Parallel Triangle Counting in Massive Streaming Graphs

[Extended Abstract] *

Kanat Tangwongsan
IBM Research
ktangwo@us.ibm.com

A. Pavan
Iowa State University
pavan@cs.iastate.edu

Srikanta Tirthapura
Iowa State University
snt@iastate.edu

ABSTRACT

The number of triangles in a graph is a fundamental metric widely used in social network analysis, link classification and recommendation, and more. In these applications, modern graphs of interest tend to both large and dynamic. This paper presents the design and implementation of a fast parallel algorithm for estimating the number of triangles in a massive undirected graph whose edges arrive as a stream. Our algorithm is designed for shared-memory multicore machines and can make efficient use of parallelism and the memory hierarchy. We provide theoretical guarantees on performance and accuracy, and our experiments on real-world datasets show accurate results and substantial speedups compared to an optimized sequential implementation.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Management—*data mining*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*; F.1.2 [**Theory of Computation**]: Modes of Computation—*parallelism and concurrency*

General Terms

Algorithms, Performance

Keywords

triangle counting; streaming algorithm; parallel algorithm; parallel cache oblivious (PCO); massive graphs

1. INTRODUCTION

The number of triangles in a graph is an important metric in social network analysis [30, 21], identifying thematic structures of networks [10], spam and fraud detection [2], link classification and recommendation [29], among others. Driven by these applications and further fueled by the growing volume of graph data, the research community has developed efficient algorithms for counting and approximating the number of triangles in massive graphs. In the past decade, several streaming algorithms have been proposed (e.g., [1, 14, 6, 20, 15, 23, 13]. These algorithms were designed to handle graph evolution and graphs that are too large to fit in memory, but they cannot effectively utilize parallelism beyond the trivial "embarrassingly parallel" implementation, leaving much to be desired in terms of performance. In a similar timeframe, a number of parallel algorithms have been proposed (e.g., [26, 9]). These algorithms were designed to quickly process large volume of static data, but they cannot efficiently handle constantly changing graphs. As such, despite indications that modern graph datasets are both massive and dynamic, *none of the existing algorithms can efficiently handle graph evolution and fully utilize parallelism at the same time.*

In this paper, we describe a fast shared-memory parallel algorithm that combines the benefits of streaming algorithms and parallel algorithms. As is standard, the algorithm provides a randomized relative-error approximation: given $0 < \varepsilon, \delta \leq 1$, a random variable $\hat{X}$ is an (ε, δ)-approximation of the true quantity X if $|\hat{X} - X| \leq \varepsilon X$ with probability at least $1 - \delta$.

1.1 Our Contributions

We present the design and implementation of the *coordinated bulk parallel* algorithm—a fast shared-memory parallel algorithm for approximating the number of triangles in a large graph whose edges arrive as a stream. The algorithm can process the edges of an evolving graph at a high throughput (e.g., several millions of edges/second on a recent 12-core machine), using memory on the order of a few hundreds of MB. It is also useful for processing large static graphs by reaping the benefits of its small memory footprint and effective use of parallelism.

Features of the Algorithm: The coordinated bulk parallel algorithm is designed for multicore machines and has the following features. First, it works in the limited-space streaming model, where only a fraction of the graph can be retained in memory; this allows it to process, in a single pass, graphs much larger than the available memory. Second, it has low overhead with respect to an optimized sequential implementation and scales almost linearly with the number of cores.

Ingredients of the Algorithm: The algorithm owes its performance to two ideas: coordinated bulk processing and cache efficiency. It processes edges in batches, rather than processing each edge individually; this allows multiple edges to be processed in parallel, leading to a higher throughput. Further, in processing a batch of edges, different parallel tasks coordinate with each other through a shared data structure; this results in less redundant computation than an approach where the parallel elements process data independently of each other. Moreover, it takes full advantage of the memory hierarchy. The memory system of a modern machine has unfortu-

*A full version of this paper is available on the e-Print arXiv [27]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.
Copyright 2013 ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505741.

nately become highly sophisticated, consisting of multiple levels of caches and layers of indirection. Navigating this complex system, however, is necessary for a parallel implementation to be efficient. Our algorithm is cache-oblivious [11, 4], which allows it to make efficient use of the memory hierarchy (i.e. minimize cache misses) without knowing the specific memory/cache parameters such as the cache size and the line size.

We show how to process a batch of edges in asymptotically the same cost as sorting the batch using a cache-optimal parallel sort (see Theorem 3.1 for a precise statement). Our algorithm is expressed in terms of simple parallel operations such as sorting, merging, parallel prefix, etc. Importantly, in our scheme, the work is asymptotically the same as that of its best-known sequential counterpart.

We also experimentally evaluate the algorithm. Our implementation yields substantial speedups on massive real-world networks. On a machine with 12 cores, we obtain up to 11.24x speedup when compared with a sequential version, with the speedup ranging from 7.5x to 11.24x. In separate a stress test, a large (synthetic) power-law graph of size 167GB was processed in about 1,000 seconds. In terms of throughput, this translates to millions of edges per second.

1.2 Related Work

Approximate triangle counting is well-studied in both streaming and non-streaming settings. In the streaming context, there has been a long line of research [14, 6, 20, 15], beginning with the work of Bar-Yossef et al. [1]. Let n be the number vertices, m be the number of edges, Δ be the maximum degree, and $\tau(G)$ be the number of triangles in the graph. An algorithm of [14] uses $\tilde{O}(m\Delta^2/\tau(G))$[1] space whereas the algorithm of [6] uses $\tilde{O}(mn/\tau(G))$ space. With higher space complexity, [20] and [15] give algorithms for the more general problem of counting cliques and cycles, supporting insertion and deletion of edges. A recent work [23] presents an algorithm with space complexity $\tilde{O}(m\Delta/\tau(G))$. Jha et al. [13] give a $O(\sqrt{n})$ space approximation algorithm for triangle counting as well as the closely related problem of the clustering coefficient. Their algorithm has an additive error guarantee as opposed to the previously mentioned algorithms, 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, 17], and there are also multi-pass streaming algorithms for approximate triangle counting [16]. However, no non-trivial parallel algorithms with the benefits of small-memory streaming algorithms were known so far, other than the naïve parallel algorithms, which are inefficient. In the non-streaming (batch) context, there are many algorithms for counting and enumerating triangles—both exact and approximate [7, 18, 24, 28, 3, 8]. Recent work on parallel algorithms in the MapReduce model includes [26, 9, 22].

2. PRELIMINARIES

We will be working with a simple, undirected graph whose edges arrive as a stream, where we assume that every edge arrives exactly once. Let $\mathcal{S} = (V, E, \leq_{\mathcal{S}})$ denote the graph on $G = (V, E)$, together with a total order $\leq_{\mathcal{S}}$ on E. An edge $e \in E$ is a size-2 set consisting of its endpoints. We write $m = |E|$ for the number of edges and Δ for the maximum degree of a vertex in G. 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 incident on e that "appear after" e in the $\leq_{\mathcal{S}}$ order; that is, $\Gamma_{\mathcal{S}}(e) := \{f \in E : f \cap e \neq \emptyset \wedge f >_{\mathcal{S}} e\}$.

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, we write $x \in_R S$ to indicate that x is a random sample from S taken uniformly at random.

Neighborhood Sampling: Our parallel algorithm builds on *neighborhood sampling* [23], a technique for selecting a random triangle from a streaming graph, which we now state as a set of invariants:

Invariant 2.1 *Let $\mathcal{S} = (V, E, \leq_{\mathcal{S}})$ be 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, χ), where $f_i \in E \cup \{\emptyset\}$ and $\chi \in \mathbb{Z}_+$, satisfies the* ***neighborhood sampling invariant (NBSI)*** *if*

(1) ***Level-1 Edge:*** *$f_1 \in_R E$ is chosen uniformly from E;*
(2) $\chi = |\Gamma_{\mathcal{S}}(f_1)|$ 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 completes the triangle formed uniquely defined by the edges f_1 and f_2 (or $\emptyset$ if the closing edge is not present).*

The invariant provides a way to maintain a random—although non-uniform—triangle in a graph. This directly translates to an unbiased estimator for τ:

Lemma 2.2 ([23]) *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, χ) be a tuple satisfying NBSI. Define random variable X as $X = 0$ if $f_3 = \emptyset$ and $X = \chi \cdot |E|$ otherwise. Then, $\mathbf{E}[X] = \tau(G)$.*

To obtain a sharper estimate, we run multiple copies of the estimate (making independent random choices) and aggregate them, for example, using median-of-means aggregate:

Theorem 2.3 ([23]) *There is an (ε, δ)-approximation to the triangle counting problem that requires at most r independent estimators on input a graph G with m edges, provided that $r \geq \frac{96}{\varepsilon^2} \cdot \frac{m\Delta(G)}{\tau(G)} \cdot \log(\frac{1}{\delta})$.*

Parallel Cost Model: We focus on parallel algorithms that efficiently utilize the memory hierarchy, striving to minimize cache[2] misses across the hierarchy. Towards this goal, we analyze the (theoretical) efficiency of our algorithms in terms of the memory cost—i.e., the number of cache misses—in addition to the standard cost analysis. Recall that **work** measures the total number of operations an algorithm performs, and **depth** is the length of the longest chain of dependent tasks in the algorithm. To measure the memory cost, we adopt the *parallel cache-oblivious (PCO)* model [5], a well-accepted parallel variant of the cache-oblivious model. In this model, the **memory cost** of an algorithm A, denoted by $Q^*(A; M, B)$, is given as a function of cache size M and line size B assuming the optimal offline replacement policy[3]. In the context of a parallel machine, it represents the number of cache misses across all processors for a particular level.

Parallel Primitives: We build on primitive operations that have existing parallel algorithms with optimal cache complexity and polylogarithmic depth. This helps simplify the exposition and allows us to use existing implementations.

Our algorithm relies on the following primitives: `sort`, `merge`, `concat`, `map`, `scan`, `extract`, and `combine`. The primitive `sort`

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

[2]The term cache is used as a generic reference to a level in the memory hierarchy; it could be an actual cache level (L1, L2, L3), the TLB, or page faults)

[3]In reality, practical heuristics such least-recently used (LRU) are used and are known to have competitive performance with the in-hindsight optimal policy.

takes a sequence and a comparison function, and outputs a sorted sequence. The primitive `merge` combines two sorted sequences into a new sorted sequence. The primitive `concat` concatenates the input sequences. The primitive `map` takes a sequence A and a function f, and it applies f on each entry of A. The primitive `scan` (aka. prefix sum or parallel prefix) takes a sequence A ($A_i \in D$), an associative binary operator $\oplus$ ($\oplus$: $D \times D \to D$), a left-identity id for $\oplus$ ($\mathsf{id} \in D$), and it produces the sequence $\langle \mathsf{id}, \mathsf{id} \oplus A_1, \mathsf{id} \oplus A_1 \oplus A_2, \ldots, \mathsf{id} \oplus A_1 \oplus \cdots \oplus A_{|A|-1} \rangle$. The primitive `extract` takes two sequences A and B, where $B_i \in \{1, \ldots, |A|\} \cup \{\texttt{null}\}$, and returns a sequence C of length $|B|$, where $C_i = A[B_i]$ or `null` if $B_i = \texttt{null}$. Finally, the primitive `combine` takes two sequences A, B of equal length and a function f, and outputs a sequence C of length $|A|$, where $C[i] = f(A[i], B[i])$.

On input of length N, the cache complexity of sorting in the PCO model[4] is $Q^*(\texttt{sort}(N); M, B) = O(\frac{N}{B} \log_{M/B}(1 + \frac{N}{B}))$, whereas `concat`, `map`, `scan`, and `combine` all have the same cost as `scan`: $Q^*(\texttt{scan}(N); M, B) = O(N/B)$. For `merge` and `concat`, N is the length of the two sequences combined. We also write $\texttt{sort}(N)$ and $\texttt{scan}(N)$ to denote the corresponding cache costs when the context is clear. In this notation, the primitive `extract` has $O(\texttt{sort}(|B|) + \texttt{scan}(|A| + |B|))$. 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. Also, let $T = \langle k'_1, \ldots, k'_m \rangle$ be a sequence of m keys from the same domain. The *exact multi-search 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. Existing cache-optimal `sort` and `merge` routines directly imply the following cost bounds:

Lemma 2.4 ([5, 4]) *The algorithms* $\texttt{exactMultiSearch}(S, T)$ *and* $\texttt{predEQMultiSearch}(S, T)$ *each runs 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. PARALLEL STREAMING ALGORITHM

This section presents the coordinated bulk-processing algorithm. For $i = 1, \ldots, r$, let est_i be a tuple (f_1, f_2, f_3, χ) satisfying NBSI on the graph $G = (V, E, \leq_S)$, where $\leq_S$ gives a total order on E. The sequence of arriving edges is modeled as $W = \langle w_1, \ldots, w_s \rangle$; the sequence order defines a total order on W. Denote by $G' = (V', E', \leq_{S'})$ the graph on $E \cup W$, where the edges of W all come after the edges of E in the new total order $\mathcal{S}'$.

The goal of the algorithm is to take as input estimators est_i ($i = 1, \ldots, r$) that satisfy NBSI on G and the arriving edges W, and produce as output estimators est'_i ($i = 1, \ldots, r$) that satisfy NBSI on G'. We show the following:

Theorem 3.1 *Let r be the number of estimators maintained for triangle counting. There is a parallel algorithm* `bulkUpdateAll` *that processes a batch of s edges with* $O(\texttt{sort}(r) + \texttt{sort}(s))$ *cache complexity (memory-access cost) and* $O(\log^2(r + s))$ *depth.*

To meet these bounds, we cannot afford to explicitly track each estimator individually. Neither can we afford a large number of random accesses. Despite these challenges, our algorithm proceeds in 3 simple steps, which correspond to the main parts of NBSI:

Step 1: Update level-1 edges;

Step 2: Update level-2 edges and neighborhood sizes χ's;

Step 3: Check for closing edges.

After these steps, the NBSI invariant is upheld; if the application so chooses, it can aggregate the "coarse" estimates costing no more than the update process itself. In the descriptions that follow, we will write $\Gamma_A(f)$ to mean $A \cap \Gamma_{\mathcal{S}'}(f)$, where $A \subseteq \mathcal{S}'$ is a set and $f \in \mathcal{S}'$ is an edge.

Step 1: Manage Level-1 Edges: The goal of Step 1 is to make sure that for each estimator, its level-1 edge f_1 is a uniform sample of the edges that have arrived so far ($E \cup W$). The conceptual algorithm for *one* estimator is straightforward: with probability $\frac{|W|}{|W|+|E|}$, replace f_1 with a random edge from W; otherwise, retain the current edge. Implementing this step in parallel is easy using `map` and `randInt`. For the estimators receiving a new level-1 edge, we also set its χ to 0 (this helps simplify the next step).

Step 2: Update Level-2 Edges and Degrees: The goal of Step 2 is to ensure that for every estimator est_i, the level-2 edge $est_i.f_2$ is chosen uniformly at random from $\Gamma_{\mathcal{S}'}(est_i.f_1)$ and $est_i.\chi = |\Gamma_{\mathcal{S}'}(est_i.f_1)|$. Remember that $\Gamma_{\mathcal{S}'}(f_1)$ is the set of edges incident on f_1 that appear after it in $\mathcal{S}'$, so χ is the size of $\Gamma_{\mathcal{S}'}(f_1)$, and f_2 a random sample from an appropriate "substream."

To describe the conceptual algorithm for one estimator, consider an estimator $est_i = (f_1, f_2, f_3, \chi)$ that has completed Step 1. We define

$$\chi^- = |\Gamma_E(f_1)| \quad \text{and} \quad \chi^+ = |\Gamma_W(f_1)|$$

In words, χ^- is the number edges in E incident on f_1 that arrived after f_1—and χ^+ is the number edges in W incident on f_1 that arrived after f_1. Thus, $\chi^- = \chi$ (inheriting it from the current state, so if f_1 was just replaced in Step 1, χ was reset to 0 in Step 1). We also note that $|\Gamma_{\mathcal{S}'}(f_1)|$—the total number of edges incident on f_1 that arrived after it in the whole stream—is $\chi^- + \chi^+$.

In this notation, the update rule is simply:

> With probability $\frac{\chi^-}{\chi^- + \chi^+}$, keep the current f_2; otherwise, pick a random edge from $\Gamma_W(f_1)$. Then, update χ to $\chi^- + \chi^+$.

An efficient implementation: To derive the implementation, we need to answer the question: *How to efficiently compute, for every estimator, the number of candidate edges χ^+ and how to sample uniformly from these candidates?* This is challenging because all r estimators need to navigate their substreams—potentially all different—to figure out their sizes and sample from them. Because of our performance requirements, we cannot afford to explicitly track these substreams. Neither can we afford a large number of random accesses, though this seems necessary at first.

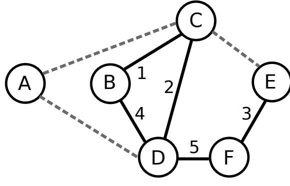
We address this challenge by first introducing the notion of *rank* and showing how it can be utilized effectively:

Definition 3.2 (Rank) *Let $W = \langle w_1, \ldots, w_s \rangle$ be a sequence of unique edges. Let $H = (V_W, W)$ be the graph on the edges W, where V_W is the set of relevant vertices on $E \cup W$. For $x, y \in V_W$, $x \neq y$, the* rank *of $x \to y$ is*

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

In words, if $\{x, y\}$ is an edge in W, $\text{rank}(x \to y)$ is the number of edges in W that are incident on x and appear after xy in W. Otherwise, $\text{rank}(x \to y)$ is simply the degree of x in the graph G_W. We provide an example in Figure 1. The rank of all arriving edges can be computed as follows:

[4] As is standard, we make a technical assumption known as the tall-cache assumption (i.e., $M \geq \Omega(B^2)$).



Arc	Rank
C → A	2
C → D	0
D → C	2
B → C	1
B → D	0

Figure 1: A 6-node streaming graph where a batch of 5 new edges (solid; arrival order labeled next to them) is being added to a stream of 3 edges (dashed) that have arrived earlier—and examples of the corresponding rank values as this batch is being incoporated.

ϕ:	0	1
src:	D	D
rank:	0	1
real edge:	DF	BD

ϕ:	0	1	2
src:	C	C	E
rank:	0	1	0
real edge:	CD	CB	EF

Figure 2: Translating rank values to substreams: (Left) f_1 = DC using u = D and v = C, so $\chi^+ = 2$. (Right) f_1 = CE using u = C and v = E, so $\chi^+ = 3$.

Lemma 3.3 *There is a parallel algorithm* `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* {`src`, `dst`, `rank`, `pos`} *such that*

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

Each input edge w_i gives rise to exactly 2 entries, one per orientation. The algorithm runs in $O($`sort`*$(s))$ cache cost and $O(\log^2 s)$ depth.*

We omit the algorithm's description and proof due to space constraints; readers are referred to [27].

Mapping rank to substreams: The following easy-to-verify observation *implicitly* defines a substream with respect to a level-1 edge f_1 in terms of rank values:

Observation 3.4 *Let $f_1 = \{u, v\} \in E \cup W$ be a level-1 edge. Let F' =* `rankAll`(W)*. The set of edges in W incident on f_1 that appears after it—i.e., the set $\Gamma_W(f_1)$—is precisely the undirected edges corresponding to $L \cup R$, where*

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

and

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

Hence, there are $\mathrm{rank}(u \to v)$ edges in $\Gamma_W(f_1)$ incident on u (those in L) and $\mathrm{rank}(v \to u)$ edges, on v (those in R)—and thus, $\chi^+ = \mathrm{rank}(u \to v) + \mathrm{rank}(v \to u)$.

The other piece of the puzzle is, how to sample from the substreams? For this, we give a "naming system" for identifying which level-2 edge to pick: Associate each number $\phi \in \{0, 1, \ldots, \chi^+ - 1\}$ with an edge in $\Gamma_W(f_1)$ as follows: if $\phi < \mathrm{rank}(u \to v)$, we associate it with the edge `src` $= u$ and `rank` $= \phi$; otherwise, associate it with the edge `src` $= v$ and `rank` $= \phi - \mathrm{rank}(u \to v)$. Therefore, the problem of picking a random level-2 replacement edge boils down to picking a random number between 0 and $\chi^+ - 1$ and locating the corresponding edge in the batch of arriving edges. We give two examples in Figure 2 to illustrate the naming system. Please refer to [27] for details.

Step 3: Locate Closing Edges: The goal of Step 3 is to detect, for each estimator, if the wedge formed by level-1 and level-2 edges closes using a edge from W. This step can be easily implemented by using `map` to compute the closing edges and using a multisearch to locate them; please refer to [27] for details.

Cost Analysis: Let $s = |W|$ and r be the number of estimators we maintain. Step 1 is implemented using `map`, `extract`, and `combine`. Therefore, the cache cost for Step 1 is at most $O($`sort`$(r)+$ `scan`$(r + s))$, and the depth is at most $O(\log^2(r + s))$. The cost of both Steps 2 and 3 is dominated by a sort, which is at most $O($`sort`$(r) +$ `sort`$(s))$ for memory cost and $O(\log^2(r + s))$ depth. Hence, the total cost of `bulkUpdateAll` is $O($`sort`$(r) +$ `sort`$(s))$ memory cost and $O(\log^2(r + s))$ depth, as promised.

4. EVALUATION

We have implemented the algorithm from Section 3 and investigated its performance on real-world datasets in terms of accuracy, parallel speedup, and parallelization overhead. More detailed analysis appears in [27].

Implementation: We followed the description in Section 3 rather closely. The algorithm combines the "coarse" estimators into a sharp estimate using a median-of-means aggregate. The main optimization we made was in avoiding malloc-ing small chunks of memory often. The `sort` primitive uses a PCO sample sort algorithm [4, 25], which offers good speedups. We implemented the multisearch routines by modifying Blelloch et al.'s `merge` algorithm [4] to stop recursing early when the number of "queries" is small. Other primitives are standard. The main triangle counting logic has about 600 lines of Cilk code, a dialect of C/C++ with keywords that allow users to specify what should be run in parallel [12] (fork/join-type code). Our benchmark programs were compiled with the public version of Cilk shipped with GNU `g++` Compiler version 4.8.0 (20130109).

Testbed and Datasets: We performed experiments on a 12-core (with hyperthreading) Intel machine, running Linux 2.6.32-279 (CentOS 6.3). The machine has *two* 2.67 Ghz 6-core Xeon X5650 processors with 96GB of memory although the experiments never need more a few gigabytes of memory.

Our study uses a collection of graphs, obtained from the SNAP project at Stanford [19].We present a summary of these datasets in Table 1. While most of these datasets are social-media graphs, our algorithm does not assume any special property about the datasets. We simulate a streaming graph by feeding the algorithm with edges as they are read from disk. We note now that disk I/O is not a bottleneck in the experiment.

For most datasets, the exact triangle count is provided by the source (which we have verified); in other cases, we compute the exact count using an algorithm developed as part of the Problem-Based Benchmark Suite [25]. We also report the size on disk of these datasets in the format we obtain them (a list of edges in plain text). In addition, we include one synthetic power-law graph, which we cannot obtain the true count, but it is added for speed testing.

Baseline: For accuracy study, we directly compare our results with the true count. For performance study, our baseline is a fairly optimized version of the nearly-linear time algorithm based on neighborhood sampling, using bulk-update, as described in [23]. We use this baseline to establish the overhead of the parallel algorithm. 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. The baseline code was also compiled with the same compiler *but* without linking with the Cilk runtime.

Performance and Accuracy: We perform experiments on graphs with varying sizes and densities. Our algorithm is randomized and

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
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).

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 with hyperthreading), as well as the speedup ratio[5]. 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] In another experiment, presented in Table 3, we compare our parallel implementation with the baseline sequential implementation [23]. Evidently, *the I/O cost is not a bottleneck in any of the experiments, justifying the need for parallelism to improve throughput.*

Several trends are clear from these experiments. *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 20 million estimators, and for smaller datasets, it can obtain better than 5% mean deviation using fewer estimators. As a general trend—though not a strict pattern—accuracy improves with the number of 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 $r = 20\text{M}$ estimators, the speedups are consistently above 10x. This shows that the algorithm is able to effectively utilize available cores except when the dataset or the number of estimators is too small to fully utilize parallelism. Additionally, we experimented with a big synthetic graph (167GB power-law graph). Althogh we were unable to calculate the true count and had to cut short the sequential experiment after a few hours, it is worth pointing out that 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 on Friendster, showing empirically that it scales well.

Third, the overhead is well-controlled. Both the sequential and parallel algorithms, at the core, maintain the same neighborhood sampling invariant but differ significantly in how the edges are processed. As is apparent from Table 3, for large datasets requiring more estimators, the overhead is less than 1.5x with $r = 20\text{M}$. For smaller datasets, the overhead is less than 1.6x with $r = 2\text{M}$. In all cases, the amount of speedup gained outweighs the overhead.

5. CONCLUSION

We have presented a cache-efficient parallel algorithm for approximating the number of triangles in a massive streaming graph. The proposed algorithm is cache-oblivious and has good theoretical performance and accuracy guarantees. We also experimentally showed that the algorithm is fast and accurate, and has low cache complexity. It will be interesting to explore other problems at the intersection of streaming algorithms and parallel processing.

[5]A batch size of 16M was used

[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`.

Acknowledgments

Tangwongsan 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. Pavan was sponsored in part by NSF grant 0916797; Tirthapura was sponsored in part by NSF grants 0834743 and 0831903.

Bibliography

[1] Z. Bar-Yossef, R. 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.

[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.

[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.

[4] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *SPAA'10*, pages 189–199. ACM, 2010.

[5] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *SPAA'11*, pages 355–366. ACM, 2011.

[6] L. S. Buriol, G. Frahling, S. Leonardi, and C. Sohler. Estimating clustering indexes in data streams. In *Proc. European Symposium on Algorithms (ESA)*, pages 618–632, 2007.

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

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

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

[10] J.-P. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings*

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	2.08	1.07	0.14	7.64	0.47	3.58	0.42	8.52	0.12	29.00	3.15	9.21	0.14
DBLP	1.18	1.21	0.17	7.12	0.39	3.71	0.43	8.63	0.08	29.20	3.14	9.30	0.15
LiveJournal	5.51	35.10	3.31	10.60	0.51	41.00	3.91	10.49	0.39	69.80	6.56	10.64	1.52
Orkut	2.39	119.00	11.20	10.62	0.16	133.00	12.40	10.73	0.66	172.00	15.40	11.17	5.00
Friendster	18.25	1730.00	181.00	9.56	8.35	1970.00	193.00	10.21	3.41	2330.00	219.00	10.64	86.00
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.

Dataset	r = 200K			r = 2M			r = 20M		
	T_{seq}	T_1	$^{T_1}/_{T_{\text{seq}}}$	T_{seq}	T_1	$^{T_1}/_{T_{\text{seq}}}$	T_{seq}	T_1	$^{T_1}/_{T_{\text{seq}}}$
Amazon	0.95	1.07	1.13	5.25	3.58	0.68	34.80	29.00	0.83
DBLP	0.92	1.21	1.32	5.16	3.71	0.72	36.90	29.20	0.79
LiveJournal	14.00	35.10	2.51	34.90	41.00	1.17	84.40	69.80	0.83
Orkut	42.00	119.00	2.83	73.20	133.00	1.82	129.00	172.00	1.33
Friendster	787.00	1730.00	2.20	1040.00	1970.00	1.89	1580.00	2330.00	1.47

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}).

of the National Academy of Sciences, 99(9):5825–5829, 2002. doi: 10.1073/pnas.032093399.

[11] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.

[12] Intel Corp. Intel Cilk Plus, 2013. http://cilkplus.org/.

[13] M. Jha, C. Seshadhri, and A. Pinar. From the birthday paradox to a practical sublinear space streaming algorithm for triangle counting. *CoRR*, abs/1212.2264, 2012.

[14] H. Jowhari and M. Ghodsi. New streaming algorithms for counting triangles in graphs. In *Proc. 11th Annual International Conference Computing and Combinatorics (COCOON)*, pages 710–716, 2005.

[15] D. M. Kane, K. Mehlhorn, T. Sauerwald, and H. Sun. Counting arbitrary subgraphs in data streams. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 598–609, 2012.

[16] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. In *WAW*, pages 15–24, 2010.

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

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

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

[20] M. Manjunath, K. Mehlhorn, K. Panagiotou, and H. Sun. Approximate counting of cycles in streams. In *Proc. European Symposium on Algorithms (ESA)*, pages 677–688, 2011.

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

[22] R. Pagh and C. E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Inf. Process. Lett.*, 112(7): 277–281, 2012.

[23] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu. Counting and sampling triangles from a graph stream. *PVLDB*, 6(14), 2013.

[24] T. Schank and D. 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.

[25] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012.

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

[27] K. Tangwongsan, A. Pavan, and S. Tirthapura. Parallel triangle counting in massive streaming graphs. *CoRR*, abs/1308.2166, 2013.

[28] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. 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.

[29] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Netw. Analys. Mining*, 1(2):75–81, 2011.

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