at least one endpoint with low-degree, we perform lookups in the tables $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ to find all triangles containing the update and increment the appropriate c_i. This takes $O(\Delta\sqrt{m}+\Delta)$ work and $O(\log^*(\Delta+m))$ depth w.h.p. Incrementing all c_i's for all newly updated triangles takes $O(\Delta)$ work and $O(1)$ depth. We then apply the equation in step **(6)** to update C, which takes $O(1)$ work and depth. ∎

The following lemma bounds the cost for minor rebalancing, where a low-degree vertex becomes high-degree or vice versa (step **(9)**).

LEMMA 3.3. *Minor rebalancing for edge updates takes $O(\Delta\sqrt{m})$ amortized work and $O(\log^*(\Delta+m))$ depth w.h.p., and $O(\Delta+m)$ space.*

Proof. We describe the case of edge insertions, and the case for edge deletions is similar. Using approximate compaction to perform the filtering, we first find the set S of low-degree vertices exceeding t_2 in degree. This step takes $O(\Delta)$ work and $O(\log^*\Delta)$ depth w.h.p. For vertices in S, we then delete the edges from their old hash tables and move the edges to their new hash tables. The work for each vertex is proportional to its current degree, giving a total work of $O(\sum_{v\in S}\deg(v)) = O(\Delta\sqrt{m}+\Delta)$ w.h.p. since the original degree of low-degree vertices is $O(\sqrt{m})$ and each edge in the batch could have caused at most 2 such vertices to have their degree increase by 1 (the w.h.p. is for parallel hash table operations).

In addition to moving the edges into new hash tables, we also have to update $\mathcal{T}$ with new pairs of vertices that became high-degree and delete pairs of vertices that are no longer both high-degree. To update these tables, we need to find all new pairs of high-degree vertices. There are at most $O(\Delta\sqrt{m+\Delta})$ such new pairs, which can be found by filtering neighbors using approximate compaction of vertices in S in $O(\Delta\sqrt{m+\Delta})$ work and $O(\log^*(\Delta+m))$ depth w.h.p. For each pair (u,v), we check all neighbors of an endpoint that just became high-degree and increment the entry $\mathcal{T}(u,v)$ for each low-degree neighbor w found that has edges (u,w) and (w,v). Low-degree neighbors have degree $O(\sqrt{m+\Delta})$, and so the total work is $O(\Delta(m+\Delta))$ and depth is $O(\log^*(\Delta+m))$ w.h.p. using atomic-add. There must have been $\Omega(\sqrt{m})$ updates on a vertex before minor rebalancing is triggered, and so the amortized work per update is $O(\Delta\sqrt{m})$ and the depth is $O(\log^* m)$ w.h.p. The space for filtering is $O(m+\Delta)$. ∎

We now finish showing Theorem 3.1. Lemma 3.2 shows that our algorithm maintains the correct count of triangles. Lemmas 3.1, 3.2, and 3.3 show that the cost of updating tables to reflect the batch, updating the triangle counts, and minor rebalancing is $O(\Delta\sqrt{m}+\Delta)$ amortized work and $O(\log^*(\Delta+m))$ depth w.h.p., and $O(\Delta+m)$ space.

Step **(7)** can be done in $O(\Delta\sqrt{m})$ work and $O(\log^* m)$ depth as follows. We scan through the batch $\mathcal{B}$ in parallel and update the hash tables $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$ in $O(\Delta)$ work and $O(\log^*(\Delta+m))$ depth w.h.p. For all updates in $\mathcal{B}$ containing one high-degree vertex and one low-degree vertex, we update the table $\mathcal{T}$ in parallel by scanning the neighbors in $\mathcal{LH}$ of the low-degree vertex. This step takes $O(\Delta\sqrt{m}+\Delta)$ work and $O(\log^*(\Delta+m))$ depth w.h.p. Major rebalancing (step **(10)**) takes $O((\Delta+m)^{3/2})$ work and $O(\log^*(\Delta+m))$ depth by re-initializing the data structures. The rebalancing happens every $\Omega(m)$ updates, and so the amortized work per update is $O(\sqrt{\Delta+m})$ and depth is $O(\log^*(\Delta+m))$ w.h.p.

Therefore, our update algorithm takes $O(\Delta\sqrt{\Delta+m})$ amortized work and $O(\log^*(\Delta+m))$ depth w.h.p., and $O(\Delta+m)$ space overall using atomic-add as stated in Theorem 3.1.

Bounds without Atomic-Add. Without the atomic-add instruction, we can use a parallel reduction [Jaj92] to sum over values when needed. This is work-efficient and takes logarithmic depth, but uses space proportional to the number of values summed over in parallel. For updates, this is bounded by $O(\Delta\sqrt{m}+\Delta)$, and for initialization and major rebalancing, this is bounded by $O(\alpha m)$ [ST15]. This would give an overall bound of $O(\Delta(\sqrt{\Delta+m}))$ work and $O(\log(\Delta+m))$ depth w.h.p., and $O(\alpha m+\Delta\sqrt{m})$ space.

4 Dynamic k-Clique Counting via Fast Static Parallel Algorithms

We present a very simple algorithm for dynamically maintaining the number of k-cliques based on statically enumerating smaller cliques in the graph, and intersecting the enumerated cliques with the edge updates in the input batch. The algorithm is space-efficient.

Our algorithm is based on a work-efficient parallel algorithm for counting k-cliques in $O(m\alpha^{k-2})$ expected work and $O(\log^{k-2} n)$ depth w.h.p. by Shi et al. [SDS20]. Using this algorithm, we show that updating the k-clique count for a batch of Δ updates can be done in $O(\Delta(m+\Delta)\alpha^{k-4})$ expected work, $O(\log^{k-2} n)$ depth w.h.p., and $O(m+\Delta)$ space. For $\Delta \geq m$ we simply call the static algorithm, and for $\Delta < m$ we use the static algorithm to (i) enumerate all $(k-2)$-cliques, and (ii) check whether each $(k-2)$-clique forms a k-clique with an edge in the batch. This procedure outperforms re-computation using the static parallel k-clique counting algorithm for $\Delta = o(\alpha^2)$. The full details of our algorithm can be found in the full version [DLSY20] of this paper.

5 Dynamic k-Clique via Fast Matrix Multiplication

In this section, we present our parallel batch-dynamic algorithm for counting k-cliques based on fast matrix multiplication in general graphs (which may be dense). Our algorithm is inspired by the static triangle counting algorithm of Alon, Yuster, and Zwick (AYZ) [AYZ97] and the static k-clique counting algorithm of Eisenbrand and Grandoni [EG04] that uses matrix multiplication-based triangle counting. We present a new dynamic algorithm that obtains better bounds than the simple algorithm based on static lower-clique enumeration in Section 4 for larger values of k.

We define the ***parallel matrix multiplication exponent*** to be the smallest exponent ω_p such that there exists a parallel matrix multiplication algorithm that multiplies two $n \times n$ matrices with $O\left(n^{\omega_p}\right)$ work and $O(\log n)$ depth, using $O\left(n^{\omega_p}\right)$ space. We show that $\omega_p = 2.373$ in the full version of the paper [DLSY20]. Assuming a parallel matrix multiplication exponent of ω_p, our algorithm handles batches of Δ edge insertions/deletions using $O\left(\min\left(\Delta m^{\frac{(2k-3)\omega_p}{3(1+\omega_p)}}, (m+\Delta)^{\frac{2k\omega_p}{3(1+\omega_p)}}\right)\right)$ work and $O(\log m)$ depth w.h.p., and $O\left((m+\Delta)^{\frac{2k\omega_p}{3(1+\omega_p)}}\right)$ space where m is the number of edges in the graph after applying the batch of updates. To the best of our knowledge, the sequential (batch-dynamic) version of our algorithm also provides the best bounds for dynamic k-clique counting in the sequential model for dense graphs for large constant values of k (assuming we use the best currently known matrix multiplication algorithm) [DT13].

More formally, we obtain the following results:

THEOREM 5.1. *Our fast matrix multiplication based k-clique algorithm takes $O\left(\min\left(\Delta m^{\frac{2(k-1)\omega_p}{3(\omega_p+1)}}, (\Delta+m)^{\frac{(2k+1)\omega_p}{3(\omega_p+1)}}\right)\right)$ work and $O(\log(m+\Delta))$ depth w.h.p., and $O\left((\Delta+m)^{\frac{(2k+1)\omega_p}{3(\omega_p+1)}}\right)$ space assuming a parallel matrix multiplication algorithm with coefficient ω_p when $k \bmod 3 = 1$, and $O\left(\min\left(\Delta m^{\frac{(2k-1)\omega_p}{3(\omega_p+1)}}, (\Delta+m)^{\frac{2(k+1)\omega_p}{3(\omega_p+1)}}\right)\right)$ work and $O(\log(m+\Delta))$ depth w.h.p., and $O\left((\Delta+m)^{\frac{2(k+1)\omega_p}{3(\omega_p+1)}}\right)$ space when $k \bmod 3 = 2$.*

COROLLARY 5.1. *Provided the best known parallel matrix multiplication exponent $\omega_p = 2.373$, we obtain a parallel fast matrix multiplication k-clique algorithm that takes $O\left(\min\left(\Delta m^{0.469k-0.469}, (\Delta+m)^{0.469k+0.235}\right)\right)$ work and $O(\log m)$ depth w.h.p., and $O\left((\Delta+m)^{0.469k+0.235}\right)$ space when $k \bmod 3 = 1$, and $O\left(\min\left(\Delta m^{0.469k-0.235}, (\Delta+m)^{0.469k+0.469}\right)\right)$ work and $O(\log m)$ depth w.h.p., and $O\left((\Delta+m)^{0.469k+0.469}\right)$ space when $k \bmod 3 = 2$.*

High-Level Approach and Techniques. For a given graph $G=(V,E)$, we create an auxiliary graph $G'=(V',E')$ with vertices and edges representing cliques of various sizes in G. For a given k-clique problem, vertices in V' represent cliques of size $k/3$ in G and edges (u,v) between vertices $u,v \in V'$ represent cliques of size $2k/3$ in G. Thus, a triangle in G' represents a k-clique in G. Specifically, there exist exactly $\binom{k}{k/3}\binom{2k/3}{k/3}$ different triangles in G' for each clique in G.

Given a batch of edge insertions and deletions to G, we create a set of edge insertions and deletions to G'. An edge is inserted in G' when a new $2k/3$-clique is created in G and an edge is deleted in G' when a $2k/3$-clique is destroyed in G. Suppose, for now, that we have a dynamic algorithm for processing the edge insertions/deletions into G'. Counting the number of triangles in G' after processing all edge insertions/deletions and dividing by $\binom{k}{k/3}\binom{2k/3}{k}$ provides us with the exact number of cliques in G.

There are several challenges that we must deal with when formulating our dynamic triangle counting algorithm

for counting the triangles in G':

(**1**) We cannot simply count all the triangles in G' after inserting/deleting the new edges as this does not perform better than a trivial static algorithm.

(**2**) Any trivial dynamization of the AYZ algorithm will not be able to detect all new triangles in G'. Specifically, because the AYZ algorithm counts all triangles containing a low-degree vertex separately from all triangles containing only high-degree vertices, if an edge update only occurs between high-degree vertices, a trivial dynamization of the algorithm will not be able to detect any triangle that the two high-degree endpoints make with low-degree vertices.

(**3**) We must ensure that batches of updates can be efficiently processed in parallel without overcounting.

To solve the first challenge, we dynamically count low-degree and high-degree vertices in different ways. Let $\ell = k/3$ and $M = 2m + 1$. For some value of $0 < t < 1$, we define ***low-degree*** vertices to be vertices that have degree less than $M^{t\ell}/2$ and ***high-degree*** vertices to have degree greater than $3M^{t\ell}/2$. Vertices with degrees in the range $[M^{t\ell}/2, 3M^{t\ell}/2]$ can be classified as either low-degree or high-degree. We analyze the specific value to use for t in the full version of our paper [DLSY20]. We perform rebalancing of the data structures as needed as they handle more updates. For low-degree vertices, we only count the triangles that include at least one newly inserted/deleted edge, at least one of whose endpoints is low-degree. This means that we do not need to count any pre-existing triangles that contain at least one low-degree vertex. For the high-degree vertices, because there is an upper bound on the maximum number of such vertices in the graph, we update an adjacency matrix A containing edges only between high-degree vertices. At the end of all of the edge updates, computing A^3 gives us a count of all of the triangles that contain three high-degree vertices.

This procedure immediately then leads to our second challenge. To solve this second challenge, we make the observation (stated in Lemma 5.1 below, and proven in the full version of our paper [DLSY20]) that if there exists an edge update between two high-degree vertices that creates or destroys a triangle that contains a low-degree vertex in G', then there *must* exist at least one new edge insertion/deletion *that creates or destroys a triangle representing the same clique* to that low-degree vertex in the same batch of updates to G'. Thus, we can use one of those edge insertions/deletions to determine the new clique that was created and, through this method, find all triangles containing at least one low-degree vertex and at least one new edge update. Some care must be observed in implementing this procedure in order to not increase the runtime or space usage; such details can be found in the full version of our paper [DLSY20].

LEMMA 5.1. *Given a graph $G = (V, E)$, the corresponding $G' = (V', E')$, and for $k > 3$, suppose an edge insertion (resp. deletion) between two high-degree vertices in G' creates a new triangle, (u_H, w_H, x_L), in G' which contains a low-degree vertex x_L. Let $R(y)$ denote the set of vertices in V represented by a vertex $y \in V'$. Then, there exists a new edge insertion (resp. deletion) in G' that is incident to x_L and creates a new triangle (u', w', x_L) such that $R(u') \cup R(w') = R(u_H) \cup R(w_H)$.*

Algorithm 3 Simplified parallel matrix multiplication k-clique counting algorithm.

```
1:  function COUNT-CLIQUES(B)
2:      Update graph G' with B by inserting new ℓ- and 2ℓ-cliques.
3:      Find the batch of insertions (B'_I) and batch of deletions (B'_D)
        into G'.
4:      Determine the final degrees of every vertex in G' after
        performing updates B'_I and B'_D.
5:      δ ← threshold for low-degree vs. high-degree.
6:      ▷ The precise value of δ is defined in the full version of our
        paper [DLSY20].
7:      parfor insert(u, v) ∈ B'_I, delete(u, v) ∈ B'_D do
8:          if either u or v is low-degree (degree ≤ δ) then
9:              Enumerate all triangles containing (u, v). Let this
                set be T.
10:             By Lemma 5.1, find all possible triangles
                representing the same triangle t ∈ T.
11:             Correct for duplicate counting of triangles.
12:         else
13:             Update A (adjacency matrix for high-degree
                vertices).
14:     Compute A^3. The diagonal provides the triangle counts for
        all triangles containing only high-degree vertices.
15:     Sum the counts of all triangles.
16:     Correct for duplicate counting of cliques.
```

Incorporating Batching and Parallelism. When dealing with a batch of updates containing both edge insertions and deletions, we must be careful when vertices switch from being high-degree to being low-degree and vice versa.

If we intersperse the edge insertions with the edge deletions, then there is the possibility that a vertex switches between low and high-degree multiple times in a single batch. Thus, we batch all edge deletions together and perform these updates first before handling the edge insertions. After processing the batch of edge deletions, we must subsequently move any high-degree vertices that become low-degree to their correct data structures. After dealing with the edge insertions, we must similarly move any low-degree vertices that become high-degree to the correct data structures. Finally, for triangles that contain more than one edge update, we must account for potential double counting by different updates happening in parallel. Such challenges are described and dealt with in detail in the full version of our paper [DLSY20]. A high-level description of the algorithm is given in Algorithm 3.

Graph Dataset	Num. Vertices	Num. Edges
Orkut	3,072,627	234,370,166
Twitter	41,652,231	2,405,026,092
rMAT	16,384	121,362,232

Table 1: Graph inputs, including number of vertices and edges.

m	unique edges	m	unique edges
2×10^6	1,569,454	4×10^8	55,395,676
2×10^7	9,689,644	8×10^8	74,698,492
1×10^8	27,089,362	3.2×10^9	121,362,232
2×10^8	39,510,764		

Table 2: Number of unique edges in the first m edges from the *rMAT* generator.

6 Experimental Results

Experimental Setup. Our experiments are performed on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4×2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use a work-stealing scheduler that we implemented [BAD20]. The scheduler is implemented similarly to Cilk for parallelism. Our programs are compiled using `g++` (version 7.3.0) with the `-O3` flag.

Graph Data. Table 1 lists the graphs that we use. ***com-Orkut*** is an undirected graph of the Orkut social network [LK14]. ***Twitter*** is a directed graph of the Twitter network [KLPM10]. We symmetrize the Twitter graph for our experiments. For some of our experiments which ingest a stream of edge updates, we sample edges from an rMAT generator [CZF04] with $a = 0.5, b = c = 0.1, d = 0.1$ to perform the updates. The update stream can have duplicate edges, and Table 2 reports the number of unique edges found in prefixes of various sizes of the rMAT stream that we generate. The unique edges in the full stream represents the rMAT graph described in Table 1.

6.1 Our Implementation

Parallel Primitives. We implemented a multicore CPU version of our algorithm using the Graph Based Benchmark Suite (GBBS) [DBS18b], which includes a number of useful parallel primitives, including high-performance parallel sorting, and primitives such as prefix sum, reduce, and filter [Jaj92]. In what follows, a ***filter*** takes an array A and a predicate function f, and returns a new array containing $a \in A$ for which $f(a)$ is true, in the same order that they appear in A. Our implementations use the atomic compare-and-swap and atomic-add instructions available on modern CPUs.

Implementation. For $\mathcal{T}$, we used the concurrent linear probing hash table by Shun and Blelloch [SB14]. For each of the data structures $\mathcal{HH}$, $\mathcal{HL}$, $\mathcal{LH}$, and $\mathcal{LL}$, we created an array of size n, storing (possibly null) pointers to hash tables [SB14]. For an edge (u, v) in one of the data structures, the value v will be stored in the hash table pointed to by the u'th slot in the array. We also tried using hash tables for both levels, but found it to be slower in practice. For deletions, we used the folklore *tombstone* method. In this method, when an element is deleted, we mark the slot in the table as a tombstone, which is a special value. When inserting, we can insert into a tombstone, but we have to first check until seeing an empty slot to make sure that we are not inserting a duplicate key. In the preprocessing phase of the algorithm, instead of using approximate compaction, we used filter. To find the last update for duplicate updates, we use a parallel sample sort [SBF+12] to sort the edges first by both endpoints, and then by timestamp. Then we use filter to remove duplicate updates. When we initialize the dynamic data structures, a vertex is considered high-degree if it has degree greater than $2t_1$ and low-degree otherwise.

During minor rebalancing, a vertex only changes its status if its degree drops below t_1 or increases above t_2 due to the batch update. In major rebalancing, we merge our dynamic data structure and the updated edges into a compressed sparse row (CSR) format graph and use the static parallel triangle counting algorithm by Shun and Tangwongsan [ST15] to recompute the triangle count. We then build a new dynamic data structure from the CSR graph. We also implement several natural optimizations which improve performance. To reduce the overhead of using hash tables, we use an array to store the neighbors of vertices with degree less than a certain threshold (we used 128 in our experiments). Moreover, we only keep a single entry for (u, v) and (v, u) in the wedges table $\mathcal{T}$.

Experiments. Table 3 report the parallel running times on varying insertion and deletion batch sizes for our implementation of our new parallel batch-dynamic triangle counting algorithm designed. For the two graphs based on static graph inputs (Orkut and Twitter), we generate updates for the algorithm by representing the edges of the graph as an array, and randomly permuting them. The algorithm is then run using batches of the specified size. For insertions, we start with an empty graph and apply batches from the beginning to the end of the permuted array. For deletions, we start with the full graph and apply batches from the end to the beginning of the permuted array. The table also reports the running time for the GBBS implementation of the state-of-the-art static triangle counting algorithm of Shun and Tangwongsan [ST15, DBS18b].

Across varying batch sizes, our algorithm achieves throughputs between 1.05–16.2 million edges per second for the Orkut graph, 0.935–5.46 million edges per second for the Twitter graph, and 3.08–32.4 million edges per second for the rMAT graph. We obtain much higher throughput for the rMAT graph due to the large number of duplicate edges found in this graph stream, as illustrated in Table 2. We observe that in all cases, the average time for processing a batch is smaller than the running time of the static algorithm. The maximum speedup of our algorithm over the static algorithm is $22709\times$ for the rMAT graph with a deletion batch of size 2×10^3, but

Algorithm	Graph	Batch Size 2×10^3	2×10^4	2×10^5	2×10^6	m
Ours (INS)	Orkut	1.90e-3	4.76e-3	0.0235	0.168	–
	Twitter	**2.11e-3**	**7.10e-3**	**0.0430**	**0.366**	–
	rMAT	**6.42e-4**	**2.09e-3**	**8.62e-3**	0.0618	–
Makkar et al. (INS) [MBG17]	Orkut	**9.76e-4**	**2.69e-3**	**0.0143**	**0.0830**	–
	Twitter	time-out	0.0644	0.437	3.88	–
	rMAT	1.98e-3	6.90e-3	0.012	**0.0335**	–
Ours (DEL)	Orkut	1.80e-3	4.37e-3	0.0189	0.124	–
	Twitter	**2.14e-3**	**7.76e-3**	**0.0486**	**0.385**	–
	rMAT	6.48e-4	2.23e-3	9.21e-3	0.0723	–
Makkar et al. (DEL) [MBG17]	Orkut	**4.63e-4**	**1.46e-3**	**8.12e-3**	**0.0499**	–
	Twitter	time-out	0.0597	0.401	3.64	–
	rMAT	**4.47e-4**	**1.81e-3**	**5.12e-3**	**0.027**	–
Static [ST15]	Orkut	–	–	–	–	1.027
	Twitter	–	–	–	–	32.1
	rMAT	–	–	–	–	14.7

Table 3: Running times (seconds) for our parallel batch-dynamic triangle counting algorithm and Makkar et al. [MBG17]'s algorithm on 72 cores with hyper-threading. We apply the edges in each graph as batches of edge insertions (INS) or deletions (DEL) of varying sizes, ranging from 2×10^3 to 2×10^6, and report the average time for each batch size. The update time of Makkar et al. algorithm for Twitter batch size 2×10^3 is missing because the expriment timed out. We also report the update time for the state-of-the-art static triangle counting algorithm of Shun and Tangwongsan [ST15], which processes a single batch of size m. Note that for the Twitter and Orkut datasets, all of the edges are unique. However, for the rMAT dataset, batches can have duplicate edges. For each batch size of each dataset, we list the fastest time in bold.

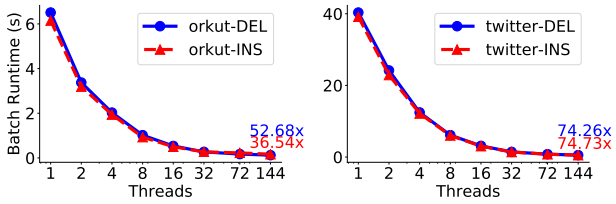

Figure 1: Running times of our parallel batch-dynamic triangle counting algorithm with respect to thread count (the x-axis is in log-scale) on the Orkut (average time across all batches) and Twitter (running time for the 6th batch) graph for both insertion (red dashed line) and deletion (blue solid line). "144" indicates 72 cores with hyper-threading. The experiment is run with a batch size of 2×10^6. The parallel speedup on 144 threads over a single thread is displayed.

in general our algorithm achieves good speedups across the entire range of batches that we evaluate.

Lastly, Figure 1 shows the parallel speedup of our algorithm with varying thread-count on the Orkut and Twitter graph, for a fixed batch size of 2×10^6. Our algorithm achieves a maximum of $74.73\times$ speedup using 72 cores with hyper-threading for this experiment.

6.2 Comparison with Existing Algorithms

Comparison with Ediger et al. We compared our implementation with a shared-memory implementation of the Ediger et al. algorithm [EJRB10], which is implemented as part of the STINGER dynamic graph processing system [EMRB12]. Unfortunately, we found that their implementation is much slower than ours due to bottlenecks in the update time for the underlying dynamic graph data structure. We note that recent work on streaming graph processing observed similar results for using STINGER [DBS19]. To obtain a fair comparison, we chose to focus on implementing a more recent GPU batch-dynamic triangle counting algorithm ourselves, which we discuss next.

Comparison with Makkar et al. The Makkar et al. algorithm [MBG17] is a state-of-the-art parallel batch-dynamic triangle counting implementation designed for GPUs. To the best of our knowledge, there is no multicore implementation of this algorithm, and so in this paper we implement an optimized multicore version of their algorithm. The algorithm works as follows. First, their algorithm separates the batch of updates into batches for insertions and deletions. Then, for each batch of updates, it creates an *update graph*, $\hat{G}$, for each batch consisting of only the updates within each batch. Then, it merges the updates from each batch with the original edges in the graph to create an updated graph for each of the batches, G'. Note that this graph contains both the edges previously in the graph, as well as the new edges.

The merging process to construct G' first sorts the batch to obtain sorted lists of neighbors to add/delete from the adjacency lists of vertices in the graph. Then, the algorithm performs a simple linear-work procedure to merge each existing adjacency list with the sorted updates. In particular, doing t edge updates on a vertex with degree d takes $O(d + t)$ work. Finally, the algorithm counts the triangles by intersecting the adjacency lists of the endpoints of each edge in the batch. For each edge (u, v), they intersect $G'(u)$ with $G'(v)$, $G'(u)$ with $\hat{G}(v)$, and $\hat{G}(u)$ with $\hat{G}(v)$. The count of the number of triangles can be obtained from the number of intersections obtained from each of these cases using a simple inclusion-exclusion formula. They provide a further optimization by only intersecting *truncated* adjacency lists in some of the cases where a truncated adjacency list is one where the list only contains vertices with IDs less than the ID of the vertex that the adjacency list belongs to. Their algorithm has a worst case work bound of $O(n^2)$.

Implementation. We developed a new multicore implementation of the Makkar et al. algorithm using the same parallel primitives and framework described earlier for the implementation of our algorithm. We implemented several optimizations that improved performance. First, we handle vertices with degree lower than 16 by storing their incident edges in a special array of size $16n$, and only allocate memory for vertices with larger degree. Second, we note that their algorithm does not specify how to handle redundant insertions that are already present in the graph. We remove these edge updates by modifying the merge algorithm that constructs G' from G. Specifically, during the merge, if we identify that a given edge is already present in G, we mark it in the sorted sequence

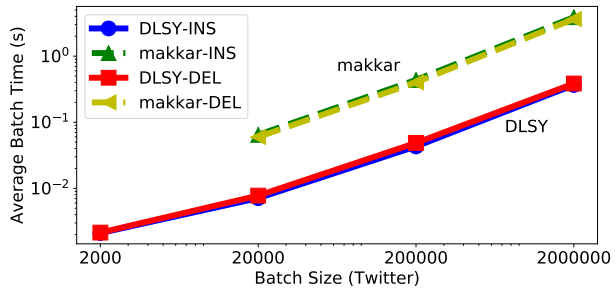

Figure 2: This figure plots the average insertion and deletion round times for each batch size (log-log scale) on Twitter using 72 cores with hyper-threading. The plot is in log-log scale. The lines for our algorithm are solid (blue for insertion and red for deletion) while the lines for Makkar et al. algorithm are dashed (green for insertion and yellow for deletion). The update time of Makkar et al. algorithm for Twitter batch size 2×10^3 is missing because the experiment timed out.

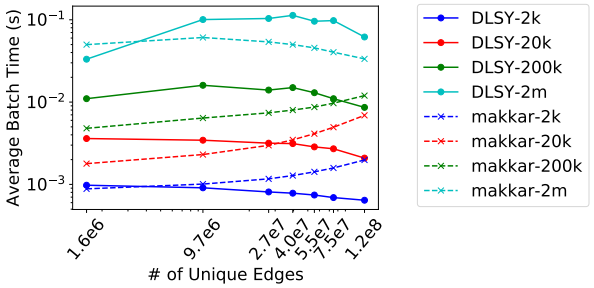

Figure 3: Comparison of the performance of our implementation (DLSY, solid line) and Makkar et al. algorithm [MBG17] (makkar, dotted line) for batches of insertions. The figure shows the average batch time for different batch sizes on the rMAT graph with varying prefixes of the generated edge stream to control density. The number of unique edges in the prefix is shown on the x-axis. The number of vertices is fixed at 16,384. The dark blue, red, green, and light blue lines are for batches of size 2×10^3, 2×10^4, 2×10^5, and 2×10^6, respectively. We see that our new algorithm is faster for small batches and on denser graphs.

of batch updates that we are merging in. Removing these marked updates to construct $\hat{G}$ without redundant updates is done by using a parallel filter.

Performance Comparison. Table 3 shows the running times of the Makkar et al. algorithm on batches of insertions and deletions of different sizes. The data points for the Twitter graph are also plotted in Figure 2. We observe that the Makkar et al. algorithm is faster than our algorithm on the Orkut graph, especially for large batches. On the other hand, for the Twitter graph, our algorithm is consistently faster for both insertions and deletions across all batch sizes. This is because there are no vertices with very high degree in the Orkut graph, and so the Makkar et al. algorithm does less work in merging adjacency lists with updates, while the Twitter graph has vertices with extremely high degree, which are costly to merge. Both algorithms are significantly faster than simply applying the static triangle counting algorithm for the range of batch sizes that we considered.

Next, we evaluate the performance of insertion batches in our algorithm and the Makkar et al. algorithm on the synthetic rMAT graph with 3.2 billion generated edges (which have duplicates). This synthetic experiment allows us to study how both algorithms perform as the graph becomes more dense. We evaluate the performance for different insertion batch sizes. The experiment uses prefixes of the rMAT graph (the number of unique edges per prefix is shown in Table 2) to control the density of the graph. The vertex set in this experiment is fixed, and thus a larger number of unique edges corresponds to a denser graph.

Figure 3 plots the running time of both implementations for varying batch sizes as a function of the graph density. We observe that for small batch sizes, the performance of the Makkar et al. algorithm degrades significantly as the graph grows more dense and contains more high-degree vertices. On the other hand, our algorithm's performance generally does not degrade as the graph grows denser, across all batch sizes. We also significantly outperform the Makkar et al. algorithm for small batch sizes. Specifically, we obtain a maximum speedup of $3.31\times$ for a batch of size 2×10^4. This is because the overhead of updating of high-degree vertices in the Makkar et al. algorithm becomes relatively higher, as work proportional to the vertex degree must be done regardless of the number of new incident edges.

7 Conclusion

In this paper, we have given new dynamic algorithms for the k-clique problem. We study this fundamental problem in the batch-dynamic setting, which is better suited for parallel hardware that is widely available today, and enables dynamic algorithms to scale to high-rate data streams. We have presented a work-efficient parallel batch-dynamic triangle counting algorithm. We also gave a simple, enumeration-based algorithm for maintaining the k-clique count. In addition, we have presented a novel parallel batch-dynamic k-clique counting algorithm based on fast matrix multiplication, which is asymptotically faster than existing dynamic approaches on dense graphs. Finally, we provide a multicore implementation of our parallel batch-dynamic triangle counting algorithm and compare it with state-of-the-art implementations that have weaker theoretical guarantees, showing that our algorithm is competitive in practice.

Acknowledgements. We thank Josh Alman, Nicole Wein, and Virginia Vassilevska Williams for helpful discussions on various aspects of our paper. We also thank anonymous reviewers for their helpful suggestions. This research was supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a

JUMP Center co-sponsored by SRC and DARPA.

References

[AABD19] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. Parallel batch-dynamic graph connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 381–392, 2019.

[AAW17] Umut A. Acar, Vitaly Aksenov, and Sam Westrick. Brief announcement: Parallel dynamic tree contraction via self-adjusting computation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 275–277, 2017.

[ALT+17] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[AMSJ18] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.*, 11(6):691–704, February 2018.

[AYZ97] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, Mar 1997.

[BAD20] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Brief announcement: ParlayLib – a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 507–509, 2020.

[CN85] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, February 1985.

[CZF04] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SIAM International Conference on Data Mining (SDM)*, pages 442–446, 2004.

[DBS18a] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. In *International Conference on World Wide Web (WWW)*, pages 589–598, 2018.

[DBS18b] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 393–404, 2018.

[DBS19] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 918–934, 2019.

[DDK+20] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1300–1319, 2020.

[DF94] Sajal K. Das and Paolo Ferragina. An $o(n)$ work EREW parallel algorithm for updating MST. In *Annual European Symposium on Algorithms (ESA)*, pages 331–342, 1994.

[DF95] Rod G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

[DLSY20] Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. *CoRR*, abs/2003.13585, 2020.

[DT13] Zdeněk Dvořák and Vojtěch Tůma. A dynamic data structure for counting subgraphs in sparse graphs. In *Algorithms and Data Structures*, pages 304–315, 2013.

[EG04] Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.*, 326(1-3):57–67, October 2004.

[EGST12] David Eppstein, Michael T. Goodrich, Darren Strash, and Lowell Trott. Extended dynamic subgraph statistics using h-index parameterized data structures. *Theoretical Computer Science*, 447:44 – 52, 2012.

[EJRB10] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader. Massive streaming data analytics: A case study with clustering coefficients. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pages 1–8, 2010.

[EMRB12] David Ediger, Robert McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing (HPEC)*, pages 1–5, 2012.

[ES09] David Eppstein and Emma S. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. In *Algorithms and Data Structures (WADS)*, pages 278–289, 2009.

[FFF15] Irene Finocchi, Marco Finocchi, and Emanuele G. Fusco. Clique counting in MapReduce: Algorithms and experiments. *J. Exp. Algorithmics*, 20:1.7:1–1.7:20, October 2015.

[FL94] Paolo Ferragina and Fabrizio Luccio. Batch dynamic algorithms for two graph problems. In *Parallel Architectures and Languages Europe (PARLE)*, pages 713–724, 1994.

[GMV91] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 698–710, 1991.

[Gra77] Mark S Granovetter. The strength of weak ties. In *Social Networks*, pages 347–367. Elsevier, 1977.

[HR05] Robert A. Hanneman and Mark Riddle. *Introduction to Social Network Methods*. University of California, Riverside, 2005.

[ILMP19] Giuseppe F. Italiano, Silvio Lattanzi, Vahab S. Mirrokni, and Nikos Parotsidis. Dynamic algorithms for the massively parallel computation model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 49–58, 2019.

[Jaj92] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[JS17] Shweta Jain and C. Seshadhri. A fast and provable method for estimating clique counts using Turán's theorem. In *International Conference on World Wide Web (WWW)*, pages 441–449, 2017.

[Kha17] Shahbaz Khan. Near optimal parallel algorithms for dynamic DFS in undirected graphs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 283–292, 2017.

[KLPM10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media?

pages 591–600, 2010.

[KNN+19] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates in worst-case optimal time. In *International Conference on Database Theory (ICDT)*, volume 127, pages 4:1–4:18, 2019.

[KPR18] Tsvi Kopelowitz, Ely Porat, and Yair Rosenmutter. Improved worst-case deterministic parallel dynamic minimum spanning forest. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 333–341, 2018.

[Lat08] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, 2008.

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 296–303, 2014.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. 2014.

[MBG17] D. Makkar, D. A. Bader, and O. Green. Exact and parallel triangle counting in dynamic graphs. In *IEEE International Conference on High Performance Computing (HiPC)*, pages 2–12, Dec 2017.

[New03] Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[NP85] Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 026(2):415–419, 1985.

[NPRR18] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, March 2018.

[RT94] John H. Reif and Stephen R. Tate. Dynamic parallel tree contraction (extended abstract). In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 114–121, 1994.

[SB14] Julian Shun and Guy E Blelloch. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.

[SBF+12] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012.

[SDS20] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. *CoRR*, abs/2002.10047, 2020.

[ST15] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *IEEE International Conference on Data Engineering (ICDE)*, pages 149–160, 2015.

[STTW18] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel union-find. *Concurrency and Computation: Practice and Experience*, 30(4), 2018.

[TDB19] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. Batch-parallel Euler tour trees. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 92–106, 2019.

[Vas09] Virginia Vassilevska. Efficient algorithms for clique problems. *Information Processing Letters*, 109(4):254–257, 2009.

[Vis10] Uzi Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. 2010.

[Wil12] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *ACM Symposium on Theory of Computing Conference (STOC)*, pages 887–898, 2012.

[WS98] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440, 1998.