RC25352 (WAT1302-023) February 7, 2013
Computer Science

IBM Research Report

Parallel and Distributed Triangle Counting on Graph Streams

A. Pavan[1], Kanat Tangwongan[2], Srikanta Tirthapura[1]

[1]Iowa State University

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

Parallel and Distributed Triangle Counting On Graph Streams[*]

(Regular Submission)

A. Pavan[‡] Kanat Tangwongsan[†] Srikanta Tirthapura[‡]

[‡]*Iowa State University and* [†]*IBM Research*

Abstract

This paper studies the problem of approximating the number of triangles in a graph whose edges arrive as a stream. The number of triangles in a graph is an important metric in social network analysis, link classification and recommendation, and more. We present efficient algorithms for both shared-memory parallel (multicore-like) processing of a single stream where edges arrive in batches and processing of streams by multiple physically distributed processors. For the shared-memory setting, we give the first parallel cache-oblivious algorithm with good theoretical guarantees for accuracy; we also experimentally verify that the algorithm obtains substantial speedups and accurate estimates. For the distributed setting, we present a distributed algorithm with low message complexity, improving upon existing sketch-based algorithms in common settings.

Keywords: graph streams, triangle counting, parallel cache-oblivious, distributed streams

[*]Contact Address: 1101 Kitchawan Rd, Yorktown Heights, NY 10598. E-mail: `pavan@cs.iastate.edu, ktangwo@us.ibm.com, snt@iastate.edu`

1 Introduction

The number of triangles in a graph is an important metric in social network analysis[28, 21], identifying thematic structures of networks [10], spam and fraud detection [2], link classification and recommendation [26], among others. Driven by these applications, the problem of accurately estimating the number of triangles in a large graph has been an active area of research. We consider this problem in the setting of an *evolving* massive graph, whose edges arrive as a centralized or a distributed stream.

There have been several prior streaming algorithms for this problem, including [1, 13, 6, 20, 14, 22, 12]. However, they have almost exclusively focused on processing the stream using a single processor and cannot effectively take advantage of parallelism[1]. On the other hand, there have been parallel algorithms for triangle counting, such as [25, 9]. While these algorithms excel at processing static graphs using multiple processors, they cannot efficiently handle constantly changing graphs.

In this work, we design algorithms with both the above advantages: they are able to use multiple processors and are able to process, in a single pass, an evolving graph whose edges arrive as a stream in an arbitrary order. We consider two related settings: **(1)** a shared-memory multicore machine and **(2)** a distributed set of processors connected by a network. In either model, our goal is to obtain an implementation that is as fast as possible, given the CPU and memory resources at our disposal. Towards this goal, we have used the following models to guide the algorithm design.

First, we work in the *limited-space streaming model*, where we assume that the entire graph cannot be stored in memory; perhaps, only a small fraction can be. For instance, a graph of size 32GB can be processed with less than 8GB of memory, with accurate results, using the shared-memory algorithm. As a result, our algorithms are able to process evolving large graphs, using a machine with relatively modest resources. Then, to effectively use a multicore machine without being tied to a specific memory-hierarchy configuration, we have used the *parallel cache-oblivious model (PCO)*. As a result, the algorithm need not know the parameters of the caches, yet makes efficient use of them (cost measured as the number of cache misses). Finally, in the distributed setting, the stream is physically distributed among different processors that process them in parallel, and communicate with each other. We pay particular attention to minimize the amount of communication among processors.

Our algorithms provide a randomized relative-error approximation to the number of triangles. Given $\varepsilon, \delta \in [0, 1]$, a random variable $\hat{X}$ is a (ε, δ) approximation of X if $\mathbf{Pr}\left[|\hat{X} - X| \geq \varepsilon X\right] \leq \delta$.

1.1 Our Contributions

In the context of our overall goal of parallel processing of very large streaming graphs, we make the following contributions:

—**Parallel Cache-Oblivious Triangle Counting:** We present the first parallel cache-oblivious algorithm for approximately counting the number of triangles in a graph stream. Our algorithm processes edges in batches, and *the processing cost of each batch is no more expensive than that of a cache-optimal sort in the parallel cache-oblivious (PCO) model*, allowing us to use prior algorithms for parallel multicore sorting to process triangles in a graph. To our knowledge, this is also the first parallel streaming algorithm for estimating the number of triangles on a multicore machine. Our parallel algorithms rely on a technique called *neighborhood sampling*, proposed in our recent work [22].

—**Low Message Complexity Distributed Triangle Counting:** In the distributed model with k sites and a coordinator, we present an algorithm that returns an (ε, δ)-estimator for the number of triangles in the graph, using communication $O(m\Delta \log m \log \Delta / \tau(G))$, where m is the number of edges, Δ the maximum vertex degree, and $\tau(G)$ the number of triangles in the graph. The message complexity of this algorithm is substantially smaller than previous

[1]While many such algorithms maintain multiple independent copies of the same "estimator" logic which will be trivial to update in parallel, an efficient sequential implementation can often do much better than explicitly updating them all; a good parallel implementation must not perform much more *total work* than the efficient sequential implementation, so it can be applied efficiently to both a modest number of processors (*one* being the most modest) and a larger number.

distributed streaming algorithms [20, 14, 13].

—**Experimental Study of the Parallel Algorithm:** Results from our implementation of the above parallel algorithm on a multicore machine, using real-world networks indicate substantial speedup when compared with a sequential version. On a machine with 12 cores, we obtain up to 11.24x speedup on large datasets; this lets us process a graph with 1.2 billion edges (Twitter-2010) in less than 2 minutes, for a processing throughput of more than 10 million edges per second.

Roadmap. We summarize the model and primitives used in the parallel algorithms in Section 2, review the neighborhood sampling technique in Section 3, give multicore algorithms in Section 4, distributed algorithms in Section 5, and experimental results in Section 6.

1.2 Related Work

Approximate triangle counting has been well studied in both streaming and non-streaming settings. In the streaming context, beginning with the work of Bar-Yossef et al. [1], several algorithms have been proposed for the case of single streams [13, 6, 20, 14]: an algorithm of [13] uses $\tilde{O}(m\Delta^2/\tau(G))$[2] space whereas the algorithm of [6] uses $\tilde{O}(mn/\tau(G))$ space. With higher space complexity, [20] and [14] gave algorithms for the more general problem of counting cliques and cycles, supporting the insertion and deletion of edges. In a recent work, the authors proposed a single-stream algorithm with space complexity $\tilde{O}(m\Delta/\tau(G))$ [22]. Kolountzakis et al. [15] presented a multipass streaming algorithm for approximate triangle counting, requiring space $(m^{1/2}\log n + 1/\varepsilon^2 m^{3/2}\Delta \log n/\tau(G))$ and a constant number of passes. More recently, Jha et al. [12] gave a $O(\sqrt{n})$ space algorithm for estimating the number of triangles and the closely related problem of computing the clustering coefficient of a graph stream. Their algorithm has an additive error guarantee as opposed to the algorithms mentioned earlier, which had relative error guarantees. The related problem of approximating the triangle count associated with each vertex has also been studied in the streaming context [2, 16].

Some of the aforementioned streaming algorithms build linear "sketches" [1, 20, 13], which can be easily combined in a distributed setting, resulting in the message complexity of $O(ks)$, where k is the number of distributed stream processors and s is the space complexity of the single stream algorithm. The algorithms we present in this paper build upon the single stream algorithm of [22], which is not sketch-based. As we describe in Section 5, the message complexity of our algorithm is better than the above bounds especially for typical graph instances.

In the non-streaming (batch) context there are many works on counting and enumerating triangles—both exact and approximate [7, 18, 23, 27, 3, 8]. Recent works on parallel algorithms in the MapReduce model include [25, 9].

2 Preliminaries and Notation

Throughout the paper, denote by $G = (V, E)$ a simple, undirected graph. The edges of a given graph arrive as a stream; we assume that every edge arrives exactly once. Use m to denote the number of edges and Δ to denote the maximum degree. An edge $e \in E$ is a size-2 set consisting of its endpoints. In this notation, for $e, f \in E$, we say that e is **adjacent** to f or e is **incident** on f if they share a vertex—i.e., $|e \cap f| = 1$. When the graph G has a total order (e.g., imposed by the stream arrival order), we denote by $\mathcal{S} = (V, E, \leq_{\mathcal{S}})$ the graph $G = (V, E)$, together with a total order $\leq_{\mathcal{S}}$ on E. The total order fully defines the standard relations $<_{\mathcal{S}}, >_{\mathcal{S}}, \geq_{\mathcal{S}}, =_{\mathcal{S}}, \neq_{\mathcal{S}}$, which we use without explicitly defining. When the context is clear, we sometimes drop the subscript. Further, for a sequence $A = \langle a_1, \ldots, a_{|A|}\rangle$, we write $G_A = (V_A, E_A, \leq_A)$, where V_A is the relevant vertex set, $E_A = \{a_1, \ldots, a_{|A|}\}$, and $\leq_A$ is the total order defined by the sequence order. Given $\mathcal{S} = (V, E, \leq_{\mathcal{S}})$, the **neighborhood of an edge** $e \in E$, denoted by $\Gamma_{\mathcal{S}}(e)$, is the set of all edges in E that "appear after" e in the $\leq_{\mathcal{S}}$ order; that is, $\Gamma_{\mathcal{S}}(e) := \{f \in E : f >_{\mathcal{S}} e\}$.

Let $\mathcal{T}(G)$ (or $\mathcal{T}(\mathcal{S})$) denote the set of all triangles in G—i.e., the set of all closed triplets, and $\tau(G)$ be the number of triangles in G. For a triangle $t^* \in \mathcal{T}(G)$, define $C(t^*)$ to be $|\Gamma_{\mathcal{S}}(f)|$, where f is the smallest edge of t^* w.r.t. $\leq_{\mathcal{S}}$. Finally, the notation $x \in_R S$ indicates that x is a random sample from S taken uniformly at random.

[2]The notation $\tilde{O}$ suppresses factors polynomial in $\log m$, $\log(1/\delta)$, and $1/\varepsilon$.

Parallel Model. Parallel algorithms in this work are expressed in the nested parallel model. It allows arbitrary dynamic nesting of parallel loops and fork-join constructs but no other synchronizations, corresponding to the class of algorithms with series-parallel dependency graphs. More details about the model appear in Appendix C.

We will analyze memory cost of parallel algorithms in the parallel cache-oblivious (PCO) model [4], a parallel variant of the cache oblivious (CO) model. The Cache Oblivious (CO) model [11] is a model for measuring cache misses of an algorithm when run on a single processor machine with a two-level memory hierarchy: one level of finite cache and unbounded memory. Specifically, it counts $Q(n; M, B)$ the number of cache misses incurred by a problem instance of size n when run on a fully associative cache of size M and line size B using the optimal (offline) cache replacement policy.

Extending the CO model, the parallel cache-oblivious (PCO) model gives a way to analyze the number of cache misses for the tasks that run in parallel in a parallel block. In PCO, the cache complexity of an algorithm A is denoted by $Q^*(A; M, B)$ and behaves as a "work-like" cost measure. When applied to a parallel machine, it represents the number of misses across all processors. An algorithm in this model relies on an appropriate scheduler to evenly balance the load.

Parallel Primitives. Throughout this work, we assume the *tall cache assumption*; that is, $M \geq \Omega(B^2)$. We describe our algorithms in terms of primitives such as sorting, prefix sums, merge, filter, and map. These primitives have parallel algorithms with optimal cache complexity in the PCO model and polylogarithmic depth (for detail, see [5, 4]). On input of length n, the cache complexity of sorting in the PCO model is $Q^*(\texttt{sort}(N); M, B) = O(\frac{N}{B}\log_{M/B}(1+\frac{N}{B}))$, and the complexity of the other primitives is $Q^*(\texttt{scan}(N); M, B) = O(N/B)$. We also write $\texttt{sort}(N)$ and $\texttt{scan}(N)$ to denote the corresponding cache costs when the context is clear. All these primitives have at most $O(\log^2 N)$ depth.

In addition, we will rely on a primitive for looking up multiple keys from a sequence of key-value pairs. Specifically, let $S = \langle (k_1, v_1), \ldots, (k_n, v_n)\rangle$ be a sequence of n key-value pairs, where k_i belongs to a total-order domain of keys $\mathbb{K}$. Also, let $T = \langle k'_1, \ldots, k'_m\rangle$ be a sequence of m keys from the same domain. The *exact multisearch problem* (`exactMultiSearch`) is to find for each $k'_j \in T$ the matching $(k_i, v_i) \in S$. We will also use the *predecessor multisearch* (`predEQMultiSearch`) variant, which asks for the pair with the largest key no larger than the given key. The exact version has a simple hash table implementation that will not be cache friendly. But existing cache-optimal `sort` and `merge` routines directly imply an implementation with $\texttt{sort}(n) + \texttt{sort}(m) + O(\texttt{scan}(n+m))$ cost:

Lemma 2.1 *There are algorithms* $\texttt{exactMultiSearch}(S, T)$ *and* $\texttt{predEQMultiSearch}(S, T)$ *each running in* $O(\log^2(n+m))$ *depth and* $O(\texttt{sort}(n) + \texttt{sort}(m))$ *cache complexity, where* $n = |S|$ *and* $m = |T|$*. Furthermore, if* S *and* T *have been presorted in the key order, these algorithms take* $O(\log(n+m))$ *depth and* $O(\texttt{scan}(n+m))$.

3 Neighborhood Sampling

The algorithms in this paper rely on a technique, called neighborhood sampling, for selecting a random triangle. The technique was implicit in the streaming algorithm in our recent work (under submission) [22]. In this section, we restate it as a set of invariants (Invariant 3.1) and show how they lead to an estimate for the number of triangles in a graph (Lemma 3.2).

Invariant 3.1 (Neighborhood Sampling Invariant (NBSI)) *Let* $\mathcal{S} = (V, E, \leq_{\mathcal{S}})$ *denote a simple, undirected graph* $G = (V, E)$*, together with a total order* $\leq_{\mathcal{S}}$ *on* E*. The tuple* (f_1, f_2, f_3, δ)*, where* $f_i \in E \cup \{\emptyset\}$ *and* $\delta \in \mathbb{Z}_+$*, satisfies the* ***neighborhood sampling invariant (NBSI)*** *if*

(1) ***Level-1 Edge:*** $f_1 \in_R E$ *is chosen uniformly at random from* E*;*
(2) $\delta = |\Gamma_{\mathcal{S}}(f_1)|$ *is the number of edges in* $\mathcal{S}$ *incident on* f_1 *that appear after* f_1 *according to* $\leq_{\mathcal{S}}$*.*
(3) ***Level-2 Edge:*** $f_2 \in_R \Gamma_{\mathcal{S}}(f_1)$ *is chosen uniformly from the neighbors of* f_1 *that appear after it (or* $\emptyset$ *if the neighborhood is empty); and*
(4) ***Closing Edge:*** $f_3 >_{\mathcal{S}} f_2$ *is an edge that closes the wedge* $f_1 f_2$ *(or* $\emptyset$ *if the closing edge is not present).*

This invariant gives a way to maintain a random—although non-uniform—triangle in a graph. The following lemma shows how to turn this into an unbiased estimator for τ (the proof of this lemma, which originally appears in [22], is reproduced in Appendix B for reference):

Lemma 3.2 (An Unbiased Estimator [22]) *Let $\mathcal{S} = (V, E, \leq_{\mathcal{S}})$ denote a simple, undirected graph $G = (V, E)$, together with a total order $\leq_{\mathcal{S}}$ on E. Further, let (f_1, f_2, f_3, δ) be a tuple satisfying NBSI. Define random variable X as: $X = 0$ if $f_3 = \emptyset$ and $X = \delta \cdot |E|$ otherwise. Then, $\mathbf{E}[X] = \tau(G)$.*

The proof of the following appears in Appendix B.

Theorem 3.3 ([22]) *There is an (ε, δ)-approximation to the triangle counting problem that on input a graph G with m edges, uses $\left(\frac{96}{\varepsilon^2} \cdot \frac{m\Delta(G)}{\tau(G)} \cdot \log(\frac{1}{\delta})\right)$ independent estimators.*

4 Parallel Triangle Counting

In this section, we present a parallel cache-oblivious algorithm for approximating the triangle count $\tau(G)$. We first describe a conceptual algorithm for maintaining one estimator for $\tau(G)$, and then describe a parallel algorithm to efficiently update multiple estimators simultaneously.

We note that a trivial way to parallelize r estimators is to update them in parallel; however, this method performs significantly more work than the efficient sequential counterpart [22] because the trivial parallelization performs r times more work than that of a single estimator whereas the efficient sequential algorithm does asymptotically the same as the work of 1 estimator. Hence, we cannot hope to improve upon the sequential algorithm by parallelizing across estimators, but we need to parallelize over the stream elements, and this is the technical challenge here. The main result of this section is summarized in the following theorem:

Theorem 4.1 *Let r be the number of estimators maintained. There is an algorithm* `bulkUpdateAll` *for triangle counting such that a batch update of size s takes $O(\mathtt{sort}(r) + \mathtt{sort}(s))$ I/O cost and $O(\log^2(r + s))$ depth.*

With a batch size of $s = \Theta(r)$, the total cost of processing a batch is $O(\mathtt{sort}(r))$ or equivalently, an amortized update cost of $O(\frac{1}{B} \log_{M/B}(1 + r/B))$—comparable to the cost of performing a single lookup in a table of size r.

4.1 One Estimator, Bulk Arrival

We discuss how to maintain the neighborhood sampling invariant for *one estimator* when a batch of edges arrives. Let (f_1, f_2, f_3, δ) be a NBSI-satisfying tuple on the graph $G = (V, E)$ and the total order $\leq_{\mathcal{S}}$ on E. Let $W = \langle w_1, \ldots, w_s \rangle$ be a sequence of arriving edges; the sequence defines a total order on W. Denote by $\mathcal{S}' = (V', E', \leq_{\mathcal{S}'})$ the graph on $E \cup W$, where the edges of W all come after the edges of E in the new total order.

In this setting, the level-1 and closing edges are relatively straightforward to maintain: For the level-1 edge f_1, we use a simple variant of reservoir sampling: with probability $\frac{|W|}{|W|+|E|}$, replace f_1 with an edge uniformly chosen from W; otherwise, keep the current edge. For the closing edge f_3, we only have to check the presence of the anticipated closing edge after f_2.

More care is needed, however, to handle level-2 edge f_2 and δ. By definition, f_2 is a random edge from the set $\Gamma_{\mathcal{S}'}(f_1) = \{e > f_1 : e \cap f_1 \neq \emptyset\}$—or, in words, the set of edges incident on f_1 that "appear after" it in the $\mathcal{S}'$ order. In this view, δ is simply the size of $\Gamma_{\mathcal{S}'}(f_1)$, and f_2 is a random sample from an appropriate "substream." Like in the previous case, a variant of reservoir sampling gives us the following update rules:

1. If f_1 was replaced by an edge in W, assign to f_2 a random edge from W incident on f_1 that appears after it in W; that is, $f_2 \in_R \Gamma_W(f_1)$. Furthermore, we set δ to $|\Gamma_W(f_1)|$.
2. Otherwise, we know that there are $d_0 = \delta$ edges before W that can be a level-2 edge and there are $d_1 = |\Gamma_W(f_1)|$ edges inside W that can be level-2 edge. Therefore, reset δ to $d_0 + d_1$. Now, with probability $\frac{d_0}{d_0+d_1}$, we keep the current f_2—and with the remaining probability, we pick a random edge from $\Gamma_W(f_1)$.

Although tedious to spell out, these rules are conceptually simple; however, from an efficiency point of view, it is clear that to maintain NBSI, in particular the level-2 edge, we will need to identify and sample from $\Gamma_W(f_1)$ apart from computing parameters such as d_1. This is an important challenge which we address next.

4.2 Parallel Bulk Update

The previous section describes a conceptual algorithm for updating one estimator. In this section, we describe a parallel algorithm to update multiple estimators efficiently. Let r be the number of *indepedent* estimators the user decides to maintain. We keep $est_1, est_2, \ldots, est_r$, each maintaining a NBSI-satisfying tuple (f_1, f_2, f_3, δ). These give r unbiased estimates (Lemma 3.2), ready for aggregation using, for example, Theorem 3.3.

As outlined earlier, a key challenge involves maintaining the level-2 edges: specifically, how to efficiently compute the number of candidate edges d_1 for every estimator and how to sample uniformly from these candiates? We address this challenge in 2 steps: First, we define the notion of *rank* and present a fast preprocessing algorithm so that rank queries can be answered efficiently. Second, we show how to sample efficiently with it and how it relates to the number of potential candiates.

Definition 4.2 (Rank) *Let $W = \langle w_1, \ldots, w_s \rangle$ be a sequence of unique edges. Let $G_W = (V_W, W)$ be the graph on the edges W, where V_W is the set of relevant vertices. For $x, y \in V_W$, $x \neq y$, the* rank *of $x \to y$ is*

$$\text{rank}(x \to y) = \begin{cases} |\{j : x \in w_j \wedge j > i\}| & \textit{if } \exists i, \{x, y\} = w_i \\ \deg_{G_W}(x) & \textit{otherwise} \end{cases}$$

In words, if $\{x, y\}$ is an edge in W, $\text{rank}(x \to y)$ is the number of edges in W that is incident on x and appears after xy in W. For other pair of vertices, $\text{rank}(x \to y)$ is simply the degree of x in the graph G_W. This function is, in general, not symmetric: $\text{rank}(x \to y)$ is not the same as $\text{rank}(y \to x)$.

Computing rank efficiently. The following lemma shows how to compute the rank of $x \to y$ and $y \to x$ for every edge $\{x, y\} \in W$; it outputs a sequence of length $2|W|$ in an order convenient for the bulk-update algorithm:

Lemma 4.3 *There is a parallel algorithm* $\texttt{rankAll}(W)$ *that takes a sequence of edges $W = \langle w_1, \ldots, w_s \rangle$ and produces a sequence of length* $2|W|$, *where each entry is a record* $\{\texttt{src}, \texttt{dst}, \texttt{rank}, \texttt{pos}\}$ *such that*

1. $\{\texttt{src}, \texttt{dst}\} = w_i$ *for some* $i = 1, \ldots, |W|$*;*
2. $\texttt{pos} = i$*; and*
3. $\texttt{rank} = \text{rank}(\texttt{src} \to \texttt{dst})$.

The algorithm runs in $O(\texttt{sort}(|W|))$ *I/O complexity and depth in the PCO model*

Proof: We describe an algorithm and reason about its complexity. First, we form a sequence F, where each $w_i = \{u, v\}$ appears twice, one per direction. Each directed edge is labeled with the position in the original sequence W. That is, each $w_i = \{u, v\}$ gives rise to $\{\texttt{src} = u, \texttt{dst} = v, \texttt{pos} = i\}$ and $\{\texttt{src} = v, \texttt{dst} = u, \texttt{pos} = i\}$. This step can be accomplished in $O(\texttt{scan}(|W|))$ I/O cost and depth.

In the view of a directed edge $e \in F$, $\text{rank}(e.\texttt{src} \to e.\texttt{dst})$ is the number of edges in F emanting from $e.\texttt{src}$ with $\texttt{pos} > e.\texttt{pos}$. To take advantage of this equivalent definition, we sort F by $\texttt{src}$ and for two entries of the same $\texttt{src}$, order them in the decreasing order of $\texttt{pos}$. This has $O(\texttt{sort}(|W|))$ I/O cost and depth. Now, in this ordering, the rank of a particular edge is one more than the rank of the edge immediately before it unless it is the first edge of that $\texttt{src}$; the latter has rank 0. Consequently, the rank computation for the entire sequence has cost $O(\texttt{scan}(|W|))$ since all that is required is to figure out whether an edge is the first edge of that $\texttt{src}$ and a prefix scan operation. Overall, the algorithm runs in $O(\texttt{sort}(|W|) + \texttt{scan}(|W|))$ cost, as claimed. ■

Updating the estimators. We have now defined rank and showed how to compute it for all relevant pairs. The following easy-to-verify observation establish the connection between rank and the sample space for level-2 edges:

Observation 4.4 *Let $f_1 = \{u, v\}$ be an edge, either previously processed or new. Let $F = \texttt{rankAll}(W)$. The set of edges in W incident on f_1 that appears after f—i.e., the set $\Gamma_W(f_1)$—is exactly*[3]

$$\{e \in F : e.\texttt{src} = u, e.\texttt{rank} < \text{rank}(u \to v)\} \cup \{e \in F : e.\texttt{src} = v, e.\texttt{rank} < \text{rank}(v \to u)\}.$$

[3]With a slight abuse of notation but to simplify the presentation, these edges are treated as undirected for the time being although technically they are directed and contain other information.

This directly gives the size of the sample space for level-2 sampling given a level-1 edge f_1. For a level-1 edge $f_1 = \{u, v\}$, there are $\mathrm{rank}(u \to v)$ edges in $\Gamma_W(f_1)$ incident on u and $\mathrm{rank}(v \to u)$ edges, on v. Therefore, the d_1 value for this edge is $d_1 = \mathrm{rank}(u \to v) + \mathrm{rank}(v \to u)$. This lets us update δ easily.

The observation, in fact, tells us more: in particular, for $f_1 = \{u, v\}$, the edges incident on u has rank values $0, 1, \ldots, \mathrm{rank}(u \to v) - 1$; likewise, the edges incident on v has rank values $0, 1, \ldots, \mathrm{rank}(v \to u) - 1$. This gives a "naming system" that forms the basis for efficient sampling of level-2 edges: pick a number ϕ between 0 and $\mathrm{rank}(u \to v) + \mathrm{rank}(v \to u) - 1$ uniformly at random. If $\phi < \mathrm{rank}(u \to v)$, pick the edge with `src` $= u$ and `rank` $= \phi$; otherwise, pick the edge with `src` $= v$ and `rank` $= \phi - \mathrm{rank}(u \to v)$.

With this in place, the main algorithm involves multisearch queries of the following forms:

(Q1) Given (u, r), locate an edge with `src` $= u$ with the highest rank less than or equal to r.

(Q2) Given (u, r), locate an edge with `src` $= u$ with the rank exactly equal to r.

(Q3) Given (u, v), locate the edge (u, v).

Armed with these ingredients, we are ready to give an algorithm and analyze it to prove Theorem 4.1:

Proof of Theorem 4.1: Let $W = \langle w_1, \ldots, w_s \rangle$ be a sequence of s edges. Let m be the number of the edges prior to the arrival of W. We first present an outline and explain how to implement the steps after that:

Step 1: Let $F \leftarrow$ `rankAll`(W). As a convenient byproduct, F is ordered by `src`, then by `rank`—or equivalently, F is ordered by `src`, then inversely by `pos`.

Step 2: For $i = 1, \ldots, r$, flip a coin with probability $\frac{w}{m+w}$ to decide if it will replace $f_1[i]$ with a random edge from W and pick a random replacement edge accordingly. In addition, store in $d_0[i] \leftarrow \delta$ if estimator i keeps the edge and $d_0[i] \leftarrow 0$ if it sought a replacement.

Step 3: For $i = 1, \ldots, r$, compute $xd_1[i] = \mathrm{rank}(u \to v)$ and $yd_1[i] = \mathrm{rank}(v \to u)$, where $\{u, v\} = f_1[i]$.

Step 4: For $i = 1, \ldots, r$, flip a coin with probability $\frac{d_1[i]}{d_0[i]+d_1[i]}$, where $d_1[i] = xd_1[i] + yd_1[i]$ to decide if it will replace $f_2[i]$. Find level-2 replacement edges accordingly.

Step 5: For $i = 1, \ldots, r$, form the candidate closing edge $h[i]$ and look for it.

Step 1 can be computed by Lemma 4.3 in $O(\texttt{sort}(s))$ cost. Then, in Step 2, we flip r coins in parallel and for the estimators that need a replacement edge, we pick a random number $\texttt{idx}[i] \in_R \{0, \ldots, r-1\}$ and "extract" these indices from W. This can be accomplished in $O(\texttt{scan}(r) + \texttt{scan}(s))$ cost using standard primitives (Section 2) For Step 3, compute the rank using a multisearch with $2r$ queries of the form *(Q1)*, which can be answered in $O(\texttt{sort}(r) + \texttt{scan}(r + s))$ total cost using Lemma 2.1.

In Step 4, we perform r coin flips in parallel using `map` and for the estimators needing a replacement edge, we use the naming system discussed above. Therefore, locating replacement level-2 edges becomes a multisearch with at most r queries of the form *(Q2)*. This step costs $O(\texttt{sort}(r) + \texttt{scan}(r + s))$. Finally, with a `map` operation, we compute the candidate closing edge and use a multisearch with at most r queries of the form *(Q3)* to see if they are present and come after the level-2 edge (by checking their `pos`). But since neither F nor W was appropriately sorted, we need another sort on $O(s)$ elements. So, this last step costs $O(\texttt{sort}(s) + \texttt{sort}(r) + \texttt{scan}(r + s))$. In total, the cost to perform a batch update is $O(\texttt{sort}(s) + \texttt{sort}(r))$. This concludes the proof. ■

5 Distributed Streams

We present triangle counting algorithms for distributed streams. Consider a system with k different sites, numbered $1, \ldots, k$, where site i receives a local stream $\mathcal{S}_i$; each element in the stream is an edge of an undirected graph. In addition, there is a *separate* coordinator node. Assume, for simplicity, that the coordinator does not observe any local stream. Let $\mathcal{S} = \cup_{i=1}^{k} \mathcal{S}_i$ be the conceptual stream observed by the whole system. The order of arrival of edges and the distribution of edges between the sites is arbitrary (possibly adversarially chosen). The goal is to return an estimate of $\tau(G)$ after the entire stream $G = G(\mathcal{S})$ is observed. The primary cost metrics are: **(1)** *Communication*: the number of messages and the number of bytes sent, assuming that edge and node ids, and the random weights can be stored in a constant number of bytes, and **(2)** *Memory*: the memory usage per processor and combined.

We first note that prior sketch-based algorithms for counting triangles (e.g., [20, 14, 13]) translate directly into distributed algorithms; these algorithms effectively compute linear transformations of the input vectors, so the sketch of the union of data is simply the sum of the individual sketches. This means the message cost of [20] in the distributed model is $\tilde{O}(km^3/\tau^2(G))$, which is also their processing time per item. The message cost of [14] is $\tilde{O}(km^3\Delta^3/\tau^2(G))$, and the second algorithm of [13] requires communication and per-item processing time of $\tilde{O}(k(m^3 + mC_4 + C_6)/\tau^2(G))$. In the following, we present a distributed algorithm with a message cost of $\tilde{O}(km\Delta/\tau(G))$—a significant improvement for typical input instances.

Distributed Algorithms for $\tau(G)$. We first present a basic distributed algorithm that gives an unbiased estimator for $\tau(G)$. Like before, the final algorithm runs several independent copies of this algorithm and combine their results. The algorithm is based on neighborhood sampling, and the communication cost is mainly determined by how the Level-1 edges and the Level-2 edges are maintained in a distributed manner.

The Level-1 edge f_1 is chosen uniformly at random from the set of all edges seen so far across all sites, using a distributed sampling procedure. Once f_1 is picked, the second level edge f_2 is chosen uniformly at random (again by using a distributed sampling procedure) from all neighbors of f_1 that arrive after f_1 is chosen. The sites also keep track of the total number of edges of the graph m and the total number of neighbors of f_1, c, that arrive after f_1 is picked. Each site checks whether a triangle can be formed by using f_1, f_2 and an edge from the local stream. If at least one site can form a triangle, the coordinator outputs cm otherwise the coordinator outputs 0. The algorithm is described below.

Algorithm 5.1 A distributed algorithm for $\tau(G)$

```
Initialization: At each site set $f_1, f_2$ to $\phi$, $c = 0$, $t = \phi$, $\ell_1 = \ell_2 = 1$. At the coordinator, set $g_1 = g_2 = 1$.
When site $i$ gets edge $(u, v)$:
    • Let $w_1(e)$ be a random real number in $[0, 1]$.
    • (Level-1 Sampling) If $(w_1(e) < \ell_1)$, then set $f_1 = e$; $f_2 = \phi$; $\ell_1 = w_1(e)$; $\ell_2 = 1$; $c = 0$.
      Send $\langle 1, e, w_1(e) \rangle$ to the Coordinator
    • (Level-2 sampling) If $e$ was not sampled in Level-1 and $e$ is adjacent to $f_1$, then set $c \leftarrow c + 1$.
      Let $w_2(e)$ be a random real number in $[0, 1]$;
      If $(w_2(e) < \ell_2)$ then set $f_2 = e$; $\ell_2 = w_2(e)$, and send $\langle 2, e, w_2(e) \rangle$ to Coordinator
    • (Triangle completion) If $e$ was not sampled in Level-1 and Level-2, then if $e, f_1, f_2$ form a triangle, then set
      $t \leftarrow \{e, f_1, f_2\}$.
When site $i$ receives $\langle b, w, e \rangle$ from coordinator: If $(b = 1)$ then set $f_1 = e$, $\ell_1 = w$, $f_2 = \phi$, $c = 0$, $t = \phi$;
    if $(b = 2)$ then set $f_2 = e$; $\ell_2 = w$; $t = \phi$;
When coordinator receives a message $\langle b, w, e \rangle$ from a site: If $(b = 1)$ then If $g_1 < w$, send $\langle b, w, e \rangle$ to all sites; $g_1 = w$; If
    $(b = 2)$ then If $g_2 < w$, send $\langle b, w, e \rangle$ to all sites; $g_2 = w$;
At the end of all observations: Each site $i$ sends $\langle c^i, t^i, m^i \rangle$ to the central coordinator, where $c^i$, $t^i$ are respectively the values
    of $c$, and $t$ local to node $i$, and $m^i = |S_i|$. The coordinator sets $c \leftarrow \sum_{i=1}^{k} c^i$, and $m \leftarrow \sum_{i=1}^{k} m^i$. If any of the $t^i$s is a
    triangle, then return $c \cdot m$ else return 0.
```

Correctness and Analysis. For the sake of analysis, we consider a global total order among all the edges arriving at all sites. We assume without loss of generality that at any time instance only site receives an edge. If two sites receive an edge at the same time, ties can be broken according to site id. This implicitly defines a total order $\leq_S$ among the edges. We assign a sequence number, from 1 to m, to each message based on this total order. Let e_i denote the i-th edge in this order. Note that this order is used for the analysis only; the algorithm is oblivious to it. Further, we assume that the all communications are instantaneous. This assumption simplifies the analysis and can be removed by using a buffer at each site. In the rest of the paper, we use ψ to denote $\frac{m\Delta}{\tau(G)} \frac{1}{\varepsilon^2} \log(\frac{1}{\delta})$. Due to lack of space, all proofs in this section appear in Appendix D.

Lemma 5.1 *The expected of messages and words exchanged in processing a graph G by Algorithm 5.1 is* $O(k \log m \log \Delta)$.

Theorem 5.2 *There is a distributed algorithm that returns an* (ε, δ)*-estimator of* $\tau(G)$ *using total number of messages* $O(\psi k \log \Delta \log m)$*, and* $O(\psi)$ *memory per site.*

We now describe three methods for improving the communication cost relative to the above algorithm.

Batch Processing. One improvement is that a processor does not process an edge at a time, but processes a batch of edges at once. Suppose that we consider a batch size of $O(B)$. The algorithm at a site waits until the memory fills up with B edges, and only then communicates with the coordinator. When the coordinator hears from a site, all the batches stored at all the sites are processed, and estimators are updated, and this cycle continues. The communication may decrease since fewer changes in Level-1 samples need to be communicated to all the nodes. For a buffer size B, batch processing leads to a communication of $O(k(\log(m/B))(\log \Delta))$ for the maintenance of a single estimator. We omit further details.

Trade-off Between Communication Cost and Memory. We now describe an algorithm that reduces the communication relative to Algorithm 5.1. at the expense of increasing memory consumption. As before, each site maintains a random Level-1 edge, f_1, and whenever f_1 changes locally, the change is communicated to all other sites. The Level-2 edges are handled differently, however. Once f_1 is chosen, each site stores the entire neighborhood of f_1 that appears in the local stream after f_1. A random Level-2 edge f_2 is chosen from among all the Level-2 edges that are stored among the various sites at the end of observation. Since the third edge that completes the triangle with f_1 and f_2 is also a neighbor of f_1, it is also stored, and it is possible to check if this triangle is complete. The algorithm description and details of analysis are presented in Appendix D.

Theorem 5.3 *There is an algorithm that observes a graph G as a stream distributed among k sites and computes a* (ε, δ) *approximation of number of triangles. The communication cost of this algorithm is* $O(\psi k \log m)$ *and the total memory used across all sites is* $O(\Delta\psi)$*.*

Improved Algorithm Using Knowledge About Stream Sizes. In some cases, a site may have prior knowledge about the size of the local stream. Such information can be used to further reduce communication, even if only a lower bound on the size is known B^i such that $B^i \leq |\mathcal{S}_i| \leq d \cdot |\mathcal{S}_i|$ for some constant d. For example, if the edges are partitioned in a round-robin manner, or when each edge is sent to a random processor, the different sites get an approximately equal number of edges, and when used in conjunction with knowledge about the overall stream size, we have a bound on the local stream size. We show that for such scenarios, we can reduce the total communication by a factor of $\log m$.

We will first describe the idea of an algorithm for the case when $B^i = |\mathcal{S}_i|$, i.e. the site knows the exact local stream size. Before any stream element is observed, each site i uniformly at random picks an index I_i from $\{1, \cdots B_i\}$, and sends I_i and B^i to the coordinator. The coordinator sets index I to I_i with probability $\frac{B_i}{\sum B_i}$. Suppose $I = I_j$, then the coordinator designates site j as "special site" and the special site will be pick the I-th edge of its local stream as the Level-1 edge and communicates with all other sites. Note that the total amount of communication needed to pick the Level-1 edge is now $O(k)$ as opposed to $O(k \log m)$ in previous algorithms, leading to a $\log m$ factor improvement. Our algorithm described in Appendix D handles the general case where each stream processor knows only a bound on its stream size.

Theorem 5.4 *Fix a constant d. Suppose that the edges of a graph* $G = (V, E)$ *are distributed arbitrarily across k sites, and suppose that each site knows a bound* B^i *such that* $B^i \leq |\mathcal{S}_i| \leq d.B^i$*. There is a distributed streaming algorithm that processes the graph G and returns* (ε, δ) *estimator of* $\tau(G)$*, with message complexity* $O(\psi k \log \Delta)$ *and the memory used per site is* $O(\psi)$*.*

6 Implementation and Experiments

We implemented the parallel algorithm described in Section 4 and investigated its performance on real-world datasets.

Implementation. Our implementation closely follows the description in Section 4. The `sort` primitive implements a PCO sample sort algorithm [5, 24], which offers good speedups. The multisearch routines implement a modified Blelloch et al.'s `merge` algorithm, which stops recursing early when the number of "queries" is small. The main triangle counting logic has about 600 lines of Cilk code, a dialect of C/C++ that supports fork-join parallelism with simple annotations.

Experimental Setup. We designed the experiments to study the following important metrics: **(1)** *Solution's Quality:* the algorithm should deliver accurate solutions; **(2)** *Parallel Speedup:* the parallel algorithm should achieve good speedups[4], indicating that the algorithm can successfully take advantage of parallelism; and **(3)** *Parallel Overhead:* the parallel algorithm running on a single core should not take much longer than its sequential counterpart, showing empirically that it is "cost efficient".

Environment. We performed experiments on a 12-core (with hyperthreading) Intel machine, which has *two* 2.67 Ghz 6-core Intel Xeon X5650 processors with 96GB of memory. It is running Linux 2.6.32-279 (CentOS 6.3). The experiments were programmed to use no more than 8GB of memory. This is to exercise the algorithm when it is resource-constrained. All programs were compiled with GNU `g++` version 4.8.0 20130109 using the flag `-O3`. This version of `g++` has the Intel Cilk runtime, which implements a work-stealing scheduler.

Datasets. Our study uses a collection of graphs, obtained from the SNAP project at Stanford [19] and a recent Twitter dataset [17]. We present a summary of these datasets in Table 1.

Dataset	n	m	Δ	τ	$m\Delta/\tau$	Size
Amazon	334,863	925,872	1,098	667,129	1,523.85	13M
DBLP	317,080	1,049,866	686	2,224,385	323.78	14M
LiveJournal	3,997,962	34,681,189	29,630	177,820,130	5,778.89	0.5G
Orkut	3,072,441	117,185,083	66,626	627,584,181	12,440.68	1.7G
Twitter-2010	41,652,230	1,202,513,046	2,997,487	34,824,916,864	103,503.97	20G
Friendster	65,608,366	1,806,067,135	5,214	4,173,724,142	2,256.22	31G
Powerlaw (synthetic)	267,266,082	9,326,785,184	6,366,528	-	-	167GB

Table 1: A summary of the datasets used in our experiments, showing for every dataset, the number of nodes (n), the number of edges (m), the maximum degree (Δ), the number of triangles in the graph (τ), the ratio $m\Delta/\tau$, and size on disk (stored as a list of edges in plain text).

For most datasets, the exact triangle count is provided by the source (which we verified); in other cases, we compute the exact count using an algorithm developed as part of the Problem-Based Benchmark Suite [24]. We also report the size on disk of these datasets as a list of edges in plain text[5]. In addition, we include one synthetic power-law graph; on this graph, we cannot get the true count, but it is added to speed test the algorithm.

Baseline. We directly compare our results with the true count to assess the accuracy. We also study the overhead of the parallel algorithm by comparing it to the sequential algorithm from our recent work (in submission) [22]. The sequential algorithm uses the same neighborhood-sampling-based estimators as the parallel version but with a sequential bulk-update algorithm that appears inherently sequential. We do not compare the accuracy between the two algorithms because by design, they produce the exact same answer given the same sequence of random bits.

6.1 Results

We perform experiments on graphs with varying sizes and densities. Our algorithm is randomized and may behave differently on different runs. For robustness, we perform *five* trials—except when running the biggest datasets on a single core, where only two trials are used. Table 2 shows for different numbers of estimators $r = 200\text{K}, 2\text{M}, 20\text{M}$, the accuracy, reported as the mean deviation value, and processing times (excluding I/O) using 1 and all 12 cores (24 threads via hyperthreading), as well as the speedup ratio. Mean deviation is a well-accepted measure of error, which, we believe, accurately depicts how well the algorithm performs. In addition, it reports the median I/O time[6]

[4]This measures how much faster it is running on many cores than running sequentially.

[5]While storing graphs in, for example, the compressed sparse-row (CSR) format can result in a smaller footprint, this set of experiments focuses on the settings where we do not have the luxury of preprocessing the graph. Arguably, in such settings, listing edges in plain text is he universal format for how graphs are made available.

[6]Like in the (streaming) model, the update routine to our algorithm takes in a batch of edges, represented as an array of a pair of `int`'s. We note that the I/O reported is based on an optimized I/O routine, in place of the `fstream`'s `cin`-like implementation or `scanf`.

Dataset	r = 200K				r = 2M				r = 20M				I/O
	MD	T_1	T_{12H}	$\frac{T_1}{T_{12H}}$	MD	T_1	T_{12H}	$\frac{T_1}{T_{12H}}$	MD	T_1	T_{12H}	$\frac{T_1}{T_{12H}}$	
Amazon	2.38	1.07	0.14	7.64	0.47	3.58	0.42	8.52	0.11	29.10	3.14	9.27	0.14
DBLP	1.47	1.20	0.16	7.50	0.43	3.70	0.42	8.81	0.09	29.30	3.17	9.24	0.14
LiveJournal	6.84	33.70	3.35	10.06	0.47	40.90	3.95	10.35	0.40	69.70	6.62	10.53	1.54
Orkut	1.93	114.00	11.00	10.36	0.20	133.00	12.30	10.81	0.65	170.00	15.40	11.04	5.00
Twitter-2010	4.63	1140.00	114.00	10.00	2.57	1290.00	120.00	10.75	3.86	1560.00	150.00	10.40	61.00
Friendster	16.74	1690.00	192.00	8.80	8.41	2000.00	201.00	9.95	3.41	2350.00	209.00	11.24	104.00
Powerlaw (synthetic)	–	–	–	–	–	–	–	–	–	–	1050.0	–	970.00

Table 2: The accuracy (MD is mean deviation, **in percentage**), median processing time on 1 core T_1 (**in seconds**), median processing time on 12 cores T_{12H} with hyperthreading (**in seconds**), and I/O time (**in seconds**) of our parallel algorithm across five runs as the number of estimators r is varied.

Several trends are evident from this experiment. *First, the algorithm is accurate with only a modest number of estimators.* In all datasets, including the one with more than a billion edges, the algorithm achieves less than 4% mean deviation using about 20 million estimators, and for smaller datasets, it can obtain better than 5% mean deviation using fewer estimators. Indeed, the accuracy, in general, improves with the number of estimators, consistent with the theoretical findings. Furthermore, in practice, far fewer estimators than suggested by the pessimistic theoretical bound is necessary to reach a desired accuracy. For example, on Twitter-2010, which has the highest $m\Delta/\tau$ ratio among the datasets, using $\varepsilon = 0.0386$, the expression $96/\varepsilon^2 \cdot m\Delta/\tau$ (see Theorem 3.3) is at least 6.6 billion, but we reach this accuracy using 20 million estimators.

Second, the algorithm shows substantial speedups on all datasets. On all datasets, the experiments show that the algorithm achieves up to 11.24x speedup on 12 cores, with the speedup numbers ranging between 7.5x and 11.24x. On the biggest datasets using $r = 20$M estimators, the speedups are consistently above 10x. Furthermore, the time to run the algorithm sequentially (T_1) is significantly more than the I/O time, showing that I/O is not a bottleneck and that parallelism speedup does pay off. Additionally, we experimented with a big synthetic graph (167GB power-law graph) to get a sense of the running time. For this dataset, we were unable to calculate the true count; we also cut short the sequential experiment after a few hours. But this dataset has 5x more edges than Friendster, and our algorithm running on 12 cores finishes in 1050 seconds (excluding I/O)—about 5x longer than the Friendster dataset.

Third, the overhead is well-controlled. In a different experiment, presented in Table 3, we compare our parallel implementation with an implementation of our previous algorithm (sequential) [22]; both algorithms, at the core, maintain the same neighborhood sampling invariant but differ significantly in how the edges are processed. The theoretical bounds are not directly comparable: using $s = \Theta(r)$, the sequential algorithm takes $O(1)$ time on average per edge (assuming large enough graph) whereas the parallel algorithm incurs $O(\frac{1}{B}\log_{M/B}(1 + s/B))$ I/O cost on average. As is apparent from Table 3, for large datasets requiring more estimators, the overhead is less than 1.5x with $r = 20$M. For smaller datasets, the overhead is less than 1.6x with $r = 2$M. In all cases, the amount of parallel speedup gained outweighs the overhead.

We also examine the breakdown of time for different components of the algorithm, in efforts to understand how the speedup is gained and how it is likely to scale. This is presented in Appendix E, but in summary, `sort` dominates the running time (up to 95%) and any improvement to it will improve our algorithm.

Acknowledgments. This research was in part sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under the Social Media in Strategic Communication (SMISC) program, Agreement Number W911NF-12-C-0028. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Defense Advanced Research Projects Agency or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

Dataset	r = 200K			r = 2M			r = 20M		
	T_{seq}	T_1	T_1/T_{seq}	T_{seq}	T_1	T_1/T_{seq}	T_{seq}	T_1	T_1/T_{seq}
Amazon	0.93	1.07	1.15	5.01	3.58	0.71	34.10	29.10	0.85
DBLP	0.91	1.20	1.32	5.01	3.70	0.74	33.50	29.30	0.87
LiveJournal	13.80	33.70	2.44	26.90	40.90	1.52	81.30	69.70	0.86
Orkut	43.40	114.00	2.63	72.50	133.00	1.83	127.00	170.00	1.34
Twitter-2010	393.00	1140.00	2.90	569.00	1290.00	2.27	1180.00	1560.00	1.32
Friendster	866.00	1690.00	1.95	1280.00	2000.00	1.56	1890.00	2350.00	1.24

Table 3: The median processing time of the sequential algorithm T_{seq} (**in seconds**), the median processing time of the parallel algorithm running on 1 core T_1 (**in seconds**), and the overhead factor (i.e., T_1/T_{seq}).

References

[1] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 623–632, 2002. 1, 2

[2] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proc. ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 16–24, 2008. 1, 2

[3] J. W. Berry, L. Fosvedt, D. Nordman, C. A. Phillips, and A. G. Wilson. Listing triangles in expected linear time on power law graphs with exponent at least 7/3. Technical report, Sandia National Laboratories, 2011. 2

[4] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *SPAA'11*, pages 355–366, New York, NY, USA, 2011. ACM. 3, 14

[5] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *SPAA'10*, pages 189–199, New York, NY, USA, 2010. ACM. 3, 8

[6] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, and Christian Sohler. Estimating clustering indexes in data streams. In *Proc. European Symposium on Algorithms (ESA)*, pages 618–632, 2007. 1, 2

[7] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14:210–223, 1985. 2

[8] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Knowledge Data and Discovery (KDD)*, pages 672–680, 2011. 2

[9] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11:29–41, 2009. 1, 2

[10] Jean-Pierre Eckmann and Elisha Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the National Academy of Sciences*, 99(9):5825–5829, 2002. 1

[11] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999. 3, 14

[12] Madhav Jha, C. Seshadhri, and Ali Pinar. From the birthday paradox to a practical sublinear space streaming algorithm for triangle counting. *CoRR*, abs/1212.2264, 2012. 1, 2

[13] Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In *Proc. 11th Annual International Conference Computing and Combinatorics (COCOON)*, pages 710–716, 2005. 1, 2, 7

[14] Daniel M. Kane, Kurt Mehlhorn, Thomas Sauerwald, and He Sun. Counting arbitrary subgraphs in data streams. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 598–609, 2012. 1, 2, 7

[15] Mihail N. Kolountzakis, Gary L. Miller, Richard Peng, and Charalampos E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. In *WAW*, pages 15–24, 2010. 2

[16] Konstantin Kutzkov and Rasmus Pagh. On the streaming complexity of computing local clustering coefficients. In *Proceedings of 6th ACM conference on Web Search and Data Mining (WSDM)*, 2013. 2

[17] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010. 9

[18] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407:458–473, 2008. 2

[19] Jure Leskovec. Stanford large network dataset collection. `http://snap.stanford.edu/data/index.html`. Accessed Dec 5, 2012. 9

[20] Madhusudan Manjunath, Kurt Mehlhorn, Konstantinos Panagiotou, and He Sun. Approximate counting of cycles in streams. In *Proc. European Symposium on Algorithms (ESA)*, pages 677–688, 2011. 1, 2, 7

[21] M. E. J. Newman. The structure and function of complex networks. *SIAM REVIEW*, 45:167–256, 2003. 1

[22] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. Technical Report RC25339, IBM Research, December 2012. 1, 2, 3, 4, 9, 10, 12

[23] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Workshop on Experimental and Efficient Algorithms (WEA)*, pages 606–609, 2005. 2

[24] 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 *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012. 8, 9

[25] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proc. 20th International Conference on World Wide Web (WWW)*, pages 607–614, 2011. 1, 2

[26] Charalampos E. Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Netw. Analys. Mining*, 1(2):75–81, 2011. 1

[27] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 837–846, 2009. 2

[28] S. Wasserman and K. Faust. *Social Network Analysis*. Cambridge University Press, 1994. 1

A Useful Bounds

We refer the following standard measure-concentration bounds in our proofs; we state them here for reference.

Theorem A.1 (Chebyshev's Inequality) *Let $\lambda > 0$, X be a random variable with $\mathbf{E}[X] < \infty$, and $0 < \sigma^2 < \infty$ be the variance of X. Then,*

$$\mathbf{Pr}\Big[|X - \mathbf{E}[X]| \geq \lambda\sigma\Big] \leq 1/\lambda^2.$$

Theorem A.2 (Chernoff's Bounds) *Let $\lambda > 0$ and $X = X_1, \ldots, X_n$, where each X_i, $i = 1, \ldots, n$, is independently distributed in $[0, 1]$. Then,*

$$\mathbf{Pr}\Big[X \geq (1+\lambda)\,\mathbf{E}[X]\Big] \leq e^{-\frac{\lambda^2}{2+\lambda}\cdot\mathbf{E}[X]} \quad \textit{and} \quad \mathbf{Pr}\Big[X \geq (1-\lambda)\,\mathbf{E}[X]\Big] \leq e^{-\frac{\lambda^2}{2}\cdot\mathbf{E}[X]}.$$

B Neighborhood Sampling: Deferred Proofs

We begin by computing the probability that a particular triangle in the graph is sampled by the invariant. In particular, we reason about the probability that the edges f_1, f_2, f_3 coincide with a particular triangle in the graph G. We reproved this lemma, originally appeared in [22], using the present paper's terminology.

Lemma B.1 (Discovery Probability) *Let $\mathcal{S} = (V, E, \leq_{\mathcal{S}})$ denote a simple, undirected graph $G = (V, E)$, together with a total order $\leq_{\mathcal{S}}$ on E. Let $t^* \in \mathcal{T}(G)$ be any triangle in G. If (f_1, f_2, f_3, δ) satisfies NBSI, then the probability that $\{f_1, f_2, f_3\}$ represents the triangle t^* is is*

$$\mathbf{Pr}[\{f_1, f_2, f_3\} = t^*] = \frac{1}{|E| \cdot C_{\mathcal{S}}(t^*)}$$

where we recall that $C(t^*) = |\Gamma_{\mathcal{S}}(f)|$ if f is the t^*'s first edge in the $\leq_{\mathcal{S}}$ ordering.

Proof: Let $t^* = \{e_1, e_2, e_3\} \in \mathcal{T}(G)$, where $e_1 < e_2 < e_3$ without loss of generality. Further, let $\mathcal{E}_1$ be the event that $f_1 = e_1$, and $\mathcal{E}_2$ be the event that $f_2 = e_2$. It is easy to check that $f_3 = e_3$ if and only if both $\mathcal{E}_1$ and $\mathcal{E}_2$ hold. By NBSI, we have that $\mathbf{Pr}[\mathcal{E}_1] = \frac{1}{|E|}$ and $\mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] = \frac{1}{|\Gamma_{\mathcal{S}}(e_1)|} = \frac{1}{C_{\mathcal{S}}(t^*)}$. Thus, $\mathbf{Pr}[t = t^*] = \mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] = \frac{1}{|E| \cdot C_{\mathcal{S}}(t^*)}$, concluding the proof. ■

An unbiased estimator can be constructed from NBSI even though the triangle it maintains may be non-uniform. The idea is to output a value which counterbiases the probability so that in expectation, the contribution of a triangle is exactly 1. This is easy to achieve because we know how much it is biased by.

Proof of Lemma 3.2: If $f_3 = \emptyset$, then $X = 0$. Otherwise, we know that $\{f_1, f_2, f_3\}$ is a triangle $t^* \in \mathcal{T}(G)$ and this particular triangle is sampled with probability $\mathbf{Pr}[\{f_1, f_2, f_3\} = t^*] = \frac{1}{|E| \cdot C_{\mathcal{S}}(t^*)}$ by Lemma B.1. When this happens, $X = |E| \cdot \delta = |E| C_{\mathcal{S}}(t^*)$ because $\delta = C_{\mathcal{S}}(t^*)$ by definition. Hence,

$$\mathbf{E}[X] = \sum_{t^* \in \mathcal{T}(G)} |E| C(t^*) \cdot \mathbf{Pr}[t = t^*] = \tau(G).$$

■

Even though an estimator in Lemma 3.2 is unbiased, we need many such estimators to have a sharp estimate. We give a proof of Theorem 3.3, showing that we only need about $O(1/\varepsilon^2 \cdot m\Delta(G)/\tau(G) \cdot \log(\frac{1}{\delta}))$ independent estimators to achieve (ε, δ)-approximation using a median-of-means aggregate:

Proof of Theorem 3.3: Let $\alpha = 8/\varepsilon^2 \cdot m\Delta/\tau$ and $\beta = 12 \ln(1/\delta)$ Fix $r = \alpha\beta$. Let $X_j^{(i)}$, $i = 1, \ldots, \beta$, $j = 1, \ldots, \alpha$, be independent unbiased estimators in Lemma 3.2. Also, let $Y^{(i)}$, $i = 1, \ldots, \beta$, be the average of $X_1^{(i)}, X_2^{(i)}, \ldots, X_\alpha^{(i)}$. Each estimator has variance at most $2m\tau\Delta$ because δ is upper-bounded by the number of edges incident on f_1, of which there can be at most 2Δ. Therefore, the variance of each $Y^{(i)}$ is at most $2m\tau\Delta/\alpha = \frac{\varepsilon^2}{4} \cdot \tau^2$. So then, by Chebyshev's inequality, we have $\mathbf{Pr}\Big[|Y^{(i)} - \mathbf{E}\Big[Y^{(i)}\Big]| > \varepsilon \cdot \tau(G)\Big] \leq \frac{1}{4}$.

To boost the success probability up to $1 - \delta$, we take the median of these $Y^{(i)}$'s. The median estimator fails to produce an ε-approximation only if more than $\beta/2$ fails to produce an ε-approximation. In expectation, the number of "failed" estimators is at most $\beta/4$. Therefore, by a standard Chernoff bound (Theorem A.2), we fail with probability at most $\mathbf{Pr}[\mathsf{FAILED}] \leq e^{-\frac{1^2(\beta/4)}{3}} = \delta$, proving that our final estimate is an (ε, δ)-approximation using a total of at most $r = \alpha\beta$ estimators. ■

C The Parallel Cache-Oblivious Model

Parallel algorithms in this work are expressed in the nested parallel model. It allows arbitrary dynamic nesting of parallel loops and fork-join constructs but no other synchronizations, corresponding to the class of algorithms with series-parallel dependency graphs. In this model, computations can be recursively decomposed into tasks, parallel blocks, and strands, where the top-level computation is always a task:

- The smallest unit is a **strand** `s`, a serial sequence of instructions not containing any parallel constructs or subtasks.
- A **task** `t` is formed by serially composing $k \geq 1$ strands interleaved with $k - 1$ parallel blocks, denoted by $\mathtt{t} = \mathtt{s}_1; \mathtt{b}_1; \ldots; \mathtt{s}_k$.
- A **parallel block** `b` is formed by composing in parallel one or more tasks with a fork point before all of them and a join point after, denoted by $\mathtt{b} = \mathtt{t}_1 \| \mathtt{t}_2 \| \ldots \| \mathtt{t}_k$. A parallel block can be, for example, a parallel loop or some constant number of recursive calls.

The **depth** (aka. **span**) of a computation is the length of the longest path in the dependence graph.

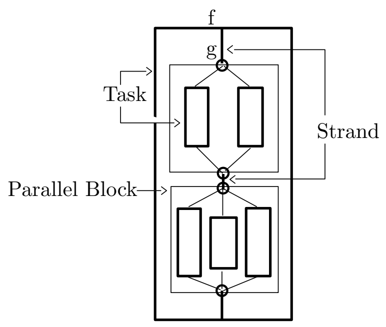

Figure 1: Decomposing the computation: tasks, strands and parallel blocks

Task $\mathtt{t}$ forks subtasks $\mathtt{t}_1$ and $\mathtt{t}_2$,
with $\kappa = \{\ell_1, \ell_2, \ell_3\}$

$\mathtt{t}_1$ accesses ℓ_1, ℓ_4, ℓ_5 incurring 2 misses
$\mathtt{t}_2$ accesses ℓ_2, ℓ_4, ℓ_6 incurring 2 misses

At the join point: $\kappa' = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6\}$

Figure 2: Applying the PCO model (Definition C.1) to a parallel block. Here, $Q^*(\mathtt{t}; M, B; \kappa) = 4$.

Measuring Memory Access Costs. We will analyze memory-access cost of parallel algorithms in the Parallel Cache Oblvivious (PCO) model [4], a parallel variant of the cache oblivious (CO) model. The Cache Oblivious (CO) model [11] is a model for measuring cache misses of an algorithm when run on a single processor machine with a two-level memory hierarchy—one level of finite cache and unbounded memory. The cache complexity measure of an algorithm under this model $Q(n; M, B)$ counts the number of cache misses incurred by a problem instance of size n when run on a fully associative cache of size M and line size B using the optimal (offline) cache replacement policy.

Like the CO model, the **Parallel Cache-Oblivious (PCO) model** assumes a memory of unbounded size and a single cache with size M and line size B (in words) using optimal replacement policy[7]. Extending the CO model, the PCO model gives a way to analyze the number of cache misses for the tasks that run in parallel in a parallel block. The PCO model approaches it by (i) ignoring any data reuse among the parallel subtasks and (ii) assuming the cache is flushed at each fork and join point of any task that does not fit within the cache.

More precisely, let $loc(\mathtt{t}; B)$ denote the set of distinct cache lines accessed by task $\mathtt{t}$, and $S(\mathtt{t}; B) = |loc(\mathtt{t}; B)| \cdot B$ denote its size. Also, let $s(\mathtt{t}; B) = |loc(\mathtt{t}; B)|$ denote the size in terms of number of cache lines. Let $Q(\mathtt{c}; M, B; \kappa)$ be the cache complexity of $\mathtt{c}$ in the sequential CO model when starting with cache state κ.

Definition C.1 (Parallel Cache-Oblivious Model) *For cache parameters M and B the **cache complexity** of a strand, parallel block, and a task starting in cache state κ are defined recursively as follows (see [4] for detail).*

- *For a strand, $Q^*(\mathtt{s}; M, B; \kappa) = Q(\mathtt{s}; M, B; \kappa)$.*
- *For a parallel block $\mathtt{b} = \mathtt{t}_1 \| \mathtt{t}_2 \| \ldots \| \mathtt{t}_k$, $Q^*(\mathtt{b}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathtt{t}_i; M, B; \kappa)$.*
- *For a task $\mathtt{t} = \mathtt{c}_1; \ldots; \mathtt{c}_k$, $Q^*(\mathtt{t}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathtt{c}_i; M, B; \kappa_{i-1})$, where $\kappa_i = \emptyset$ if $S(\mathtt{t}; B) > M$, and $\kappa_i = \kappa \bigcup_{j=1}^{i} loc(\mathtt{c}_j; B)$ if $S(\mathtt{t}; B) \le M$.*

We use $Q^*(\mathtt{c}; M, B)$ to denote a computation $\mathtt{c}$ starting with an empty cache and overloading notation, we write $Q^*(n; M, B)$ when n is a parameter of the computation. We note that $Q^*(\mathtt{c}; M, B) \ge Q(\mathtt{c}; M, B)$. That is, the PCO gives cache complexity costs that are always at least as large as the CO model. Therefore, any upper bound on the PCO is an upper bound on the CO model. Finally, when applied to a parallel machine, Q^* is a "work-like" measure and represents the total number of cache misses across all processors. An appropriate scheduler is used to evenly balance them across the processors.

D Proofs from Section 5

Proof of Lemma 5.1.: Let M_1 be the total number of messages sent from the nodes to the coordinator by Algorithm 5.1 across all nodes, due to the first level sampling, i.e. the condition $w_1(e) < \ell_1$. Let M_2 denote the total number of messages sent from the nodes to the coordinator due to the level two sampling, i.e. the condition

[7]That is, accessing a non-resident cache line ℓ in a full cache will result in ℓ replacing the line in the current cache furthest accessed in the future. The (offline) optimal algorithm knows the future.

$w_2(e) < \ell_2$. The total number of messages sent is $M = (k+1)(M_1 + M_2)$, since each message from a site to the coordinator is relayed to all the sites in the system.

We first consider M_1. Consider an edge e_i that is observed by a site. Note that the value of ℓ_1 at the site is equal to $\min_{j \le i} w(e_j)$, i.e. the globally smallest weight from among all edges so far. Thus the probability that $w(e_i) < \ell_1$ is equal to $1/i$, and the probability that a message is sent to the coordinator due to a change in the first level sample is $1/i$.

$$\mathbf{E}[M_1] = \sum_{i=1}^{m} \frac{1}{i} = O(\log m)$$

We next consider M_2. For each edge e selected as the sample at the first level, there are at most 2Δ edges that arrive after e (as per the total order $\le_{\mathcal{S}}$). For each edge e_i, let $M_2(e_i)$ be defined as follows. $M_2(e_i)$ is 0 if e_i was not selected as a first level sampled edge. Otherwise, $M_2(e_i)$ equals the number of instances when a neighboring edge e' of e_i arrives with a sequence number greater that i, and e' causes a change in the second level sample maintained for e_i. Clearly, $M_2 = \sum_{e \in E} M_2(e)$. The expected value of $M_2(e_i)$ is bounded by:

$$\mathbf{E}[M_2(e_i)] \le \frac{1}{i} \sum_{j=1}^{\Delta} \frac{1}{j} = O\left(\frac{\log \Delta}{i}\right)$$

We have:

$$\mathbf{E}[M_2] = \sum_{i=1}^{m} \mathbf{E}[M_2(e_i)] = O((\log \Delta)(\log m))$$

The expected total number of messages is thus $(k+1)(M_1 + M_2) = O(k(\log \Delta)(\log m))$. ■

Lemma D.1 *Suppose graph $G = (V, E)$, whose edges were distributed arbitrarily across the k sites was processed by the Algorithm 5.1. Let X denote the estimator returned by the coordinator. Then $\mathbf{E}[X] = \tau(G)$.*

Proof of Lemma D.1. : We will use superscript i to denote the value of a local variable at site i. For example, ℓ_1^i denotes the value of ℓ_1 local to site i. At the beginning of all observations, we have that $g = \ell_1^i$, $1 \le i \le k$. When the the values of ℓ_1^i changes at site i, this information is instantaneously relayed to the coordinator and so the coordinator sets g_1 to the new value of ℓ_1^i and communicates this to all sites. Each site in turn changes its own value of ℓ_1 to g_1. Thus at any time the values of all ℓ_1^i, $1 \le i \le, k$, are the same across among all the sites. Similarly, the values of ℓ_2^i, f_1^i, and f_2^i are the same across all the sites.

Let f_1^i and f_2^i be the values of f_1 and f_2 local to Site i at the end of all observations. Since these values are same across among all sites let us denote them with f_1 and f_2. Note that ,

$$\forall i, w_1(f_1) = w_1(f_1^i) = \min_{e \in \mathcal{S}_i} w_1(e),$$

and

$$\forall i, w_2(f_2) = w_2(f_2^i) = \min_{e \in \Gamma_{\mathcal{S}_i}(f_i^1)} w_2(e),$$

it follows that f_1 is a randomly chosen of E and f_2 is a randomly chosen edge of $\Gamma_{\mathcal{S}}(f_1)$. Since each c^i is $|\Gamma_{\mathcal{S}_i}(f_1)|$ we have that $c = \sum_1^k c^i = |\Gamma_{\mathcal{S}}(f_1)|$.

Note that f_1 and f_2 belong to every t^i. If at least one of t^i is a triangle, then set f_3 be the third edge of one (any) such triangle. So none of t^i's is a triangle, then set f_3 to ϕ. Now it follows that (f_1, f_2, f_3, c) satisfies Invariant NBSI 3.1. Thus by Lemma 3.2 the output of the coordinator is an unbiased estimator of $\tau(G)$, and so $\mathbf{E}[X] = \tau(G)$.

■

Proof of Theorem 5.2: Each site runs $r = \psi$ independent copies of the basic algorithm, and the coordinator will compute r unbiased estimators. These estimates can be aggregated using standard techniques, as in Theorem 3.3. This yields a (ε, δ) approximation of $\tau(G)$. Since the memory user per instance per site is $O(1)$ words, the space bound follows. The bound on communication follows from Lemma 5.1. ■

Trade-off Between Communication Cost and Memory. We now present details of the algorithm which presents a trade-off between communication and memory. The sites and the coordinator perform the same initialization as in Algorithm 5.1, except there is an additional state variable at each site i, $\mathbb{N}_i$, which is the set of all neighbors of the level 1 sampled edge f_1 that appear in $\mathcal{S}_i$ after f_1.

Algorithm 1: When site i observes edge (u, v) in local stream $\mathcal{S}_i$:

```
Let $w(e)$ be a random real number in $[0, 1]$.
if $w(e) < \ell_1$ then
    $f_1 = e$; $\mathbb{N}_i(f_1) = \phi$; $\ell_1 = w_1(e)$;
    Send $\langle e, w_1(e) \rangle$ to Coordinator;
else
    if $e$ is adjacent to $f_1$ then
        $\mathbb{N}_i(f_1) = \mathbb{N}_i(f_1) \cup \{e\}$
```

When site i receives $\langle e, w \rangle$ from coordinator, it sets $f_1 = e$, $\ell_1 = w$, and $\mathbb{N}(f_1) = \phi$. When the coordinator receives $\langle e, w \rangle$ from a site, it compares w with g. If $w < g$, then it sets $g = w$ and transmits $\langle e, w \rangle$ to all sites.

At the end of all observations, the sites and the coordinator communicate as follows. Each site i randomly picks f_2^i from $N_i(f_1)$ and sends f_2^i and $|N_i(f_1)|$ to the coordinator. The coordinator upon receiving f_2^i and $|\mathbb{N}_i(f_1)|$, $1 \leq i \leq k$, sets f_2 to f_2^i with probability $|N_i(f_1)|/c$ where $c = \sum_1^k |N_i(f_1)|$. The coordinator send f_2 to all sites. Each site i checks if a triangle can be formed with f_1, f_2 and an edge from $N_i(f_1)$. If such a triangle is found, then the site sends a bit 1 to the coordinator, else it sends bit 0 to the coordinator. If the coordinator receives 1 from at least one site, then it outputs $c \cdot m/2$, where m is the number of all edges seen across all sites. Otherwise the coordinator outputs 0.

We will first establish a bound on the number of messages exchanged by the above algorithm.

Lemma D.2 *The expected number of works exchanged by the above algorithm is $O(k \log m)$.*

Proof of D.2. : Let M be a random variable that denotes the number of words exchanged while executing the Algorithm 1. By following the same argument as in Lemma 5.1, it follows that $E[M] = O(k \log m)$. Note that, at the end of all observations, each site exchanges a constant number of words with the coordinator, Thus the expected number of messages exchanged by the algorithm is $O(klogm)$. ■

Lemma D.3 *Let X be the random variable that denotes the output of the coordinator after processing a graph $G = (V, E)$ whose edges are distributed among k sites. Then $\mathbf{E}[X] = \tau(G)$.*

Proof of Lemma D.3. : Let us use f_1^i to denote the local value of f_1 at site i. Observe that the values of f_1^i is the same across all the sites. Let us denote this with f_1. Since each f_2^i is a randomly chosen edge among $\Gamma_{\mathcal{S}_i}(f_1)$ and f_2 is set to f_2^i with probability $\frac{|\Gamma_{\mathcal{S}_i}(f_1)|}{\sum |\Gamma_{\mathcal{S}_i}(f_1)|}$, it follows that f_2 is a randomly chosen edge from $\cup_1^k \Gamma_{\mathcal{S}_i}(f_1)$. By appealing to the total order $\leq_{\mathcal{S}}$ and arguing as in the proof of Lemma D.1, we have that f_2 is a uniformly chosen edge from $\Gamma_{\mathcal{S}}(f_1)$.

Let $t^* = \{u_1, u_2, u_3\}$ be a triangle of the graph so that u_1 is the first edge as per the order $\leq_{\mathcal{S}}$. Let $c(t^*)$ denote the cardinality of $\Gamma_{\mathcal{S}}(u_1)$. We consider the probability that a site detects the triangle t^*. Since the graph stream does not have duplicate edges, exactly one site can detect the triangle t^*. This happens if the following disjoint events happen:

$$\mathcal{E}_1 : (f_1 = u_1) \text{ and } (f_2 = u_2),$$

$$\mathcal{E}_2 : (f_1 = u_1) \text{ and } (f_2 = u_3).$$

Since the events are disjoint, the probability that a site detects t^* is $\Pr[\mathcal{E}_1] + \Pr[\mathcal{E}_2]$. Since f_1 is uniformly chosen edge, probability that f_1 equals u_1 is $1/m$. Since f_2 is a random edge from $\Gamma_{\mathcal{S}}(u_1)$, the probability that f_2 equals u_2 (conditioned on the vent $f_1 = u_1$) is $1/c(t^*)$. Thus $\Pr[\mathcal{E}_1] = 1/mc(u_1)$, similarly $\Pr[\mathcal{E}_2] = 1/mc(t^*)$. Thus the probability that t^* is detected by a site is $2/mc(t^*)$. When this happens, the coordinator outputs $mc(t^*)/2$. Note that the coordinator outputs a non-zero value, when some detects a triangle. Thus

$$\mathbf{E}[X] = \sum_{t^* \in \mathcal{T}(G)} mc(t^*)/2 \Pr[\text{ a site detects } t^*] = \tau(G).$$

■

Proof of Theorem 5.3: Each site runs ψ copies of the basic algorithm, and the coordinator aggregates all the estimators. Since each instance of the basic algorithm causes a communication of $O(k \log m)$ words the communication bound follows. The memory cost per instance at site i is $|\Gamma_{\mathcal{S}_i}(f_1)|$. Since Δ is the maximum degree the total number of edges stored for each instance is no more than 2Δ, leading to the desired space bound. ■

At the beginning the sites and the coordinator communicate as per Algorithm 2.

Algorithm 2: Initialization: Site knows the bound B^i

```
At site $i$: Set $f_1^i, f_2^i$ to $\phi$, $c = 0$. $t = \phi$. $\ell_1^i = \ell_2^i = 1$.
Randomly generate weights $w_1(1), \cdots w_1(B^i)$ all between 0 and 1. Let the smallest weight among them is
$w_1(p)$. Set $\ell_1^i$ to $w_1(p)$. Set $I$ to $p$. Send $\ell_1^i$ to the coordinator.
The coordinators receives $\ell_1^i$, $1 \le i \le k$. Let $s$ be the index for which $\ell_1^s = \min\{\ell_1^1, \cdots, \ell_1^k\}$. Set $g = \ell_1^s$, and
$g_2 = 1$. Transmit $\langle s, g_1 \rangle$ to all sites. Designate site $s$ as "special site".
Each local site $i$ sets $\ell_1^i = g_1$.
```

Each site i while processing its first B^i edges behaves as follows: If site i is a non special site, then it ignores all the local edges till it receives $\langle 1, w, e \rangle$ from the coordinator. Whenever it receives such a message, it sets ℓ_1^1 to w, f_1^i to e, and c^i to 0. For every edge received afterwards, it makes a call to the procedure `Level-2 Sample`.

If site i is a special site, then it ignores all the local edges until either it receives a message $\langle 1, w, e \rangle$ from the coordinator or Ith edge of the local stream arrives. If the event" Ith edge e of the local stream arrives" occurs first, then it sends $\langle 1, \ell_1, e \rangle$ to the coordinator, sets f_1^i to e, sets c^i to 0, and calls the procedure `Level-2 Sample`. If the event "receives message $\langle 1, w, e \rangle$" occurs first, then it sets ℓ_1^i to w, f_1^i to e. For every edge received after one of these two events happen, it makes a call to the procedure `Level-2 Sample`.

Once a site processes the first B_i edges, it behaves as follows: For each edge $e \in \mathcal{S}_i$, it randomly generates a weight $w_1(e)$. If $w_1(e) < \ell_1^i$, it sets ℓ_1^i to $w_1(e)$, f_1^i to e, c^i to 0, and sends $\langle 1, e, w(e) \rangle$ to the coordinator. If $f_1^i \neq \phi$ and $w_1(e) \ge \ell_1^i$, then it makes a call to the procedure `Level-2 Sample`.

Procedure `Level-2 Sample`: Site i upon receiving edge e, checks if e is a neighbor of f_1^i. If so, increments c^i and randomly generates a weight $w_2(e)$. If $w_2(e) < \ell_2^i$, then it sets f_1^i to e, ℓ_2^i to $w(e)$, and sends $\langle 2, w(e), e \rangle$ to the coordinator.

In addition to performing all of the above tasks, for every edge received e, site i sets t^i to $\{f_1, f_2, e\}$. The behavior of the coordinator is exactly as in Algorithm 5.1.

Finally at the end of all observations sends c^i and $|\mathcal{S}_i|$ to the coordinator. If t^i is a triangle, then it sends 1 to the coordinator, otherwise it sends zero. The coordinator compute $c = \sum_{i=1}^k c^i$, and $m = \sum_{i=1}^k |\mathcal{S}_i|$. If the coordinator receives 1 from at least one site, then it outputs mc otherwise it outputs 0.

Lemma D.4 *The expected number of words communicated by the above algorithm is $O(k \log \Delta)$.*

Proof of Lemma D.4. : During the initialization phase the number of messages is $O(k)$. Each site i communicates with the coordinator only when ℓ_1^i or ℓ_2^i changes. In addition the special site s may send a message when it sees $I$the edge in its stream. Let M_1 be the number of messages communicated due to changes in ℓ_1^i and M_2 be the number of messages communicated due to changes in ℓ_2^i.

We will first bound M_1. Observe that any site i, ℓ_i^1 does not change while it processes first B^i edges of the stream $\mathcal{S}_i$. Only the special site may send a message to the coordinator while processing its first B^s edges. Thus the total number messages that contribute to M_1 while the sites process their first B^i, $1 \leq i \leq k$, edges is $O(k)$.

Call an edge $e \in \mathcal{S}_i$ a *first block edge* if it is among the first B^i edges of $\mathcal{S}_i$, otherwise it is a *second block edge*. Note that for each second block edge e, the algorithm generates a weight $w_1(e)$. Order all second block edges as per the realtime at which a weight is generated. I.e, e_j is the jth second block edge for which a weight is generated for jth time. Note that the algorithm generated exactly $B = \sum B^i$ many weights during the initialization phase. Thus edge e_j will cause a ℓ_1^i to change if $w_1(e_j)$ is the minimum among $B + j$ random weights generated. This happens with probability $1/(B + j)$. Thus expected number of times a message is sent while processing second block edges is

$$\sum_{1}^{m-B} \frac{1}{B+i}.$$

Since the total number of edges $m \leq dB$, the above expression evaluates to $\log d$ which is a constant. Thus the expected value of M_1 is $O(k)$. Similar arguments show that the expected value of M_2 is $O(k \log \Delta)$. Thus the expected number of messages is $O(k \log \Delta)$.

■

Lemma D.5 *Let X be a random variable that denotes that output of the coordinator after processing a graph $G = (V, E)$. Then* $\mathbf{E}[X] = \tau(G)$.

Proof: Note that at any time the values of f_1^i, $1 \leq i \leq k$, are all the same. Similarly, the values of f_2^i, $1 \leq i \leq k$, are all the same. Let f_1 and f_2 be edges corresponding to f_1^1 and f_2^1 at the end of all observations. Observe that $w_1(f_1)$ is $\min\{w_1(e) \mid e \in \mathcal{S}\}$. Thus f_1 is a randomly chosen edge of $\mathcal{S}$. Consider the total order $\leq_{\mathcal{S}}$ among edges the is implied by the real time arrival of the edges. We have that $w_2(f_2) = \min\{w_2(e) \mid e \in \Gamma_{\mathcal{S}}\}$. Thus f_2 is a random neighbor of f_1. Note that every t^i contains f_1 and f_2. If any of t^i's is a valid triangle, then set f_3 to the third edge of that triangle, otherwise set f_3 to ϕ. Now, note that $\langle f_1, f_2, f_3, c\rangle$ satisfy the Invariant 3.1. Thus by Lemma 3.2, the output of the coordinator is an unbiased estimator of $\tau(G)$, and so $\mathbf{E}[X] = \tau(G)$. ■

Proof of Theorem 5.4 follows by running ψ copies of the basic estimator and aggregating the answers, and appealing to Chernoff bound.

E Further Thoughts On Experiments

To expand on the I/O cost: the data shows that the algorithm is compute bound on 1 core, suggesting benefits to reap from parallelism. We ran these experiments streaming in data directly from a network-area storage via NFS. Even in this setting, the time to run the algorithm sequentially (T_1) is more than the I/O time, showing that I/O is not a bottleneck. With an input source with a higher throughput, we expect parallelism to be more relevant to match the data arrival speed.

Timing Breakdown. We examine the breakdown of how the time is spent in different components of the algorithm. This study helps us understand how the speedup is gained and how it is likely to scale. Figure 3 shows the fractions of time spent inside `sort`, multisearch routines, and other components for the 3 biggest datasets. As apparent from the algorithm's description, a main building block of our algorithm is the multisearch operation, implemented using sort and "merge-like" routines. The figure shows that the majority of the time is spent on sorting (up to 94%), which has great speedups in our setting. Because we can directly use an off-the-shelf sorting implementation, this portion will scale with the performance of sorting routines, which have been shown to scale well. The merge-like portion makes up of less than 5% of the running time. For this reason, we did not focus much effort into optimizing it

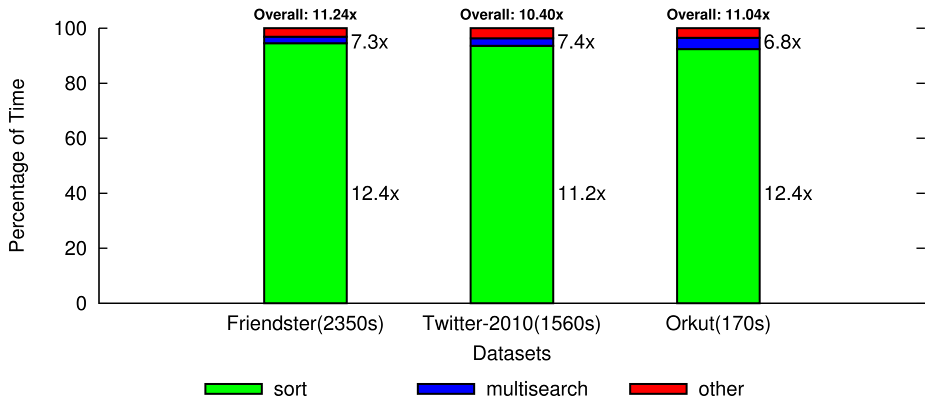

Figure 3: The timing breakdown of the parallel algorithm on the 3 largest datasets running on 1 core vs. running on all 12 cores (24 threads with hyperthreading). The number in parenthesis next to each dataset is the T_1 (**in seconds**) whereas the numbers next to the columns are the speedup ratio for the corresponding components.

since although it does not scale as well, it does not significantly degrade the overall speedup. We suspect a better implementation can improve the performance.