Parallel Five-cycle Counting Algorithms

JESSICA SHI, LOUISA RUIXUE HUANG, and JULIAN SHUN, MIT CSAIL, USA

Counting the frequency of subgraphs in large networks is a classic research question that reveals the underlying substructures of these networks for important applications. However, subgraph counting is a challenging problem, even for subgraph sizes as small as five, due to the combinatorial explosion in the number of possible occurrences. This article focuses on the five-cycle, which is an important special case of five-vertex subgraph counting and one of the most difficult to count efficiently.

We design two new parallel five-cycle counting algorithms and prove that they are work efficient and achieve polylogarithmic span. Both algorithms are based on computing low out-degree orientations, which enables the efficient computation of directed two-paths and three-paths, and the algorithms differ in the ways in which they use this orientation to eliminate double-counting. Additionally, we present new parallel algorithms for obtaining unbiased estimates of five-cycle counts using graph sparsification. We develop fast multicore implementations of the algorithms and propose a work scheduling optimization to improve their performance. Our experiments on a variety of real-world graphs using a 36-core machine with two-way hyper-threading show that our best exact parallel algorithm achieves 10–46× self-relative speedup, outperforms our serial benchmarks by 10–32×, and outperforms the previous state-of-the-art serial algorithm by up to 818×. Our best approximate algorithm, for a reasonable probability parameter, achieves up to 20× self-relative speedup and is able to approximate five-cycle counts 9–189× faster than our best exact algorithm, with between 0.52% and 11.77% error.

CCS Concepts: • **Theory of computation → Shared memory algorithms**; **Graph algorithms analysis**; • **Computing methodologies → Shared memory algorithms;**

Additional Key Words and Phrases: Cycle counting, parallel algorithms, graph algorithms

ACM Reference format:
Jessica Shi, Louisa Ruixue Huang, and Julian Shun. 2022. Parallel Five-cycle Counting Algorithms. *J. Exp. Algorithmics* 27, 4, Article 4.1 (October 2022), 23 pages.
https://doi.org/10.1145/3556541

1 INTRODUCTION

Subgraph or graphlet counting is a long standing research topic in graph processing with rich applications in bioinformatics, social network analysis, and network model evaluation [13, 20, 21, 25]. While there has been significant recent work on counting subgraphs of size three or four [2, 18, 19], counting subgraphs of size five or more is a difficult task even on the most modern

This research was supported by NSF Graduate Research Fellowship #1122374, DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, Google Research Scholar Award, CSAIL CAP Initiative, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.
Authors address: J. Shi, 32 Vassar Street, 32-G728, Cambridge, MA 02139; email: jeshi@mit.edu; L. R. Huang, 32 Vassar Street, 32-G728, Cambridge, MA 02139; email: louisah8191@gmail.com; J. Shun, 32 Vassar Street, 32-G736, Cambridge, MA 02139; email: jshun@mit.edu.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2022 Copyright held by the owner/author(s).
1084-6654/2022/10-ART4.1
https://doi.org/10.1145/3556541

hardware due to the massive number of such subgraphs in large graphs. As the subgraph sizes grow, the number of possible subgraphs grows exponentially.

We consider specifically the efficient counting of five-cycles. This pattern is particularly important for fraud detection [37]. Compared to other connected five-vertex patterns, five-cycles are much more difficult to count, because they are the only such pattern that requires first counting all directed three-paths. Notably, the **Efficient Subgraph Counting Algorithmic PackagE (ESCAPE)**, a software package by Pinar et al. that serially counts all five-vertex subgraphs in large graphs [36], spends between 25% and 58% of the total runtime on counting five-cycles alone based on our measurement.

While there has been prior work on developing and implementing serial five-cycle counting algorithms [24, 28, 36], there has been no prior work on designing and implementing theoretically efficient and scalable parallel five-cycle counting algorithms. We focus on designing multicore solutions, as nearly all publicly available graphs (which have up to hundreds of billions of edges [33]) can fit on a commodity multicore machine [16, 17].

We present two new parallel five-cycle counting algorithms that not only have strong theoretical guarantees but are also demonstrably fast in practice. These algorithms are based on two different serial algorithms, namely by Kowalik [28] and from ESCAPE by Pinar et al. [36]. Kowalik studied k-cycle counting in graphs for $k \leq 6$ and proposed a five-cycle counting algorithm that runs in $O(md^2) = O(m\alpha^2)$ time for d-degenerate graphs [28], where m is the number of edges in the graph and α is the arboricity of the graph.[1] The ESCAPE implementation contains a five-cycle counting algorithm that, with an important modification that we make, achieves the same asymptotic complexity of $O(m\alpha^2)$ [36]. The arboricity of a graph is a measure of its sparsity, and having running times parameterized by α is desirable, since most real-world graphs have low arboricity [16].

The main procedure in both algorithms and the essential modification to the ESCAPE algorithm is to first compute an appropriate arboricity orientation of the graph in parallel, where the vertices' out-degrees are upper bounded by $O(\alpha)$. This orientation then enables the efficient counting of directed two-paths and three-paths, which are then appropriately aggregated to form five-cycles. Notably, the counting and aggregation steps can each be efficiently parallelized. The two algorithms differ fundamentally in the ways in which they use the orientations of these path substructures to eliminate double-counting. We prove theoretical bounds that show that both of our algorithms match the work of the best sequential algorithms, taking $O(m\alpha^2)$ work and $O(\log^2 n)$ span with high probability (w.h.p.).[2] Additionally, we present two approximate five-cycle counting algorithms based on counting five-cycles in a sparsified graph, and we prove that both approximation algorithms give unbiased estimates on the global five-cycle count. We show that both algorithms take $O(pm\alpha^2+m)$ expected work and $O(\log^2 n)$ span w.h.p. for a sampling probability p.

We present optimized implementations of our algorithms, which use thread-local data structures, fast resetting of arrays, and a new work scheduling strategy to improve load balancing. We provide a comprehensive experimental evaluation of our five-cycle counting algorithms. On a 36-core machine with two-way hyperthreading, our best exact parallel algorithm achieves between 10× and 46× self-relative speedup and between 162× and 818× speedups over the fastest prior serial five-cycle counting implementation, which is from ESCAPE [36]. We also implement our own serial versions of the two exact algorithms, which are 7–38.91× faster than ESCAPE's algorithm due to improved theoretical work complexities. Our best parallel algorithm achieves between 10× and 32× speedups over our best serial algorithm. Moreover, we show the tradeoffs

[1]A graph is d-degenerate if every subgraph has a vertex of degree at most d, and a graph has arboricity α if the minimum number of spanning forests needed to cover all of the edges of the graph is α.
[2]With high probability (w.h.p.) means that the probability is at least $1-1/n^c$ for some constant $c > 0$ for an input of size n.

between error and running time of our approximate five-cycle counting algorithms. In particular, we are able to approximate five-cycle counts with 11.77% error with a 9.10–188.94× speedup over exact five-cycle counting on the same graphs. Our parallel five-cycle counting code is available at https://github.com/ParAlg/gbbs/tree/master/benchmarks/CycleCounting.

This is an extended version of our article appearing in the *International Symposium on Experimental Algorithms* [26]. This extended version includes new contributions regarding parallel approximate five-cycle counting. Specifically, we introduce two approximate five-cycle counting algorithms based on graph sparsification, and we prove that both algorithms give unbiased estimates on the global five-cycle count. We also prove bounds on the variance, obtain a (ε, δ)-estimator of the global five-cycle count, and provide a comprehensive evaluation of our implementations, demonstrating significant speedups over exact five-cycle counting while maintaining good accuracy.

2 BACKGROUND AND RELATED WORK

The difficulty of cycle counting has attracted considerable research effort over the years. Counting the number of k-cycles with k as an input parameter is NP-complete, since it includes the problem of finding a Hamiltonian cycle. However, efficient algorithms have been developed to count k-cycles for $k \leq 5$. Notably, Alon et al. [3] developed algorithms for efficiently finding a k-cycle for general k, but these translate to efficient k-cycle counting algorithms only for planar graphs where $k \leq 5$. For $k = 3, 4$, Chiba and Nishizeki [14] proposed algorithms that take $O(m\alpha)$ time. More recently, Bera et al. [6] analyzed the subgraph counting problem for $k = 5$ and gave an algorithm that takes $O(m\alpha^3)$ time, and the five-cycle counting part of the algorithm takes $O(m\alpha^3)$ time. However, it is shown in the same study that this result is unlikely to be extended to $k > 5$, due to the Triangle Detection Conjecture, which puts a lower bound of $\Omega(m^{1+\gamma})$ time with $\gamma > 0$ on any triangle detection algorithm on an input graph with m edges [1]. If the conjecture holds, then a reduction of the triangle detection problem to the six-cycle counting problem implies that there cannot be a $o(f(\alpha)m^{1+\gamma})$ time algorithm for six-cycle counting.

Until recently, because of the high computational power required, exact five-vertex subgraph counting was often deemed impractical on graphs with more than a few million edges. Most effort has focused on obtaining approximate counts or approximate graphlet frequency distributions [9, 38, 48]. Hocevar and Demsar [24] developed Orca to count subgraphs of up to size five and tested them on graphs with tens of thousands of vertices. Pinar et al. [36] developed ESCAPE, which is the first package that aims to perform exact counting of all five-vertex subgraphs on moderately large graphs. However, ESCAPE does not exploit parallelism and is not optimized for cycle-counting. Kowalik [28] gave a serial algorithm for five-cycle counting that takes $O(m\alpha^2)$, the best known theoretical bound for five-cycle counting, but does not provide an implementation. In Section 4, we describe Kowalik's and Pinar et al.'s five-cycle counting algorithms in more detail.

While there has not been prior work on parallel five-cycle counting algorithms, parallel cycle counting algorithms for smaller cycles have been studied over the years. Specifically, for the case of three-cycles, or triangles, there has been a significant amount of attention over the past two decades (e.g., References [7, 35, 42, 45], among many others).

Moreover, fast sequential algorithms for four-cycles have been studied extensively. For bipartite graphs, four-cycles, also known as butterflies, are the smallest non-trivial subgraphs. Chiba and Nishizeki's work [14] described a four-cycle counting algorithm that takes $O(m\alpha)$ work by using a degree ordering of the graph. Subsequently, butterfly counting algorithms using degree ordering and other orderings have also been designed [40, 41, 46, 49, 50].

There have been fewer studies on parallel four-cycle counting algorithms. The Parametrized Graphlet Decomposition package by Ahmed et al. [2] provides efficient parallel implementations

of exact counting of subgraphs of up to size four, including four-cycles. Wang et al. [46] implement a distributed algorithm using MPI that partitions the vertices across processors, where each processor sequentially counts the number of butterflies for vertices in its partition. Shi and Shun [44] presented a framework for parallel butterfly counting with several algorithms achieving $O(m\alpha)$ expected work and $O(\log m)$ span with high probability. Wang et al. [47] describe a similar parallel butterfly counting algorithm, with an additional cache optimization in their implementation.

3 PRELIMINARIES

Graph Notation. The input to our algorithms is a simple, undirected, unweighted graph $G(V, E)$. The number of vertices is $|V| = n$ and the number of edges is $|E| = m$. Vertices are labeled $0, 1, \ldots, n-1$. In our analysis, we assume that $m = \Omega(n)$. For a vertex v, we use $N(v)$ to denote the neighbors of v and $\deg(v)$ to denote the degree of v. When discussing directed graphs, $N^{\rightarrow}(v)$ denotes v's out-neighbors and $N^{\leftarrow}(v)$ denote the in-neighbors.

Furthermore, we use $N_v(u)$ ($N_v^{\rightarrow}(u)$ for directed graphs) to represent the neighbors of vertex u that are after v given a non-increasing degree ordering. When vertices are relabeled by non-increasing degree order, we can easily obtain $N_v(u) = N(u) \cap \{w \in V \mid w > v\}$.

The *arboricity* of a graph G, denoted $\alpha(G)$, is defined as the minimum number of spanning forests needed to cover the graph. $\alpha(G)$ is known to be upper bounded by $O(\sqrt{m})$. Due to the fact that graphs modeling the real world tend to be sparse, $\alpha(G)$ tends to be small for graphs that we are interested in processing. It is also known that $\sum_{(u,v)\in E} \min(\deg(u), \deg(v)) = O(m\alpha(G))$ [14]. Closely related to arboricity is the *degeneracy* of a graph G, $d(G)$, or the smallest k such that every subgraph of G contains a vertex of degree at most k. It is known that $d(G) = \Theta(\alpha(G))$ [34]. As such, our asymptotic bounds can use $\alpha(G)$ or $d(G)$ interchangeably. When it is unambiguous, we write the arboricity and degeneracy of a graph as α and d, respectively.

Graph Format. For the theoretical analysis, we assume the graphs are stored in hash tables in the adjacency list format to obtain constant time edge queries. In our implementations, graphs are stored in **Compressed Sparse Row (CSR)** format, which is more compact and has better cache locality.

The Work-Span Model. To analyze the complexity of our parallel algorithms, we use the work-span model [27] with arbitrary forking. The work W of an algorithm is the total number of operations, and the *span* S of an algorithm is the longest dependency path. The work of a sequential algorithm is the same as its time. A *work-efficient* parallel algorithm has a work complexity matching the time of the best sequential algorithm for the problem. For an algorithm with work W and span S, the running time on P processors is upper bounded by $W/P + S$ [12].

Parallel Primitives. We use the following parallel primitives. A **parallel for-loop (parfor)** with n iterations that can be executed in parallel launches all of its iterations in $O(n)$ work and $O(1)$ span, assuming no dependencies between iterations. *Prefix sum* takes as input a sequence A of size n, an identity ε, and an associative binary operator $\oplus$, and returns a sequence B where $B[i] = \bigoplus_{j<i} A[i] \oplus \varepsilon$ for $0 \le i < n$. *Filter* takes as input a sequence A of size n and a predicate function f, and returns a sequence B of all $a \in A$ such that $f(a)$ is true, preserving the order of elements in A. Prefix sum and filter can be implemented to take $O(n)$ work and $O(\log n)$ span [27]. *Parallel integer sort* sorts n integers in the range $[0, O(n)]$ in $O(n)$ work and $O(\log n)$ span w.h.p. [39]. We also use *parallel hash tables*, which support a batch of n instructions in $O(n)$ work and $O(\log^* n)$ span w.h.p. [22]. We assume atomic adds take $O(1)$ work and span.

Estimators. We define a (ε, δ)-estimator of a number X, to be a random variable Y, where $Pr[|Y - X| \ge \varepsilon X] \le \delta$ for any $\varepsilon, \delta \in [0, 1]$.

4 FIVE-CYCLE COUNTING ALGORITHMS

In this section, we present two new parallel algorithms for counting five-cycles. The first algorithm is based on the serial algorithm by Kowalik [28]. Kowalik shows that the algorithm achieves a time complexity of $O(m)$ on planar graphs, in which $\alpha = O(1)$, or $O(md^2) = O(m\alpha^2)$ on d-degenerate graphs. The second algorithm is based on the serial algorithm by Pinar et al. in their ESCAPE framework for counting all 5-vertex subgraphs in a graph [36]. We show that both of the parallel algorithms that we design are provably work efficient with polylogarithmic span.

Additionally, we present two techniques that, combined with our parallel exact five-cycle counting algorithms, allow us to generate unbiased estimates of the global five-cycle count in parallel using graph sparsification.

4.1 Preprocessing: Graph Orientation

Similarly to many previous subgraph counting algorithms [6, 43], a key step in our algorithms is a preprocessing step that orients the graph G, creating a directed acyclic graph $G^{\rightarrow}$ where the out-degrees of vertices are upper bounded. We use *l-orientation* to refer to an orientation where each vertex's out-degree is bounded by l. Furthermore, orientations in our context are always induced from a total ordering of the vertices, where directed edges point from vertices lower in the ordering to vertices higher in the ordering. As such, the problem of orienting the graph is reduced to the problem of finding an appropriate ordering of the vertices.

Degree Orientation. The core idea of orienting an undirected input graph based on ordering the vertices by non-increasing degree to perform subgraph counting or listing is attributed to Chiba and Nishizeki [14]. Using degree ordering, they proposed efficient triangle and four-cycle counting algorithms based on this key result:

LEMMA 4.1 ([14]). *For a graph $G = (V, E)$, $\sum_{(u,v)\in E} \min\{deg(u), deg(v)\} \leq 2\alpha m$.*

This result allows us to bound the number of wedges in graph G by $2m\alpha$, where a wedge is defined as a triple (v, w, u) where $(v, w), (w, u) \in E$, $\deg(v) \geq \deg(w)$ and $\deg(v) \geq \deg(u)$. In Kowalik's five-cycle algorithm, wedges are the building blocks of five-cycles, and we can show that $O(\alpha)$ work is done for each wedge. Combined with the $O(m\alpha)$ bound on the number of wedges, this gives us the $O(m\alpha^2)$ running time bound.

Arboricity Orientation. An *arboricity orientation* of a graph is one where the vertices' out-degrees are upper bounded by $O(\alpha(G))$. An arboricity-oriented graph has slightly different theoretical properties compared to a degree-oriented graph, but literature has shown that in some algorithms arboricity orientation can achieve the same practical efficiency as degree orientation [43]. We note that in Kowalik's five-cycle counting algorithm as well as our parallelization of the algorithm, both a degree ordering and an arboricity ordering are required to achieve work efficiency.

One way to obtain an arboricity orientation is by computing the degeneracy ordering using a standard k-core decomposition algorithm [31, 32]. The algorithm repeatedly removes the vertex with the lowest degree from the graph. When we direct edges using this orientation, we obtain a DAG where each vertex's out-degree is bounded by $d(G)$. While this algorithm can be parallelized to be work efficient, it does not attain polylogarithmic span; notably, the problem is P-complete [4].

Since the parallel algorithm for exact degeneracy ordering has sub-optimal span, we use approximate algorithms with polylogarithmic span. We test two such algorithms: Goodrich–Pszona and Barenboim–Elkin. Both algorithms work by peeling low-degree vertices in batches. Goodrich and Pszona originally designed the algorithm in the external-memory model [23], while Barenboim and Elkin designed the algorithm for a distributed model [5]. Shi et al. adapted both algorithms for shared memory and showed that both compute an $O(\alpha)$-orientation in $O(m)$ work and $O(\log^2 n)$

ALGORITHM 1: Kowalik's Five-Cycle Counting Algorithm Parallelized

```
1: procedure COUNT-FIVE-CYCLES(G = (V, E))
2:     #_c ← 0
3:     Relabel vertices of G such that d(0) ≥ d(1) ≥ ··· ≥ d(n − 1)
4:     Orient G using arboricity orientation to produce G→
5:     parfor v ← 0 to n − 1 do
6:         Initialize an empty parallel hash table U_v
7:         parfor u ∈ N_v(v) do
8:             parfor w ∈ N_v(u) do U_v[w] ← U_v[w] + 1
9:         parfor u ∈ N_v(v) do
10:            Initialize an empty parallel hash table T_{v,u}
11:            parfor w ∈ N_v(u) do
12:                T_{v,u}[w] ← 1
13:            parfor w ∈ N_v(u) do
14:                parfor x ∈ N_v→(w) do
15:                    if x ≠ u then
16:                        if w ∈ N→(v) or v ∈ N→(w) then
17:                            #_c ← #_c + U_v[x] − T_{v,u}[x] − 1
18:                        else
19:                            #_c ← #_c + U_v[x] − T_{v,u}[x]
20:    return #_c
```

span (one of which is deterministic and the other of which is randomized) [43]. A different algorithm with the same (deterministic) work and span bounds was described by Besta et al. [8].

4.2 Kowalik's Algorithm

We present in Algorithm 1 our parallelization of Kowalik's serial five-cycle counting algorithm [28]. In this algorithm, vertices are sorted and processed in non-increasing degree order. Each vertex is processed by counting all five-cycles with the vertex itself as the lowest-ranked (i.e., highest-degree) vertex. After processing all vertices, each five-cycle is counted exactly once and the counts are summed.

Recall that we use $N_v(u)$ ($N_v^{\rightarrow}(u)$ for directed graphs) to represent the neighbors of vertex u that are after v in the non-increasing degree ordering. Since the vertices are relabeled by non-increasing degree order on line 3, we can easily obtain $N_v(u) = N(u) \cap \{w \in V \mid w > v\}$.

We now focus on the iteration v of the outer for-loop. An example is shown in Figure 1. We note that for each v, we consider only vertices ranked higher than v to complete five-cycles containing v. Lines 7 and 8 count in a parallel hash table U_v all wedges, where v is one of the endpoints and v is the lowest-ranked vertex in the wedge. Then, lines 11 and 12 store in a parallel hash table $T_{v,u}$ all wedges where v is one of the endpoints, v is the lowest-ranked vertex in the wedge, and u is the center. Both hash tables are indexed on w, the other endpoint of the wedge. We label each subfigure in Figure 1 with the corresponding parallel hash table U_v, where $v = 0$ for subfigures (a)–(e), and $v = 1$ for subfigure (f). However, note that we preemptively subtract each entry in $T_{v,u}$ from U_v in our labels, for fixed u; specifically, $u = 1$ for subfigures (a) and (b), $u = 2$ for subfigures (c) and (f), $u = 3$ for subfigure (d), and $u = 4$ for subfigure (e).

On each iteration of the loop in line 13, the algorithm counts all five-cycles that contain the wedge $v — u — w$. To accomplish this, the algorithm iterates through each neighbor x of w in $G^{\rightarrow}$ and considers the number of wedges that x shares with v, which is stored in $U_v[x]$. Note that three vertices of the cycle are given (v, u, and w), so the algorithm must ensure that the two vertices used to complete the cycle do not include these existing vertices. Each subfigure in Figure 1 considers

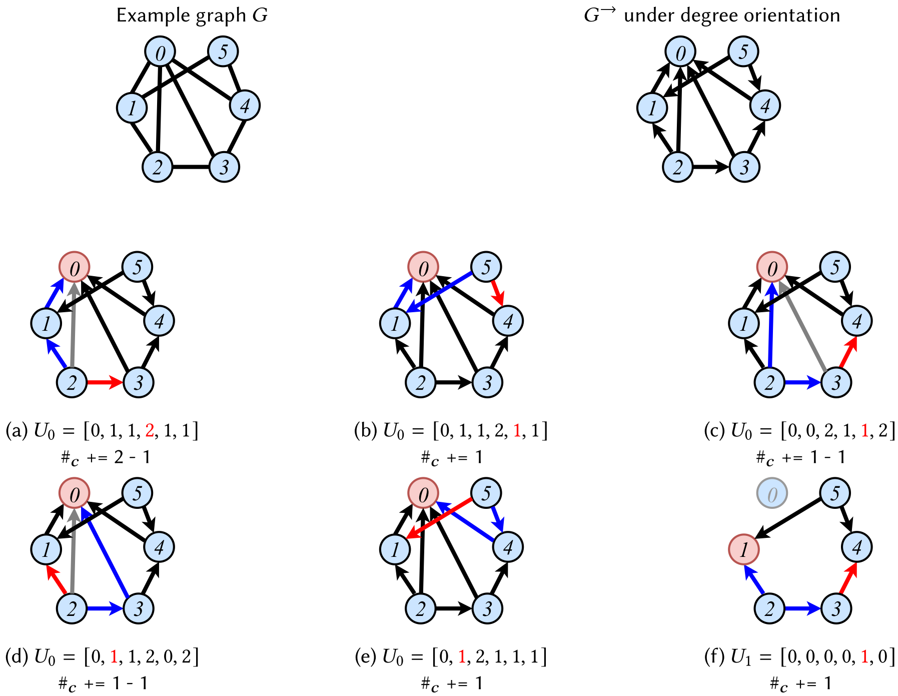

Fig. 1. This figure outlines steps in our parallelization of Kowalik's five-cycle counting algorithm where $\#_c$ is updated (Algorithm 1). Each subfigure considers a different $\{u, v, w, x\}$ from lines 13 and 14, and the corresponding U_v is displayed for each subfigure. For simplicity, the U_i hash tables are depicted as arrays, with the appropriate wedge counts stored at the index on the corresponding endpoint, and the updates to the parallel hash tables $T_{v,u}$ in lines 11 and 12 of Algorithm 1 are shown as subtracted directly from the corresponding U_v for a fixed u from line 9.
The vertices have already been relabeled by non-increasing degree and the entries in each U_v have already been computed (lines 10–12). The vertex v that we are considering on line 5 is colored in red. The edges colored in blue form wedges $v — u — w$, and the direction of those edges is irrelevant. The red edges represent the out-edge $w \rightarrow x$ on line 14. When w and v are neighbors (the edge is colored grey), the condition checked on line 16 returns true, and the subsequent line in each algorithm is executed (sub-figures (a), (c), and (d)). Otherwise, line 19 is executed (sub-figures (b), (e), and (f)). The final value of $\#_c$ is 4.

a different $\{u, v, w, x\}$, where $\{u, v, w\}$ are given by the blue edges, and x is given by the red edge. Line 15 ensures that $x \neq u$ in the cycle; note that $x \neq w$, because the graph is assumed to not contain self-loops, and $x \neq v$ by definition of $N_v^{\rightarrow}$. Lines 16–19 check if v and w are neighbors; if so, then the number of wedges ending in x includes the wedge $v — w — x$, which does not properly complete a five-cycle. In this case, there is one fewer five-cycle completed by the wedges ending in x, and so we subtract one on line 17. In Figure 1, this case holds when there is a grey edge, which is precisely the edge connecting v and w. Finally, note that if there exists the wedge $v — u — x$, then this similarly does not properly complete a five-cycle, so we subtract $T_{v,u}[x]$, which stores precisely this wedge. We assume that indexing an entry that does not exist in a hash table returns a value of 0. In Figure 1, this is already noted in our preemptive subtraction of $T_{v,u}$ from U_v in our labels.

As every thread operates on the variable $\#_c$, we use atomic add for all of these operations, which takes $O(1)$ work. In practice, we use thread-local variables to keep the count and sum them in the

ALGORITHM 2: Five-Cycle Counting in ESCAPE Parallelized

```
1: procedure COUNT-FIVE-CYCLES(G = (V, E))
2:     #_c ← 0
3:     Orient G using arboricity orientation to produce G→
4:     parfor v ← 0 to n − 1 do
5:         Initialize an empty parallel hash table U_v
6:         parfor w ∈ N←(v) do
7:             parfor u ∈ N(w) do
8:                 U_v[u] ← U_v[u] + 1
9:         parfor w ∈ N→(v) do
10:            parfor u ∈ N→(w) do
11:                U_v[u] ← U_v[u] + 1
12:        parfor u ∈ N←(v) do
13:            parfor w ∈ N←(u) do
14:                parfor x ∈ N→(w) do
15:                    if x ≠ v and x ≠ u then
16:                        #_c ← #_c + U_v[x]
17:                        if w ∈ N(v) then
18:                            #_c ← #_c − 1
19:                        if x ∈ N(u) then
20:                            #_c ← #_c − 1
21:    return #_c
```

end to avoid heavy contention. We now show that the parallel algorithm is work efficient and has polylogarithmic span.

THEOREM 4.2. *Algorithm 1 can be performed in $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p. and $O(m\alpha)$ space on a graph with m edges and arboricity α.*

PROOF. For line 3, we sort n integers in the range $[0, n-1]$, which can be done in $O(n)$ work and $O(\log n)$ span w.h.p. using parallel integer sorting [39]. As discussed in Section 4.1, line 4 can be implemented in $O(m)$ work and $O(\log^2 n)$ span [43]. As a result, the for-loops on lines 7 and 9 iterating over $u \in N_v(v)$ take at most $\min(\deg(u), \deg(v))$ iterations, and by Lemma 4.1, the total number of times we iterate through $w \in N_v(u)$ on each of lines 8, 11, and 13 is at most $2m\alpha$.

Since parallel hash tables can perform a batch of k operations in $O(k)$ work and $O(\log^* k)$ span w.h.p., the time complexities of lines 8 and 12 are given by $O(m\alpha)$ work and $O(\log^* n)$ span w.h.p. Then the for-loop of line 14 has at most $O(\alpha)$ iterations because of the $O(\alpha)$-orientation of the graph. In total, lines 15–19 are executed at most $O(\alpha) \cdot 2m\alpha = O(m\alpha^2)$ times, and again due to the parallel hash tables, the time complexity is given by $O(m\alpha^2)$ work and $O(\log^* n)$ span w.h.p. In all, the total time complexity is given by $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p.

Finally, this algorithm uses $O(m\alpha)$ space. Based on Lemma 4.1, the total number of keys stored over all U_v's is upper bounded by $O(m\alpha)$, as is the number of keys stored over all $T_{v,u}$'s (over all pairs (v, u)). The parallel hash table's space usage is linear in the number of keys [22]. Hence, the total space usage is $O(m\alpha)$. □

4.3 ESCAPE Algorithm

Another serial five-cycle counting algorithm is given by Pinar et al. as part of ESCAPE, which counts all 5-vertex subgraphs in a graph serially [36].

The first step of the ESCAPE five-cycle counting algorithm is to orient the graph. The ESCAPE framework uses degree orientation and achieves a time complexity of $O(m^2)$. We note that if

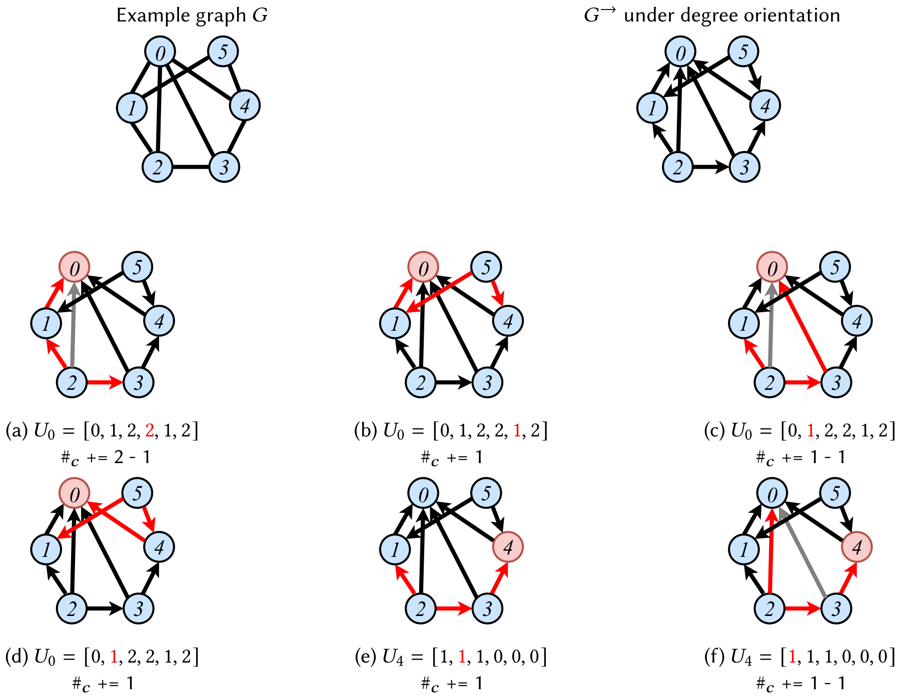

Fig. 2. This figure outlines steps in the ESCAPE five-cycle counting algorithm where $\#_c$ is updated (Algorithm 2). Each subfigure considers a different $\{u, v, w, x\}$ from lines 12–15, and the corresponding U_v is displayed for each subfigure. For simplicity, the U_i hash tables are depicted as arrays, with the appropriate wedge counts stored at the index on the corresponding endpoint. Note that the entries in each U_v have already been computed (lines 6–11). The vertex v that we are considering on line 4 is colored in red. The red edges represent the directed 3-paths $v \leftarrow u \leftarrow w \rightarrow x$ found on lines 12–15. Lines 17 and 19 check whether v and w or u and x are neighbors, respectively. When either of the conditions holds, the relevant edge is colored grey. Each grey edge subtracts one from the five-cycle count. Note that in sub-figures (a) and (c), the condition that v is adjacent to w from line 17 holds, and in sub-figure (f), the condition that u is adjacent to x from line 19 holds. In sub-figures (b), (d), and (e), neither conditions hold, and therefore 1 is not subtracted from the final count. The final value of $\#_c$ is 4.

instead an arboricity orientation is used, then the five-cycle counting algorithm achieves an improved time complexity of $O(m\alpha^2)$. We include this modification in our parallelization of the ESCAPE five-cycle counting algorithm to achieve work-efficient bounds. The proof of the serial time complexity with the arboricity orientation follows directly from the proof of our parallel algorithm.

We present in Algorithm 2 our parallelization of the algorithm from ESCAPE, and an example is shown in Figure 2. We use $u \prec v$ to indicate that u precedes v in the ordering that produced the orientation, and so an edge from u to v exists in the directed graph $G^{\rightarrow}$ if and only if $u \prec v$.

After orienting the graph using an arboricity orientation (line 3), for each vertex v (line 4), the algorithm counts all out-wedges and inout-wedges (see Figure 3). We denote the number of out-wedges with endpoints v and u by $W_{++}(v, u)$, and the number of inout-wedges with endpoints v and u, starting with a directed edge out of v, by $W_{+-}(v, u)$. For each v, the algorithm computes $W_{++}(u, v) + W_{+-}(u, v)$ on lines 6–8 and $W_{+-}(v, u)$ on lines 9–11, and stores these counts in a parallel hash table U_v. Note that lines 6–8 computes the total sum $W_{++}(u, v) + W_{+-}(u, v)$ rather than the constituent parts, so we iterate over all directed neighbors $w \in N^{\leftarrow}(v)$, and then all undirected neighbors $u \in N(w)$.

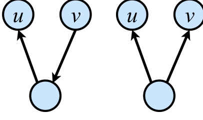

Fig. 3. An inout-wedge (left) and an out-wedge (right).

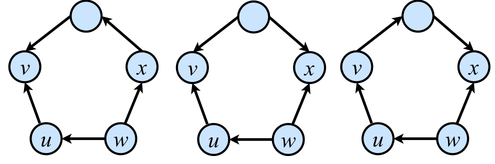

Fig. 4. All three possible acyclic orientations of directed five-cycles, assuming a graph orientation induced by a total ordering of the vertices. All three forms have the component $v \leftarrow u \leftarrow w \rightarrow x$, which is a 3-path between v and x. They are completed by an inout-wedge from x to v, an out-wedge between v and x, and an inout-wedge from v to x, respectively.

Figure 4 shows all possible orientations of acyclically directed five-cycles. We iterate over the 3-path shown in Figure 4 from vertex v to vertex x (lines 12–15), each of which can be completed by either an inout-wedge or an out-wedge with endpoints v and x, assuming $x \neq v$ and $x \neq u$. Now, any orientation of a five-cycle has one of the three configurations shown in Figure 4, where exactly one of the vertices can be assigned to be v. Thus, every 3-path between a pair (v, x) contributes $W_{+-}(v, x) + W_{++}(v, x) + W_{+-}(x, v)$ (which is stored in U from lines 6–11) to the five-cycle count. However, this over-counts five-cycles, since the wedge and the 3-path may overlap. Lines 16–20 deal with the over-counting when adding the number of wedges to the total count.

In more detail, Line 16 first adds $U_v[x]$ to the count (again, assume that indexing an entry that does not exist in a hash table returns a value of 0). Line 17 checks if w is adjacent to v; if so, then, depending on the direction of the edge between w and v, there is either an out-wedge or an inout-wedge on v, w, and x, that does not complete a five-cycle with the 3-path. Line 18 subtracts the five-cycle counted for this case. Similarly, line 19 checks if x is adjacent to u, and if so, then there is either an out-wedge or an inout-wedge on v, u, and x, that does not complete a five-cycle; line 20 corrects this.

Similarly to the parallelization of Kowalik's algorithm, in theory we use atomic adds for all of the increments on the $\#_c$ variable, and in practice we use thread-local variables.

Theorem 4.3. *Algorithm 2 can be performed in $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p., and $O(m\alpha)$ space on a graph with m edges and arboricity α.*

Proof. As discussed in Section 4.1, line 3 can be implemented in $O(m)$ work and $O(\log^2 n)$ span [43]. Lines 6–11 go through all inout-wedges and out-wedges where v is an endpoint. Because of the arboricity orientation, there are at most $m\alpha$ inout-wedges and out-wedges. Each wedge is counted at most twice, and so lines 6–11 incur $O(m\alpha)$ hash table operations, which takes $O(m\alpha)$ work and $O(\log^* n)$ span w.h.p.

There are $O(m\alpha^2)$ 3-paths (i.e., $v \leftarrow u \leftarrow w \rightarrow x$) and each is encountered exactly once in the triply-nested for-loop (lines 12–20). Again, by using an arboricity orientation, the algorithm executes lines 15–20 for at most $O(m\alpha^2)$ times, which due to the hash table operations, takes $O(m\alpha^2)$ work and $O(\log^* n)$ span w.h.p.

Overall, the algorithm takes $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p.

The parallel hash tables and the space to store the accumulated cycle counts account for all of the additional space usage. Since each wedge results in at most two additional keys in the hash tables, the number of keys in all of the hash tables U_i is upper bounded by twice the total number of out-wedges and inout-wedges. For the arboricity-oriented graph, there are $O(m\alpha)$ out-wedges

and $O(m\alpha)$ inout-wedges, and so the number of keys across all hash tables is bounded by $O(m\alpha)$. Thus, the algorithm takes $O(m\alpha)$ space. □

4.4 Approximate Counting

To further speed up five-cycle counting, we present here two techniques, *edge sparsification* and *colorful sparsification*, that allow us to compute approximate five-cycle counts in parallel. These techniques are based on prior work on approximate triangle, butterfly (bi-clique), and k-clique counting [35, 43, 44]. Both methods involve constructing a sparsified graph and running an exact parallel five-cycle counting algorithm on the sparsified graph. We prove that scaling the count returned gives an unbiased estimate of the total five-cycle count.

4.4.1 Edge Sparsification. In the *edge sparsification* method, we sparsify our input graph G by choosing to preserve each edge of G uniformly at random with probability p, in parallel. We call the resulting graph with only preserved edges G'. Then, we run our parallel five-cycle counting algorithm (Algorithm 1 or Algorithm 2) on G', which outputs a count C. We output $Y = Cp^{-5}$ as the estimate for the global five-cycle count.

We prove the following theorem about the properties of this estimator.

Theorem 4.4. *Let X be the true five-cycle count in G, and let C be the five-cycle count in the sparsified graph G' using probability p. Let $Y = Cp^{-5}$. Then, $\mathbb{E}[Y] = X$ and $\mathbb{V}ar[Y] = p^{-10}(X(p^5 - p^{10}) + \sum_{z=1}^{3} s_z(p^{10-z} - p^{10}))$ where s_z is the number of pairs of five-cycles that share z edges.*

Proof. Let X_i be the indicator variable denoting whether the ith five-cycle in G is preserved in G'. For a five-cycle to be preserved, all edges in the cycle must be preserved. This occurs with probability p^5, since each edge is preserved with probability p. Thus, since the number of five-cycles in G' is given by $C = \sum_i X_i$, we have $\mathbb{E}[C] = \sum_i \mathbb{E}[X_i] = Xp^5$. Moreover, $\mathbb{E}[Y] = \mathbb{E}[Cp^{-5}] = Xp^5 \cdot p^{-5} = X$, as desired.

Now, by definition, the variance is given by $\mathbb{V}ar[Y] = \mathbb{V}ar[Cp^{-5}] = p^{-10}\mathbb{V}ar[\sum_i X_i] = p^{-10}(\sum_i \mathbb{V}ar[X_i] + \sum_{i\neq j} \mathbb{C}ov[X_i, X_j])$. By definition, for all i, $\mathbb{V}ar[X_i] = \mathbb{E}[X_i^2] - \mathbb{E}[X_i]^2 = \mathbb{E}[X_i] - \mathbb{E}[X_i]^2 = p^5 - p^{10}$. Thus, $\mathbb{V}ar[Y] = p^{-10}(X(p^5 - p^{10}) + \sum_{i\neq j} \mathbb{C}ov[X_i, X_j])$.

Now, for $i \neq j$, $\mathbb{C}ov[X_i, X_j] = \mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j]$. This term depends on the number of edges that cycles i and j share. The covariance is 0 if they share no edges, since X_i and X_j are independent in this case. For pairs of cycles sharing z edges where z = 1, 2, or 3, we have $\mathbb{E}[X_i X_j] = Pr[X_i = 1] \cdot Pr[X_j = 1 \mid X_i = 1] = p^5 \cdot p^{5-z} = p^{10-z}$. The latter term is because the z shared edges are guaranteed to be preserved by the condition, so it remains to compute the probability of preserving the remaining $5 - z$ edges. Moreover, $\mathbb{E}[X_i]\mathbb{E}[X_j] = p^{10}$, so in total, $\mathbb{C}ov[X_i, X_j] = p^{10-z} - p^{10}$. If we let s_z denote the number of pairs of five-cycles that share z edges, then $\sum_{i\neq j} \mathbb{C}ov[X_i, X_j]) = \sum_{z=1}^{3} s_z(p^{10-z} - p^{10})$. In total, we have $\mathbb{V}ar[Y] = p^{-10}(X(p^5 - p^{10}) + \sum_{z=1}^{3} s_z(p^{10-z} - p^{10}))$, as desired. □

Corollary 4.5. *Let Δ denote the maximum degree of any vertex in G, and let X be the true five-cycle count in G. Let Y be the approximate five-cycle count computed using probability p. If $p \geq \max(208\Delta^3/X, (208\Delta^2/X)^{1/2}, (208\Delta/X)^{1/3}, (208/X)^{1/5})$, then $\mathbb{V}ar[Y] \leq X^2/8$.*

Proof. First, from Theorem 4.4, $\mathbb{V}ar[Y] = p^{-10}(X(p^5 - p^{10}) + \sum_{z=1}^{3} s_z(p^{10-z} - p^{10})) \leq Xp^{-5} + s_3 p^{-3} + s_2 p^{-2} + s_1 p^{-1}$ where s_z is the number of pairs of five-cycles that share z edges. Note that $s_z \leq X\binom{5}{z}\Delta^{4-z}$. This follows from fixing each five-cycle, and upper bounding the number of possible ways to extend the five-cycle to a new five-cycle such that the extension shares z edges with the original five-cycle and requires $5 - z$ additional edges. Substituting for s_z and the value of p in the corollary statement, we have $\mathbb{V}ar[Y] \leq Xp^{-5} + 10X\Delta p^{-3} + 10X\Delta^2 p^{-2} + 5X\Delta^3 p^{-1} \leq$

$X(208/X)^{(1/5)\cdot(-5)} + 10X\Delta(208\Delta/X)^{(1/3)\cdot(-3)} + 10X\Delta^2(208\Delta^2/X)^{(1/2)\cdot(-2)} + 5X\Delta^3(208\Delta^3/X)^{-1} = X^2/8$. □

Now, let Z be the mean of β independent instances of Y, where we define β later. Then, by Markov's inequality, $Pr[|Z - X| \geq \varepsilon X] = Pr[(Z - X)^2 \geq \varepsilon^2 X^2] \leq \frac{\mathbb{E}[(Z-X)^2]}{\varepsilon^2 X^2} = \frac{\mathbb{V}ar[Z]}{\varepsilon^2 X^2} = \frac{\mathbb{V}ar[Y]}{\varepsilon^2 X^2 \beta}$. Using Corollary 4.5, if $p \geq \max(208\Delta^3/X, (208\Delta^2/X)^{1/2}, (208\Delta/X)^{1/3}, (208/X)^{1/5})$, then $\mathbb{V}ar[Y] \leq X^2/8$. Thus, $Pr[|Z - X| \geq \varepsilon X] \leq \frac{X^2/8}{\varepsilon^2 X^2 \beta} = \frac{1}{8\varepsilon^2 \beta}$, and if we let $\beta = \frac{1}{8\varepsilon^2 \delta}$, then Z is a (ε, δ)-estimator of X. Notably, if we increase the number of samples β by a factor of b, then we can either decrease δ by a factor of b or ε by a factor of b^2. The opposite relation holds if we decrease β by a factor of b; namely, we can either increase δ by a factor of b or increase ε by a factor of b^2.

We further prove the following theorem about the complexity of our sampling algorithm based on edge sparsification.

THEOREM 4.6. *Our parallel five-cycle sampling algorithm using edge sparsification with probability p gives an unbiased estimate of the global five-cycle count and takes $O(pm\alpha^2 + m)$ expected work and $O(\log^2 n)$ span w.h.p.*

PROOF. First, Theorem 4.4 shows that our algorithm gives an unbiased estimate of the global count. We now discuss the work and span of our algorithm. Choosing each edge to preserve uniformly at random takes $O(m)$ work and $O(1)$ span. Creating a subgraph containing preserved edges can be done using a parallel filter and a parallel prefix sum in $O(m)$ work and $O(\log m)$ span. Since each edge is preserved with probability p, the subgraph has pm edges in expectation. The arboricity of the subgraph is upper bounded by the arboricity of the original graph, α. Thus, by Theorems 4.2 and 4.3, running our parallel five-cycle counting algorithm on the subgraph takes $O(pm\alpha^2)$ expected work and $O(\log^2 n)$ span w.h.p. In total, we have $O(pm\alpha^2 + m)$ expected work and $O(\log^2 n)$ span w.h.p. □

4.4.2 Colorful Sparsification. In the *colorful sparsification* method, we sparsify our input graph G by coloring each vertex of G uniformly at random with one of c colors, and preserving edges only if both endpoints are of the same color. Note that we can perform this sparsification in parallel. Let $p = 1/c$. We call the resulting graph with only preserved edges G'. Then, we run our parallel five-cycle counting algorithm (Algorithm 1 or Algorithm 2) on G', which outputs a count C. We output $Y = Cp^{-4}$ as the estimate for the global five-cycle count.

We prove the following theorem about the properties of this estimator.

THEOREM 4.7. *Let X be the true five-cycle count in G, and let C be the five-cycle count in the sparsified graph G' using c colors. Let $p = 1/c$, and let $Y = Cp^{-4}$. Then, $\mathbb{E}[Y] = X$ and $\mathbb{V}ar[Y] = p^{-8}(X(p^4 - p^8) + \sum_{z=2}^{4} t_z(p^{9-z} - p^8))$ where t_z is the number of pairs of five-cycles that share z vertices.*

PROOF. Let X_i be the indicator variable denoting whether the ith five-cycle in G is preserved in G'. For a five-cycle to be preserved, all edges in the cycle must be preserved, or in other words, all vertices in the cycle must be colored using the same color. This occurs with probability p^4, since after fixing the color of one vertex v in the cycle, the remaining four vertices must have the same color as v. Thus, since the number of five-cycles in G' is given by $C = \sum_i X_i$, we have $\mathbb{E}[C] = \sum_i \mathbb{E}[X_i] = Xp^4$. Moreover, $\mathbb{E}[Y] = \mathbb{E}[Cp^{-4}] = Xp^4 \cdot p^{-4} = X$, as desired.

Now, by definition, the variance is given by $\mathbb{V}ar[Y] = \mathbb{V}ar[Cp^{-4}] = p^{-8}\mathbb{V}ar[\sum_i X_i] = p^{-8}(\sum_i \mathbb{V}ar[X_i] + \sum_{i \neq j} \mathbb{C}ov[X_i, X_j])$. By definition, for all i, $\mathbb{V}ar[X_i] = \mathbb{E}[X_i^2] - \mathbb{E}[X_i]^2 = \mathbb{E}[X_i] - \mathbb{E}[X_i]^2 = p^4 - p^8$. Thus, $\mathbb{V}ar[Y] = p^{-8}(X(p^4 - p^8) + \sum_{i \neq j} \mathbb{C}ov[X_i, X_j])$.

Now, for $i \neq j$, $\mathbb{C}ov[X_i, X_j] = \mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j]$. This term depends on the number of vertices that cycles i and j share. The covariance is 0 if they share no vertices, and the covariance

is also 0 if they share one vertex, since the event that the remaining vertices of each cycle have the same color as the shared vertex is independent between the two cycles. For pairs of cycles sharing z vertices where $z = 2, 3$, or 4, we note that $\mathbb{E}[X_i X_j] = Pr[X_i = 1] \cdot Pr[X_j = 1 \mid X_i = 1] = p^4 \cdot p^{5-z} = p^{9-z}$. The latter term is because the z shared vertices are guaranteed to be the same color by the condition, and so it remains to compute the probability that the remaining $5 - z$ vertices are also of the same color. Moreover, $\mathbb{E}[X_i]\mathbb{E}[X_j] = p^8$, and so in total, $\mathbb{C}ov[X_i, X_j] = p^{9-z} - p^8$. If we let t_z denote the number of pairs of five-cycles that share z vertices, then $\sum_{i \neq j} \mathbb{C}ov[X_i, X_j]) = \sum_{z=2}^{4} t_z(p^{9-z} - p^8)$. In total, we have $\mathbb{V}ar[Y] = p^{-8}(X(p^4 - p^8) + \sum_{z=2}^{4} t_z(p^{9-z} - p^8))$, as desired. □

Corollary 4.8. *Let Δ denote the maximum degree of any vertex in G, and let X be the true five-cycle count in G. Let Y be the approximate five-cycle count computed using c colors, where $p = 1/c$. If $p \geq \max(208\Delta^3/X, (208\Delta^2/X)^{1/2}, (208\Delta/X)^{1/3}, (208/X)^{1/4})$, then $\mathbb{V}ar[Y] \leq X^2/8$.*

Proof. First, from Theorem 4.7, $\mathbb{V}ar[Y] = p^{-8}(X(p^4 - p^8) + \sum_{z=2}^{4} t_z(p^{9-z} - p^8)) \leq Xp^{-4} + t_4 p^{-3} + t_3 p^{-2} + t_2 p^{-1}$ where t_z is the number of pairs of five-cycles that share z vertices. Note that $t_z \leq X\binom{5}{z}\Delta^{5-z}$. This follows from fixing each five-cycle, and upper bounding the number of possible ways to extend the five-cycle to a new five-cycle such that the extension shares z vertices with the original five-cycle and requires $5 - z$ additional vertices. Substituting for t_z and the value of p in the corollary statement, we have $\mathbb{V}ar[Y] \leq Xp^{-4} + 5X\Delta p^{-3} + 10X\Delta^2 p^{-2} + 10X\Delta^3 p^{-1} \leq X(208/X)^{(1/4)\cdot(-4)} + 5X\Delta(208\Delta/X)^{(1/3)\cdot(-3)} + 10X\Delta^2(208\Delta^2/X)^{(1/2)\cdot(-2)} + 10X\Delta^3(208\Delta^3/X)^{-1} = X^2/8$. □

Again, let Z be the mean of β independent instances of Y, where we define β later. Then, by Markov's inequality, $Pr[|Z - X| \geq \varepsilon X] = Pr[(Z - X)^2 \geq \varepsilon^2 X^2] \leq \frac{\mathbb{E}[(Z-X)^2]}{\varepsilon^2 X^2} = \frac{\mathbb{V}ar[Z]}{\varepsilon^2 X^2} = \frac{\mathbb{V}ar[Y]}{\varepsilon^2 X^2 \beta}$. Using Corollary 4.8, if $p \geq \max(208\Delta^3/X, (208\Delta^2/X)^{1/2}, (208\Delta/X)^{1/3}, (208/X)^{1/4})$, then $\mathbb{V}ar[Y] \leq X^2/8$. Thus, $Pr[|Z - X| \geq \varepsilon X] \leq \frac{X^2/8}{\varepsilon^2 X^2 \beta} = \frac{1}{8\varepsilon^2\beta}$, and if we let $\beta = \frac{1}{8\varepsilon^2\delta}$, then Z is a (ε, δ)-estimator of X.

We further prove the following theorem about the complexity of our sampling algorithm based on colorful sparsification.

Theorem 4.9. *Our parallel five-cycle sampling algorithm using colorful sparsification with c colors where $p = 1/c$ gives an unbiased estimate of the global five-cycle count and takes $O(pm\alpha^2 + m)$ expected work and $O(\log^2 n)$ span w.h.p.*

Proof. First, Theorem 4.7 shows that our algorithm gives an unbiased estimate of the global count. We now discuss the work and span of our algorithm. Coloring each vertex takes $O(n)$ work and $O(1)$ span. Creating a subgraph containing edges where endpoints are the same color can be done using a parallel filter and a parallel prefix sum in $O(m)$ work and $O(\log m)$ span. Since each edge is preserved with probability $p = 1/c$, the subgraph has pm edges in expectation. The rest of the proof is the same as the proof of Theorem 4.6. □

4.5 Implementation

We implement the serial and parallel versions of Kowalik's algorithm and Pinar et al.'s algorithm, as well as our approximate algorithms, using the **Graph-based Benchmark Suite framework (GBBS)** [16, 17]. In GBBS, graphs are represented in CSR format, which does not allow us to check edge existence in $O(1)$ work; instead, we sort the neighbor lists in the preprocessing step and use binary search to locate neighbors.

In our approximate counting algorithms, the sparsification is done by simply using a hash function to sample the edges or assign a color to each vertex, and then applying a filter and prefix

sum to create the subgraph in CSR format. The rest of this section describes the optimizations for five-cycle counting either on the original graph (for exact counting) or on the sparsified graph (for approximate counting).

In our parallel implementations, we only parallelize the outer for-loop for each algorithm, since there is sufficient parallelism provided by the outer for-loop alone. For Kowalik's algorithm, instead of using parallel integer sort, we use a cache-efficient implementation of parallel sample sort [10] from GBBS to sort the vertices by degree. We also use vertex-indexed size-n arrays instead of parallel hash tables for U_i and $T_{i,j}$.

Thread-local Data Structures. As we parallelize the outer for-loop, the arrays U_i in both algorithms must be allocated per iteration. We optimize this allocation by using the `parallel_for_alloc` construct in GBBS, which allocates one array per thread and reuses this space over iterations. Each iteration uses the array as a local array, and so this incurs no synchronization overhead. With this optimization, the algorithm only requires $O(Pn)$ space, where P is the number of processors.

Fast Reset. Additionally, the thread-local arrays must be reset after each iteration of the outer for-loop. Depending on the structure of the graph, the array can be sparse, and naively resetting the entire array incurs $O(n^2)$ extra work, which is costly. We use a separate thread-local array to record the non-zero entries and reset only those entries after an iteration of the outer for-loop. This optimization at most doubles the space requirement for the algorithm, but drastically improves the running time by avoiding unnecessary writes.

Work Scheduling. The naive parallelization of the five-cycle counting algorithms blocks a fixed number of vertices together and processes them in series. For our experiments, we use a block size of 16, which we found to give the best performance in this setting. However, due to the nature of the algorithm, the amount of work per vertex is not uniform. This is particularly true for Kowalik's algorithm, which processes vertices in non-increasing degree order and deletes a vertex after processing it. The number of five-cycles that can be counted under a given vertex v in the outermost loop falls off rapidly with the vertex's degree rank. In our work scheduling optimization, we block vertices together into groups that require similar amounts of work by estimating the work required for each vertex. We use the sum of the degrees of a vertex's neighbors as the estimator. That is, for each vertex v, we estimate the amount of work done on the vertex to be $\sum_{w \in N(v)} \deg(w)$.

5 EXPERIMENTS

Environment. We run our experiments on a `c5.18xlarge` AWS EC2 instance, which is a dual-processor system with 18 cores per processor (two-way hyper-threading, 3.00-GHz Intel Xeon Platinum 8124M processors), and 144 GiB of main memory. We use Cilk Plus for parallelism [11, 29]. We use the g++ compiler (version 8.2.1) with the `-O3` flag.

We test the performance of our parallel exact and approximate five-cycle counting algorithms. Our parallel implementations use all of the optimizations described in Section 4.5, except that we test the performance with and without the work scheduling optimization. We compare the performance of the exact parallel implementations against our implementations of Kowalik's algorithm and the ESCAPE algorithm. We also tested the performance of the exact serial five-cycle counting algorithm in the ESCAPE package, the fastest known implementation of exact five-cycle counting. This algorithm is embedded inside the ESCAPE code for counting all five-vertex patterns, and so we obtained timings by running only the five-cycle counting portion of the code. We found our exact serial ESCAPE implementation to be 1.1–2.95× faster than the one provided in the ESCAPE package, and hence present only our running times in the tables.

Table 1. Relevant Statistics of Our Input Graphs

Dataset	$\|V\|$	$\|E\|$	No. 5-cycles
email-Eu-Core (email)	1005	32128	245,585,096
com-DBLP (dblp)	425957	2.10×10^6	3,440,276,253
com-YouTube (youtube)	1.16×10^6	5.98×10^6	34,643,647,544
com-LiveJournal (lj)	4.03×10^6	6.94×10^7	6,668,633,603,006
com-Orkut (orkut)	3.27×10^6	2.34×10^8	42,499,585,526,270
com-Friendster (friendster)	1.25×10^8	3.61×10^9	96,281,214,210,322

Table 2. Running Times (Seconds) of the Two Exact Serial Implementations and the Two Exact Parallel Five-cycle Counting Implementations without the Work Scheduling Optimization

	Exact Serial Algorithm		Exact Parallel Kowalik Algorithm				Exact Parallel ESCAPE Algorithm			
	Running times (s)		Running times (s)			Speedup	Running times (s)			Speedup
Dataset	T_E	T_K	T_1	T_{36}	T_{36h}	T_1/T_{36h}	T_1	T_{36}	T_{36h}	T_1/T_{36h}
email	0.36	0.026	0.027^b	0.0027^g	0.0029^b	9.3	0.376^b	0.017^g	0.0177^g	21.2
dblp	2.93	0.46	0.48^g	0.046^b	0.046^g	10.4	3.24^g	0.34^k	0.277^k	11.7
youtube	40.70	4.73	4.80^b	1.73^g	1.69^g	2.8	43.94^g	14.5^g	9.96^g	4.4
lj	2579.34	174.60	174.60^b	29.0^g	25.72^g	6.8	2582.30^g	426.38^g	308.41^g	8.4
orkut	38K	2878.38	2867.07^b	504.61^b	487.4^b	5.9	TL	8192.33^g	6384.24^g	—

All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. For the serial algorithms, T_E is our implementation of the ESCAPE algorithm with arboricity orientation, and T_K is our implementation of the serial Kowalik's algorithm. The serial runtimes are measured using the Goodrich–Pszona degeneracy ordering algorithm. For the parallel algorithms, we use superscripts to indicate the orientation that achieved the best running time. ^{g} refers to Goodrich–Pszona, ^{b} refers to Barenboim–Elkin, and ^{k} refers to k-core orientation. Note that degree orientation is never the fastest orientation. For the parallel algorithms, we list the runtimes obtained on a single thread (T_1), 36 cores without hyper-threading (T_{36}), and 36 cores with hyper-threading (T_{36h}). We also tested all implementations on friendster, but they all exceeded the time limit.

We also test the effect of using different parallel arboricity ordering algorithms. Besides Goodrich–Pszona, Barenboim–Elkin, and k-core, we also tested non-decreasing degree ordering as an approximation of degeneracy ordering. Intuitively, it limits the out-degree of the graph by directing edges from lower-degree vertices to higher-degree neighbors. Also, note that we use the parallel Gooodrich–Pszona and Barenboim–Elkin implementations by Shi et al. [43], and the parallel k-core implementation from Julienne [15], which is work efficient but does not have poly-logarithmic span. Finally, we test the accuracy of our approximate algorithms and their speedups over the exact algorithms.

We perform these tests on a number of real-world graphs from the Stanford Network Analysis Platform [30]. Table 1 describes the properties of these graphs. All graphs are simple, unweighted, and undirected.

Exact Serial Five-cycle Counting. Table 2 lists the running times of the two exact serial five-cycle counting algorithms. Our serial Kowalik implementation always outperforms our serial ESCAPE implementation, and the difference in running times between the ESCAPE algorithm and Kowalik's algorithm grows as the graph size grows. The serial Kowalik algorithm achieves between 6.37× and 14.77× speedup over our serial ESCAPE implementation and between 7× and 38.91× speedup over the original ESCAPE implementation.

Exact Parallel Five-cycle Counting. Table 2 shows the best performance with the exact Kowalik and ESCAPE algorithms with 1 thread, 36 cores without hyper-threading, and 72 hyper-threads, without the work scheduling optimization. We see that the algorithms achieve decent speedup

Table 3. These Are the Number of Binary Searches Each Exact Algorithm Performed for Each Dataset and the Ratio of the Number of Binary Searches in the ESCAPE Algorithm to Kowalik's Algorithm

	# binary searches		
Dataset	Kowalik	ESCAPE	ESCAPE/Kowalik
email	5.15×10^5	2.98×10^7	58
dblp	8.41×10^6	2.32×10^8	28
youtube	6.28×10^7	2.96×10^9	47
lj	1.39×10^9	1.72×10^{11}	124
orkut	1.25×10^{10}	3.44×10^{12}	273

Table 4. Single-thread (T_1) and 36-core with Hyper-threading (T_{36h}) Running Times (Seconds) of the Exact Parallel Kowalik and ESCAPE Algorithms with the Work Scheduling Optimization and Their Parallel Speedups

	Exact Parallel Kowalik Algorithm			Exact Parallel ESCAPE Algorithm		
	Running times (s)		Speedup	Running times (s)		Speedup
Dataset	T_1	T_{36h}	T_1/T_{36h}	T_1	T_{36h}	T_1/T_{36h}
email	0.0265[b]	0.00252[°]	10.5	0.357[b]	0.0165[b]	21.6
dblp	0.46[g]	0.0143[g]	32.2	3.07[b]	0.0866[g]	35.5
youtube	4.75[b]	0.338[b]	14.1	43.32[g]	1.42[g]	30.0
lj	171.92[b]	5.85[g]	29.4	2510.97[b]	58.75[g]	42.7
orkut	2858.18[b]	136.98[g]	20.9	TL	1269.1[b]	–
friendster	TL	8417.31[g]	–	TL	TL	–

All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. The superscripts indicate the orientation that achieved the best runtime. [g] refers to Goodrich–Pszona, [b] refers to Barenboim–Elkin, and [°] refers to degree orientation. In Table 5, we present the data for all orientations.

without the work scheduling optimization. The parallel speedup plateaus from 36 to 72 hyper-threads, especially for the parallelization of Kowalik's algorithm. The speedup for the Kowalik algorithm is usually lower, since, due to its degree ordering, it does not distribute work evenly across vertices, but rather concentrates the work on high-degree vertices. Our naive parallel algorithm groups a fixed number of vertices together regardless of whether they are high- or low-degree, resulting in unbalanced work distribution across workers.

From both the serial and parallel running times, we observe that the ESCAPE algorithm, with all of the same optimizations as the parallel Kowalik's algorithm, generally has about a 10× slowdown compared to Kowalik's algorithm. We attribute this difference to the discrepancy in the number of edge queries the two algorithms must perform. Since we store graphs in CSR format, each edge query requires a binary search. In Kowalik's algorithm, an edge query is performed for every (v, w)-pair, and it can be performed just before the for-loop with x, so there only needs to be $O(m\alpha)$ binary searches. In the ESCAPE algorithm, (x, u) needs to be queried $O(m\alpha^2)$ times. Table 3 shows that the ESCAPE algorithm does significantly more binary searches than Kowalik's algorithm.

Compared to the state-of-the-art serial five-cycle counting implementation provided in the ESCAPE package, without the work scheduling optimization, our parallel Kowalik implementation achieves a speedup of 33.78–229.79×, and our parallel ESCAPE implementation achieves a speedup of 5.73–23.16×.

Work Scheduling Optimization. We present the best running times of the exact parallel Kowalik and ESCAPE algorithms using work scheduling in Tables 4 and 5 shows the running times

Table 5. Single-thread (T_1) and 36-core with Hyper-threading (T_{36h}) Running Times (Seconds) of the Exact Parallel Kowalik and ESCAPE Algorithms with the Work Scheduling Optimization Using All Four Orientations

	Exact Kowalik			Exact ESCAPE		
	Running times (s)		Speedup	Running times (s)		Speedup
Dataset	T_1	T_{36h}	T_1/T_{36h}	T_1	T_{36h}	T_1/T_{36h}
email	0.0267	0.00289	9.3	0.396	0.0174	22.7
dblp	**0.459**	**0.0143**	32.2	3.17	**0.0866**	36.6
youtube	4.81	0.361	13.3	**43.3**	**1.42**	30.6
lj	174.40	**5.85**	29.8	2546.95	**58.75**	43.4
orkut	2867.78	**136.98**	20.9	TL	1552.70	—

(a) Goodrich–Pszona

	Exact Kowalik			Exact ESCAPE		
	Running times (s)		Speedup	Running times (s)		Speedup
Dataset	T_1	T_{36h}	T_1/T_{36h}	T_1	T_{36h}	T_1/T_{36h}
email	**0.0265**	0.00294	9.0	**0.357**	**0.0165**	21.6
dblp	0.465	0.0147	31.6	**3.07**	0.0992	31.0
youtube	**4.75**	**0.338**	14.0	48.05	1.43	33.6
lj	**171.92**	5.95	28.9	**2510.97**	59.03	42.5
orkut	**2858.18**	139.87	20.4	TL	**1269.07**	—

(b) Barenboim–Elkin

	Exact Kowalik			Exact ESCAPE		
	Running times (s)		Speedup	Running times (s)		Speedup
Dataset	T_1	T_{36h}	T_1/T_{36h}	T_1	T_{36h}	T_1/T_{36h}
email	0.0276	**0.00252**	10.9	1.49	0.0403	37.0
dblp	0.472	0.0144	32.7	10.71	0.773	13.9
youtube	4.79	0.344	13.9	2177.52	59.61	36.5
lj	178.02	5.96	29.9	16651.40	417.00	39.9
orkut	2949.47	139.37	21.2	TL	16129.4	—

(c) Degree

	Exact Kowalik			Exact ESCAPE		
	Running times (s)		Speedup	Running times (s)		Speedup
Dataset	T_1	T_{36h}	T_1/T_{36h}	T_1	T_{36h}	T_1/T_{36h}
email	0.0278	0.00304	9.2	0.675	0.0245	27.6
dblp	0.473	0.0161	29.3	7.80	0.240	32.4
youtube	5.84	0.531	11.0	922.19	23.62	39.0
lj	198.16	8.06	24.6	11151.50	309.71	36.0
orkut	3100.79	147.83	21.0	TL	7113.44	—

(d) k-Core

All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. The bold values mark the best serial and parallel runtimes for each of Kowalik and ESCAPE, out of the four orientations, which are used in Table 4.

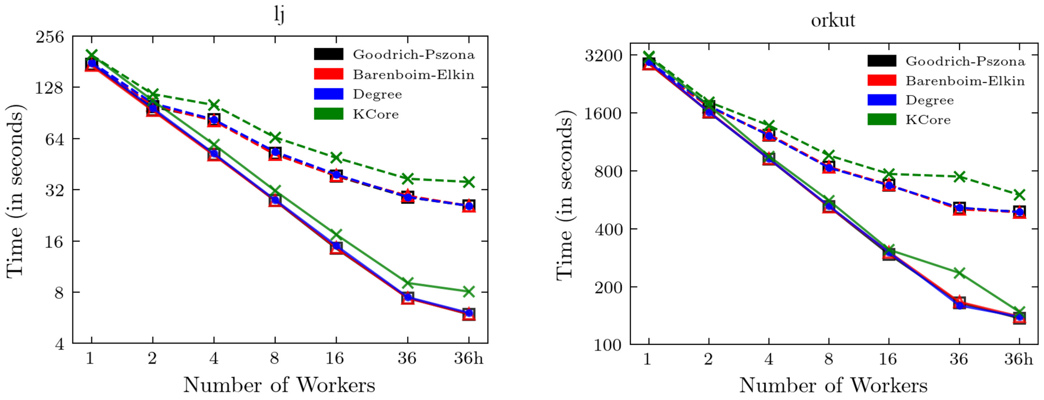

Fig. 5. Running time of the exact parallel Kowalik algorithm vs. number of threads. "36h" is 36 cores with hyper-threading. Dashed lines indicate that the work scheduling optimization is disabled and solid lines indicate that the work scheduling optimization is enabled. The lines for Goodrich–Pszona, Barenboim–Elkin, and degree ordering overlap each other for the most part.

for different orientations. Compared to Table 2, we see that the work scheduling optimization is effective on both parallel algorithms. It allows exact five-cycle counting to be performed on the friendster graph in under 2.5 hours using the parallel Kowalik algorithm. Figure 5 shows the relative running time of the parallel Kowalik algorithm with 72 hyper-threads with different arboricity orientation subroutines, including Goodrich–Pszona, Barenboim–Elkin, degree ordering, and k-core orientation, with and without the work scheduling optimization. The comparison shows that work scheduling significantly improves the running time and scaling of the parallel Kowalik algorithm.

Throughout our tests, we use the sum of neighbors' degrees as the estimator of the amount of work. Other work estimators were tested, including a simple out-degree count and the two-hop neighbor out-degree sum, but did not result in improved performance.

Compared to the state-of-the-art exact serial five-cycle counting implementation provided in the ESCAPE package, using the work scheduling optimization, our parallel Kowalik implementation achieves a speedup of 162.70–818.12×, and our parallel ESCAPE implementation achieves a speedup of 23.56–72.13×. Compared to our best serial baselines, our parallel Kowalik implementation achieves a speedup of 10.5–32.2×.

Graph Orientation. Figure 6 compares the performance of our exact parallel Kowalik implementation using different parallel orientation schemes [15, 43]. The Goodrich–Pszona and Barenboim–Elkin algorithms give very similar performance. k-core performs slightly worse on all graphs except for the small email graph. From our experiments, degree ordering results in running times that are comparable to both Goodrich–Pszona and Barenboim–Elkin.

While Goodrich–Pszona, Barenboim–Elkin, and k-core produce arboricity orderings, we may want to use degree ordering as it is much more efficient to compute and can compensate for the potentially worse counting time. Figure 7 shows the proportion of time spent on preprocessing (T_p) versus counting (T_c) on different orientation methods on three of the graphs. As the graph size grows, the preprocessing time takes up a smaller fraction of the total running time and becomes negligible in the case of the orkut graph. However, for smaller graphs, degree orientation has a clear advantage, because it takes much less time to compute while allowing for similar performance in the counting step. k-core ordering does not perform well when considering the times for both preprocessing and counting.

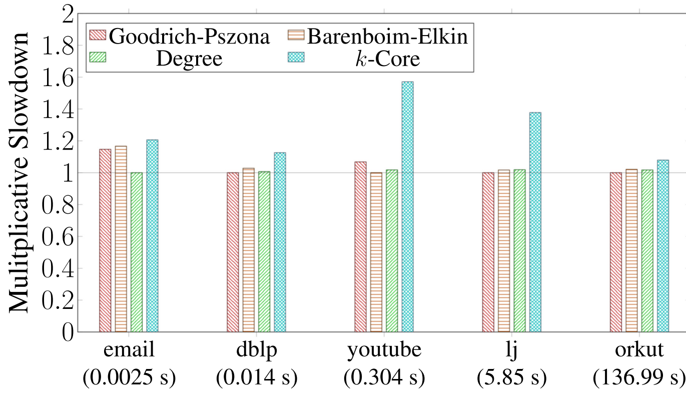

Fig. 6. Exact five-cycle counting times using our parallel Kowalik implementation, excluding preprocessing steps like relabeling and orienting the graph, under different orientation schemes for each of the graphs, using 36 cores with hyper-threading. The fastest time is labeled under each graph.

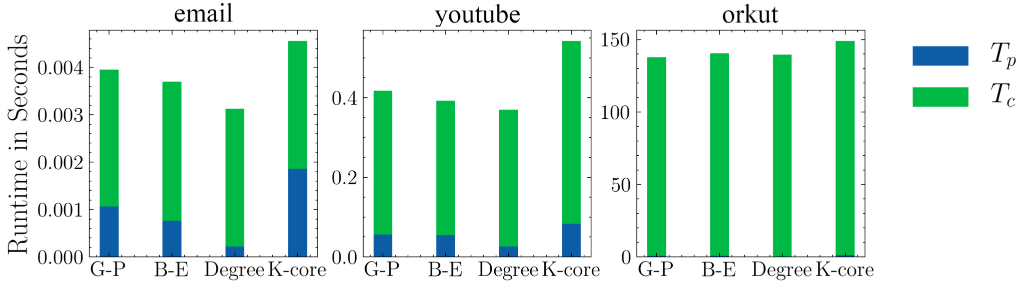

Fig. 7. Breakdown of time spent in the exact parallel Kowalik implementation on preprocessing (T_p) vs. counting (T_c) for different orientation subroutines, using 36 cores with hyper-threading. G-P is Goodrich–Pszona; B-E is Barenboim–Elkin. For the orkut graph, the time spent on preprocessing is not visible.

Approximate Five-Cycle Counting. For our approximate five-cycle counting experiments, we use the parallel Kowalik algorithm on the sparsified graph, as it outperforms the parallel ESCAPE algorithm in the exact setting. Figures 8 and 9 show the speedups of approximate five-cycle counting using edge sparsification and colorful sparsification, respectively, over exact five-cycle counting, considering different probabilities p for preserving edges and different numbers of colors c where $p = 1/c$, respectively. We generally observe higher speedups for smaller p, although for the smallest graphs email and dblp, the speedups degrade once the preserved subgraphs are small enough such that there are few to no five-cycles. Figures 10 and 11 show the fractional errors of the approximate five-cycle counts obtained using edge sparsification and colorful sparsification respectively, compared to the exact five-cycle counts. We compute our error using the formula |exact − approximate|/exact. For $p = 1/4$, across all graphs, we see 3.22–30.30× speedups of approximate five-cycle counting using edge sparsification over exact five-cycle counting, with error rates between 32.26% and 44.40%. For $p = 1/4$ using colorful sparsification, we see 2.99–38.16× speedups, with error rates between 0.20% and 11.95%. We observe more significant speedups for smaller p; in particular, for $p = 1/8$, across all graphs, we see 8.90–113.42× speedups using edge sparsification, with error rates between 9.93% and 26.70%, and we see 9.10–188.94× speedups using colorful sparsification, with error rates between 0.52% and 11.77%. For larger p, we note that while $p = 1/2$ reduces the error rate in colorful sparsification to a median of 0.53%, it gives much

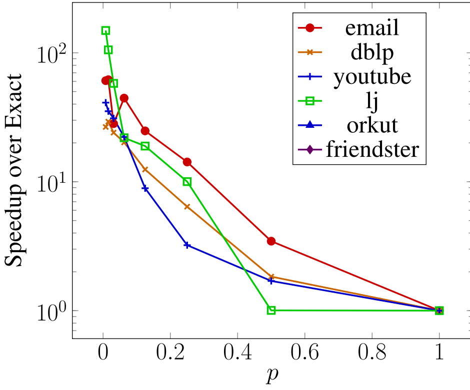

Fig. 8. These are the parallel speedups over exact running times for approximate five-cycle counting using edge sparsification, varying over different p. The best runtimes considering all orientations were used. The speedups are given in a log-scale.

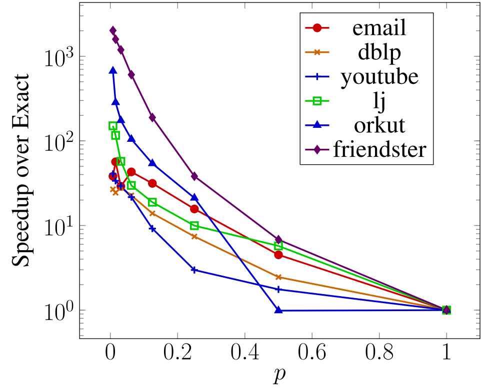

Fig. 9. These are the parallel speedups over exact running times for approximate five-cycle counting using colorful sparsification, varying over $p = 1/c$, where c is the number of colors used. The best runtimes considering all orientations were used. The speedups are given in a log-scale.

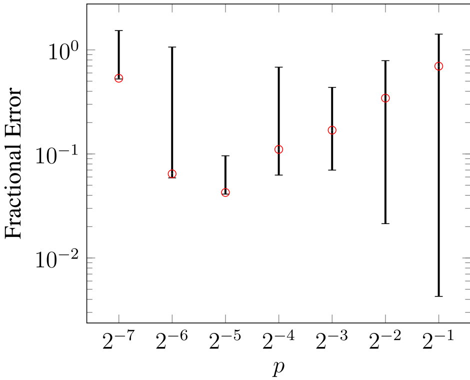

Fig. 10. These are the fractional errors of the approximate five-cycle count obtained using edge sparsification, varying over different p. The errors are taken over all graphs, with the red circle marking the median error and the bars marking the minimum and maximum errors.

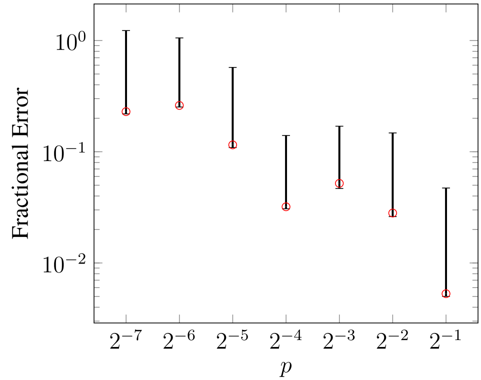

Fig. 11. These are the fractional errors of the approximate five-cycle count obtained using colorful sparsification, varying over $p = 1/c$, where c is the number of colors used. The errors are taken over all graphs, with the red circle marking the median error and the bars marking the minimum and maximum errors.

lower speedups, of up to 4.63× for edge sparsification and up to 6.80× for colorful sparsification, and the variance in the percentage error is higher in edge sparsification. Overall, we observe more significant speedups on larger graphs, and we note that colorful sparsification generally produces more accurate five-cycle counts compared to edge sparsification. Notably, on friendster, which takes over 2 hours to complete exact five-cycle counting, we can obtain an approximate count with 0.20% error in under a minute using colorful sparsification, with $p = 1/4$.

Figures 12 and 13 show the self-relative parallel speedups of approximate five-cycle counting using edge sparsification and colorful sparsification, respectively, over different numbers of workers, where $p = 1/8$ and using the Goodrich–Pszona orientation. We note that very little self-relative

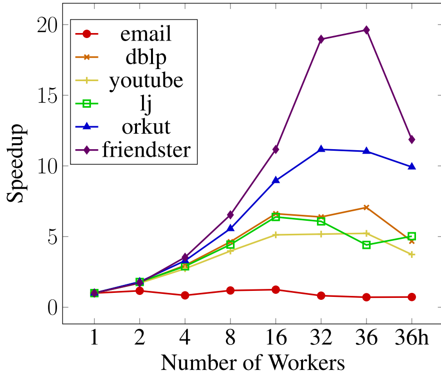

Fig. 12. These are the self-relative speedups of approximate five-cycle counting over the single-threaded running times, using edge sparsification with $p = 1/8$ over varying numbers of workers. "36h" refers to 36 cores with hyper-threading. The Goodrich–Pszona orientation was used.

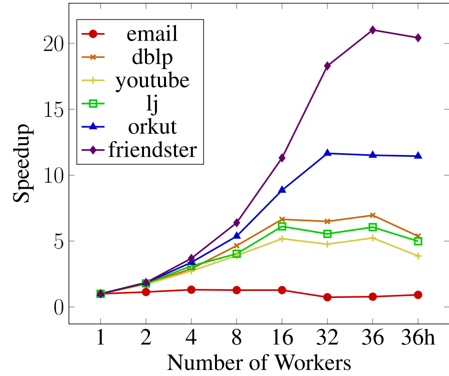

Fig. 13. These are the self-relative speedups of approximate five-cycle counting over the single-threaded running times, using colorful sparsification with eight colors ($p = 1/8$) over varying numbers of workers. "36h" refers to 36 cores with hyper-threading. The Goodrich–Pszona orientation was used.

speedup is observed on the smaller graphs, particularly the email graph, where the sparsified graph is exceedingly small. The largest self-relative speedups appear on the largest graphs, and on these large graphs, we see good scalability. Overall, we observe up to 11.87× and 20.43× self-relative speedups over the single-threaded running times, using edge sparsification and colorful sparsification, respectively. For these experiments, we see that hyper-threading usually does not help, likely due to the smaller sizes of the graphs compared to the experiments on the exact algorithms.

6 CONCLUSION

We designed the first theoretically work-efficient parallel five-cycle counting algorithms with polylogarithmic span. On 36 cores, our best exact implementation outperforms the fastest existing exact serial implementation by up to 818×, and achieves self-relative speedups of 10–46×. Our best approximate implementation, for a fixed probability parameter $p = 1/8$, achieves up to 20.43× self-relative speedups and is able to approximate five-cycle counts 9.10–188.94× faster than our best exact implementation, with between 0.52% and 11.77% error. Designing parallel algorithms for counting larger cycles is interesting for future work, although such algorithms are likely to require super-linear work, even for low-arboricity graphs [6].

REFERENCES

[1] A. Abboud and V. V. Williams. 2014. Popular conjectures imply strong lower bounds for dynamic problems. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*. 434–443.
[2] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, Nick G. Duffield, and Theodore L. Willke. 2017. Graphlet decomposition: Framework, algorithms, and applications. *Knowl. Inf. Syst.* 50, 3 (2017), 689–722.
[3] N. Alon, R. Yuster, and U. Zwick. 1997. Finding and counting given length cycles. *Algorithmica* 17, 3 (1997), 209–223.
[4] Richard Anderson and Ernst W. Mayr. 1984. *A* P*-complete Problem and Approximations to It.* Technical Report.
[5] Leonid Barenboim and Michael Elkin. 2010. Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition. *Distrib. Comput.* 22, 5 (2010), 363–379.
[6] Suman K. Bera, Noujan Pashanasangi, and C. Seshadhri. 2020. Linear time subgraph counting, graph degeneracy, and the chasm at size six. In *Proceedings of the Innovations in Theoretical Computer Science Conference*, Vol. 151. 38:1–38:20.
[7] Jonathan W. Berry, Luke K. Fostvedt, Daniel J. Nordman, Cynthia A. Phillips, C. Seshadhri, and Alyson G. Wilson. 2014. Why do simple algorithms for triangle enumeration work in the real world? In *Proceedings of the Conference on Innovations in Theoretical Computer Science*. 225–234.

[8] Maciej Besta, Armon Carigiet, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, and Torsten Hoefler. 2020. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Article 99.
[9] M. A. Bhuiyan, M. Rahman, M. Rahman, and M. Al Hasan. 2012. GUISE: Uniform sampling of graphlets for large graph analysis. In *Proceedings of the IEEE International Conference on Data Mining*. 91–100.
[10] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 189–199.
[11] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.
[12] Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM* 21, 2 (April 1974), 201–206.
[13] Ronald S. Burt. 2004. Structural holes and good ideas. *Am. J. Sociol.* 110, 2 (2004), 349–399.
[14] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM J. Comput.* 14, 1 (February 1985), 210–223.
[15] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'17)*. Association for Computing Machinery, New York, NY, 293–304. https://doi.org/10.1145/3087556.3087580
[16] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 393–404.
[17] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The graph based benchmark suite (GBBS). In *Proceedings of the Joint International Workshop on Graph Data Management Experiences & Systems (GRADES'20) and Network Data Analytics (NDA'20)*. Article 11, 8 pages.
[18] Ethan R. Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G. Dimakis. 2015. Beyond triangles: A distributed framework for estimating 3-profiles of large graphs. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 229–238.
[19] Ethan R. Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G. Dimakis. 2016. Distributed estimation of graph 4-profiles. In *Proceedings of the International Conference on World Wide Web*. 483–493.
[20] Giorgio Fagiolo. 2007. Clustering in complex directed networks. *Phys. Rev. E* 76, 2 (2007), 026107.
[21] Katherine Faust. 2010. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Soc. Netw.* 32, 3 (2010), 221–233.
[22] J. Gil, Y. Matias, and U. Vishkin. 1991. Towards a theory of nearly constant time parallel algorithms. In *Proceedings of the IEEE Symposium of Foundations of Computer Science*. 698–710.
[23] Michael T. Goodrich and Paweł Pszona. 2011. External-memory network analysis algorithms for naturally sparse graphs. In *Proceedings of the European Symposium on Algorithms*. 664–676.
[24] Tomaz Hocevar and Janez Demsar. 2014. A combinatorial approach to graphlet counting. *Bioinformatics* 30, 4 (2014), 559–565.
[25] Paul W. Holland and Samuel Leinhardt. 1970. A method for detecting structure in sociometric data. *Am. J. Sociol.* 76, 3 (1970), 492–513.
[26] Louisa Ruixue Huang, Jessica Shi, and Julian Shun. 2021. Parallel five-cycle counting algorithms. In *Proceedings of the International Symposium on Experimental Algorithms (SEA'21)*, Vol. 190. 2:1–2:18.
[27] Joseph Jaja. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.
[28] Łukasz Kowalik. 2003. Short cycles in planar graphs. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*. 284–296.
[29] Charles E. Leiserson. 2010. The Cilk++ concurrency platform. *J. Supercomput.* 51, 3 (2010).
[30] Jure Leskovec and Andrej Krevl. 2019. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved from http://snap.stanford.edu/data.
[31] Don R. Lick and Arthur T. White. 1970. k-degenerate graphs. *Can. J. Math.* 22, 5 (1970), 1082–1096. https://doi.org/10.4153/CJM-1970-125-1
[32] David W. Matula and Leland L. Beck. 1983. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM* 30, 3 (July 1983).
[33] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The graph structure in the web–analyzed on different aggregation levels. *J. Web Sci.* 1, 1 (2015), 33–47.
[34] Crispin Nash-Williams. 1964. Decomposition of finite graphs into forests. *J. Lond. Math. Soc.* s1-39, 1 (1964), 12–12.
[35] Rasmus Pagh and Charalampos E. Tsourakakis. 2012. Colorful triangle counting and a mapreduce implementation. *Inf. Process. Lett.* 112, 7 (March 2012), 277–281.
[36] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. 2017. ESCAPE: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the International Conference on World Wide Web*. 1431–1440.

[37] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.* 11, 12 (August 2018), 1876–1888.

[38] M. Rahman, M. A. Bhuiyan, and M. Al Hasan. 2014. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2466–2478.

[39] S. Rajasekaran and J. H. Reif. 1989. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.* 18, 3 (June 1989), 594–607.

[40] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. 2018. Butterfly counting in bipartite networks. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2150–2159.

[41] Ahmet Erdem Sarıyüce and Ali Pinar. 2018. Peeling bipartite networks for dense subgraph discovery. In *Proceedings of the ACM International Conference on Web Search and Data Mining*. 504–512.

[42] T. Schank. 2007. Algorithmic aspects of triangle-based network analysis. Ph.D. Dissertation. Universitat Karlsruhe (2007).

[43] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Parallel clique counting and peeling algorithms. In *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA'21)*. SIAM, 135–146. https://doi.org/10.1137/1.9781611976830.13

[44] Jessica Shi and Julian Shun. 2020. Parallel algorithms for butterfly computations. In *Proceedings of the SIAM Symposium on Algorithmic Principles of Computer Systems*, Bruce M. Maggs (Ed.). 16–30.

[45] Charalampos E. Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. 2011. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Soc. Netw. Anal. Min.* 1, 2 (1 April 2011), 75–81.

[46] J. Wang, A. W. Fu, and J. Cheng. 2014. Rectangle counting in large bipartite graphs. In *Proceedings of the IEEE International Congress on Big Data*. 17–24.

[47] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2019. Vertex priority based butterfly counting for large-scale bipartite networks. *Proc. VLDB Endow.* 12, 10 (June 2019), 1139–1152.

[48] Sebastian Wernicke and Florian Rasche. 2006. FANMOD: A tool for fast network motif detection. *Bioinformatics* 22, 9 (2006), 1152–1153.

[49] Xiangzhou Xia. 2016. *Efficient and Scalable Listing of Four-Vertex Subgraphs*. Master's thesis. Texas A&M University.

[50] R. Zhu, Z. Zou, and J. Li. 2018. Fast rectangle counting on massive networks. In *Proceedings of the IEEE International Conference on Data Mining*. 847–856.

Received 15 December 2021; revised 6 July 2022; accepted 29 July 2022