Tri-Fly: Distributed Estimation of Global and Local Triangle Counts in Graph Streams

Kijung Shin[1](✉), Mohammad Hammoud[2], Euiwoong Lee[1],
Jinoh Oh[3], Christos Faloutsos[1]

[1]Carnegie Mellon University, USA, {kijungs,euiwoonl,christos}@cs.cmu.edu
[2]Carnegie Mellon University in Qatar, Qatar, mhhamoud@cmu.edu
[3]Adobe Systems, USA, joh@adobe.com

Abstract. Given a graph stream, how can we estimate the number of triangles in it using multiple machines with limited storage?
Counting triangles (i.e., cycles of length three) is a classical graph problem whose importance has been recognized in diverse fields, including data mining, social network analysis, and databases. Recently, for triangle counting in massive graphs, two approaches have been intensively studied. One approach is streaming algorithms, which estimate the count of triangles incrementally in time-evolving graphs or in large graphs only part of which can be stored. The other approach is distributed algorithms for utilizing computational power and storage of multiple machines.
Can we have the best of both worlds? We propose Tri-Fly, the first distributed streaming algorithm for approximate triangle counting. Making one pass over a graph stream, Tri-Fly rapidly and accurately estimates the counts of global triangles and local triangles incident to each node. Compared to state-of-the-art single-machine streaming algorithms, Tri-Fly is **(a) Accurate:** yields up to *4.5× smaller* estimation error, **(b) Fast**: runs up to *8.8× faster* with linear scalability, and **(c) Theoretically sound**: gives unbiased estimates with smaller variances.

Keywords: Graph Stream, Triangle Counting, Edge Sampling

1 Introduction

Counting triangles (i.e., cycles of length three) is a classical graph problem whose importance has been recognized in diverse areas. In data mining, the count of triangles was used for dense subgraph mining [19], spam detection [5], degeneracy estimation [16], and web structure analysis [8]. In social network analysis, many important concepts (e.g., the clustering coefficients and social balance) are based on the count of triangles [20]. In databases, the count of triangles, which measures the degree of transitivity of a relation, can be used for query optimization [4].

Due to this importance, many algorithms have been developed for counting global triangles (i.e., all triangles in a graph) and/or local triangles (i.e., triangles incident to each node in a graph). Especially, for triangle counting in massive graphs, recent work has focused largely on streaming algorithms [2,7,10,11,14,15] and distributed algorithms [3,12,17].

In a graph stream, where edges are streamed from sources, streaming algorithms [2,7,10,11,14,15] estimate the count of triangles by making one pass over the stream, even when the stream does not fit in the underlying storage. Moreover, since streaming algorithms incrementally update their estimates as each edge arrives, they can naturally be used for maintaining and updating approximate triangle counts in dynamic graphs growing with new edges. However, existing streaming algorithms are designed to run on a single machine and do not utilize multiple machines for faster or more accurate estimation.

On the other hand, distributed algorithms have been employed for utilizing computational and storage resources in distributed-memory [3] and MapReduce [17,12] settings. However, they do not provide the advantages of streaming algorithms. That is, they assume that all edges can be stored in the underlying storage and accessed multiple times. Moreover, since they are batch algorithms rather than incremental algorithms, they do not support efficient updates of triangle counts in dynamic graphs growing with new edges.

In this work, we propose Tri-Fly, the first distributed streaming algorithm for approximate counting of global and local triangles. Tri-Fly gives the advantages of both streaming and distributed algorithms, outperforming state-of-the-art single-machine streaming algorithms. Our theoretical and empirical analyses show that Tri-Fly has the following advantages:

- **Accurate**: Tri-Fly produces up to *4.5*× smaller *estimation error* than baselines with similar speeds (Figure 3)
- **Fast**: Tri-Fly runs in linear time (Figure 2(c)) up to *8.8*× *faster* than baselines with similar accuracies (Figure 3).
- **Theoretically sound:** Tri-Fly gives unbiased estimates with variances inversely proportional to the number of machines (Theorems 1 and 2).

Reproducibility: The code and datasets used in the paper are available at `http://www.cs.cmu.edu/~kijungs/codes/trifly/`.

2 Related Work

Triangle Counting in Graph Streams. Streaming algorithms estimate the count of triangles by making one pass over a graph stream. Streaming algorithms use sampling because they assume limited storage that may not store all edges. A simple but effective sampling technique is edge sampling. Doulion [18] uniformly samples edges of a large graph, and estimates its global triangle count from that in the sampled graph. Mascot [11] improves upon Doulion in terms of accuracy by utilizing unsampled edges. Specifically, whenever an edge arrives, Mascot counts the global and local triangles formed by the incoming edge and edges sampled so far, even if the incoming edge is not sampled but discarded. While Mascot may discard edges even when storage is not full, Triest$_{\text{IMPR}}$ [7] always maintains as many samples as storage allows, leading to higher accuracy. When edges are streamed in the chronological order, WRS [15] improves upon Triest$_{\text{IMPR}}$ in terms of accuracy by exploiting temporal dependencies in the

Table 1: Comparison of triangle counting algorithms. Notice that **only our proposed algorithm Tri-Fly satisfies all the criteria.**

	(Distributed)		(Streaming)		(Proposed)
	[12,17]	[3]	[2,9,14]	[7,10,11,15]	**Tri-Fly**
Single-Pass Stream Processing			✓	✓	✓
Approximation for Large Graphs		✓	✓	✓	✓
Global & Local Triangle Counting	✓			✓	✓
Larger Data w/ More Machines	✓	✓			✓
More Accurate w/ More Machines		✓			✓

edges. In addition to edge sampling, wedge sampling [9], neighborhood sampling [14], and sample-and-hold [2] were used for global triangle counting, and node coloring [10] was used for local triangle counting. Neighborhood sampling was parallelized in a shared-memory setting where edges arrive in batches, and it was also extended to cases where edges are streamed from multiple sources [13].

Distributed Triangle Counting. Many MapReduce algorithms for exact counts of triangles have been proposed based on the assumption that all edges of the input graph are stored in a distributed file system. The first such algorithm [6] parallelizes node iterator, a serial algorithm for triangle counting. GP [17] divides the input graph into overlapping subgraphs and assigns them to machines, which count the triangles in the assigned subgraphs in parallel. Since the subgraphs are not disjoint, GP produces a large amount of intermediate data, which were reduced in [12]. The idea of dividing the input graph into overlapping subgraphs was used also in a distributed memory setting [3]. These existing distributed algorithms are batch algorithms for static graphs, while we propose incremental algorithms for dynamic graph streams.

The aforementioned streaming algorithms and distributed algorithms are summarized and compared in Table 1.

3 Notations and Problem Definition

3.1 Notations (Table 2)

Consider an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with the set of nodes $\mathcal{V}$ and the set of edges $\mathcal{E}$. Each unordered pair $(u, v) \in \mathcal{E}$ indicates the edge between two distinct nodes $u, v \in \mathcal{V}$. We denote the set of triangles (i.e., three nodes, every pair of which is connected by an edge) in $\mathcal{G}$ by $\mathcal{T}$ and those with node u by $\mathcal{T}[u] \subset \mathcal{T}$. We call $\mathcal{T}$ *global triangles* and $\mathcal{T}[u]$ *local triangles* of node u. Each unordered triple $(u, v, w) \in \mathcal{T}$ denotes the triangle with three distinct nodes $u, v, w \in \mathcal{V}$.

Consider a graph stream $(e^{(1)}, e^{(2)}, ...)$ where $e^{(t)}$ denotes the edge that arrives at time $t \in \{1, 2, ...\}$. We use t_{uv} to denote the arrival time of edge (u, v). Let $\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$ be the graph at time t consisting of the nodes and edges arriving at time t or earlier. Then, $\mathcal{T}^{(t)}$ denotes the set of global triangles in $\mathcal{G}^{(t)}$ and $\mathcal{T}^{(t)}[u] \subset \mathcal{T}^{(t)}$ denotes the set of local triangles of each node $u \in \mathcal{V}^{(t)}$ in $\mathcal{G}^{(t)}$.

Table 2: Table of frequently-used symbols.

	Symbol	Definition
Notations for Graph Streams (Section 3)	$\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$	graph at time t
	$e^{(t)}$	edge that arrives at time t
	(u, v)	edge between nodes u and v
	t_{uv}	arrival time of edge (u, v)
	(u, v, w)	triangle with nodes u, v, and w
	$\mathcal{T}^{(t)}$	set of global triangles in $\mathcal{G}^{(t)}$
	$\mathcal{T}^{(t)}[u]$	set of local triangles with node u in $\mathcal{G}^{(t)}$
Notations for Algorithm (Section 4)	$\mathcal{M}$, $\mathcal{W}$, $\mathcal{A}$	sets of masters, workers, and aggregators
	k	maximum number of edges stored in each worker
	l_i	number of edges that worker i has received
	$h : \mathcal{V} \cup \{*\} \to \mathcal{A}$	hash function that maps nodes to aggregators
	$\bar{c}$	estimate of the count of global triangles
	$c[u]$	estimate of the count of local triangles of node u

3.2 Problem Definition

In this work, we consider the problem of estimating the counts of global and local triangles in a graph stream using multiple machines with limited storage. Specifically, we assume the following realistic conditions:

C1 **No prior knowledge:** no information about the input graph stream (e.g., the number of edges, degree distribution, etc.) is available in advance.
C2 **Shared nothing architecture:** each machine cannot access data stored in the other machines.
C3 **Limited storage:** at most k (≥ 2) edges can be stored in each of n machines, while the number of edges in the input graph stream can be greater than k or even nk.
C4 **Single pass:** edges are processed one by one in their arrival order. Past edges cannot be accessed unless they are stored (in the storage in C3).

Based on these conditions, we define the problem of distributed estimation of global and local triangle counts in a graph stream.

Problem 1 (Distributed Estimation of Triangle Counts in a Graph Stream).

(1) **Given:** a graph stream $(e^{(1)}, e^{(2)}, ...)$, and n distributed storages each of which can store up to k (≥ 2) edges
(2) **Minimize:** the estimation errors of global triangle count $|\mathcal{T}^{(t)}|$ and local triangle counts $\{|\mathcal{T}^{(t)}[u]|\}_{u \in \mathcal{V}^{(t)}}$ for each time $t \in \{1, 2, ...\}$.

Instead of minimizing a specific measure of estimation error, we use a general approach of simultaneously reducing bias and variance to reduce various measures of estimation error robustly.

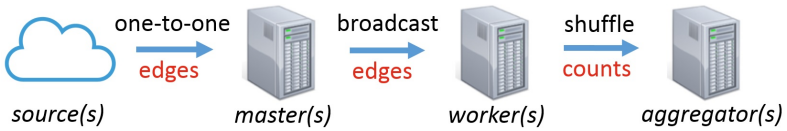


Fig. 1: Flow of data in Tri-Fly.

4 Proposed Method: Tri-Fly

We propose Tri-Fly, a distributed streaming algorithm for approximate triangle counting. We first present the overview of Tri-Fly. Then, we discuss its details. Lastly, we provide theoretical analyses on its accuracy and complexity.

4.1 Overview (Figure 1)

Figure 1 shows the flow of data in Tri-Fly. Edges are streamed from sources to masters so that each edge is sent to exactly one master. Each master broadcasts the received edges to the workers. Each worker estimates the global and local triangle counts independently using its local storage. To this end, we adapt $\text{TRIEST}_{\text{IMPR}}$, which estimates both global and local triangle counts with no prior knowledge, although any streaming algorithm can be used instead.[1] The counts are shuffled so that the counts of local triangles of each node (or the counts of global triangles) are sent to the same aggregator. The aggregators aggregate the counts and give the final estimates of the counts of global and local triangles.

4.2 Detailed Algorithm (Algorithm 1)

Algorithm 1 describes Tri-Fly. We first define the notations used in it. Then, we explain masters, workers, and aggregators. Lastly, we discuss lazy aggregation.

Notations. We use $\mathcal{M}$, $\mathcal{W}$ and $\mathcal{A}$ to indicate the set of masters, workers and aggregators, respectively. Each worker can store up to k (≥ 2) edges, and $\mathcal{E}_i$ denotes the set of edges currently stored in worker $i \in \mathcal{W}$. We let $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$ be the graph consisting of the edges in $\mathcal{E}_i$. For each node $u \in \mathcal{V}_i$, $\mathcal{N}_i[u]$ indicates the neighbors of u in $\mathcal{G}_i$. We use l_i to denote the number of edges that worker $i \in \mathcal{W}$ has received so far. If $l_i > k$, then $l_i > |\mathcal{E}_i|$ since not all received edges can be stored. We use $h : \mathcal{V}_i \cup \{*\} \to \mathcal{A}$ to denote a hash function that maps nodes (the keys for local triangle counts) and '$*$' (the key for global triangle counts) to aggregators. Lastly, $\bar{c}$ denotes the estimate of the count of global triangles, and for each node u, $c[u]$ denotes the estimate of the count of local triangles of u.

Masters (lines 2-3). Whenever each master receives an edge from the sources, the master broadcasts the edge to the workers.

Workers (lines 5-16). Each worker independently estimates the global and local triangle counts, and shuffles the counts across the aggregators. Note that

[1] e.g., WRS [15] can be used instead if edges are streamed in the chronological order.

Algorithm 1: TRI-FLY

Input : input graph stream: $(e^{(1)}, e^{(2)}, ...)$, storage budget in each worker: k
Output: estimated global triangle count: $\bar{c}$
estimated local triangle counts: $c[u]$ for each node u

- Master (each master):
2 **for** *each edge* (u, v) *from the sources* **do**
3 broadcast (u, v) to the workers

- Worker (each worker $i \in \mathcal{W}$):
5 $l_i \leftarrow 0$, $\mathcal{E}_i \leftarrow \emptyset$
6 **for** *each edge* (u, v) *from the masters* **do**
7 $sum \leftarrow 0$
8 **for** *each node* $w \in \mathcal{N}_i[u] \cap \mathcal{N}_i[v]$ **do**
9 send $(w, 1/(p_i[uvw]))$ to aggregator $h(w)$
10 $sum \leftarrow sum + 1/(p_i[uvw])$ ▷ see Eq. (1) for $p_i[uvw]$
11 send $(*, sum)$ to aggregator $h(*)$ ▷ '$*$': key for the global triangle count
12 send (u, sum) to aggregator $h(u)$ and (v, sum) to aggregator $h(v)$
13 $l_i \leftarrow l_i + 1$
14 **if** $|\mathcal{E}_i| < k$ **then** $\mathcal{E}_i \leftarrow \mathcal{E}_i \cup \{(u, v)\}$
15 **else if** *a random number in* $\mathit{Bernoulli}(k/l_i)$ *is* 1 **then**
16 replace a random edge in $\mathcal{E}_i$ with (u, v)

- Aggregator (each aggregator $j \in \mathcal{A}$):
17 **if** $h(*) = j$ **then** $\bar{c} \leftarrow 0$
18 initialize an empty map c with default value 0
19 **for** *each pair* (u, δ) *from the workers* **do**
20 **if** $u = *$ **then** $\bar{c} \leftarrow \bar{c} + \delta/|\mathcal{W}|$
21 **else** $c[u] \leftarrow c[u] + \delta/|\mathcal{W}|$

the workers use different random seeds and thus shuffle different counts. Each worker $i \in \mathcal{W}$ starts with an empty storage (i.e., $\mathcal{E}_i = \emptyset$) (line 5). Whenever it receives an edge (u, v) from a master (line 6), the worker counts the triangles composed of (u, v) and two edges in its local storage; and sends the counts to the corresponding aggregators using hash function h (lines 7-12). Then, the worker samples (u, v) in its local storage with non-zero probability (lines 13-16). Below, we explain in detail how each worker samples edges and counts triangles.

For sampling (lines 13-16), each worker $i \in \mathcal{W}$ first increases l_i, the number of edges that it has received, by one (line 13). If its local storage is not full (i.e., $|\mathcal{E}_i| < k$), the worker always stores (u, v) by adding (u, v) to $\mathcal{E}_i$ (line 14). If the local storage is full (i.e., $|\mathcal{E}_i| = k$), the worker stores (u, v) with probability k/l_i by replacing a random edge in $\mathcal{E}_i$ with (u, v) (lines 15-16). This is the standard reservoir sampling, which guarantees that each pair of the l_i edges is sampled (i.e., included in $\mathcal{E}_i$) with the equal probability $\min\left(1, \frac{k(k-1)}{l_i(l_i-1)}\right)$.

For counting (lines 7-12), each worker $i \in \mathcal{W}$ finds the common neighbors of nodes u and v in graph $\mathcal{G}_i$ (i.e., the graph consisting of the edges $\mathcal{E}_i$ in its local storage) (line 8). Each common neighbor w indicates the existence of the

triangle (u, v, w). Thus, for each common neighbor w, the worker increases the global triangle count and the local triangle counts of nodes u, v, and w by sending the increases to the corresponding aggregators (lines 9, 11, and 12). The amount of increase in the counts is $1/(p_i[uvw])$ for each triangle (u, v, w), where

$$p_i[uvw] := \min\left(1, \frac{k(k-1)}{l_i(l_i-1)}\right) \tag{1}$$

is the probability that triangle (u, v, w) is discovered by worker i (i.e., both (v, w) and (w, u) are in $\mathcal{E}_i$ when worker i receives (u, v)), as explained above. Increasing counts by $1/(p_i[uvw])$ guarantees that the expected amount of the increase sent from each worker is exactly $1 (= p_i[uvw] \times 1/(p_i[uvw]) + (1 - p_i[uvw]) \times 0)$ for each triangle, enabling Tri-Fly to give unbiased estimates (see Theorem 1).

Aggregators (lines 17-21). Each aggregator maintains and updates the triangle counts assigned by the hash function h. That is, aggregator $j \in \mathcal{A}$ maintains the estimate $c[u]$ of the count of local triangles of node u if $h(u) = j$. Likewise, aggregator $j \in \mathcal{A}$ maintains the estimate $\bar{c}$ of the count of global triangles if $h(*) = j$. Specifically, each aggregator increases the estimates by $1/|\mathcal{W}|$ of what it receives, averaging the increases sent from the workers (lines 20-21).

Lazy Aggregation (optional). In Algorithm 1, each worker sends the increase of the local triangle count of node w to the corresponding aggregator whenever it discovers each triangle (u, v, w) (line 9). Likewise, each worker sends the updates of the global triangle count and the local triangle counts of nodes u and v to the corresponding aggregators whenever it processes each edge (u, v) (lines 11-12). In cases where this eager aggregation is not needed, we can reduce the amount of shuffled data by employing lazy aggregation. That is, counts can be aggregated locally in each worker until they are queried. If queried, the counts are sent to and aggregated in the aggregators and removed from the workers.

4.3 Bias and Variance Analyses

We analyze the biases and variances of the estimates given by Tri-Fly. The biases and variances determine the errors of the estimates. For the analyses, let $\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$ be the graph with the edges arriving at time t or earlier. We define $\bar{c}^{(t)}$ as $\bar{c}$ in the aggregator $h(*)$ after edge $e^{(t)}$ is processed. Likewise, for each node $u \in \mathcal{V}^{(t)}$, let $c^{(t)}[u]$ be $c[u]$ in the aggregator $h(u)$ after $e^{(t)}$ is processed. Then, $\bar{c}^{(t)}$ is an estimate of $|\mathcal{T}^{(t)}|$, the global triangle count in $\mathcal{G}^{(t)}$, and each $c^{(t)}[u]$ is an estimate of $|\mathcal{T}^{(t)}[u]|$, the local triangle count of u in $\mathcal{G}^{(t)}$.

We first prove the unbiasedness of Tri-Fly, formalized in Theorem 1.

Theorem 1 (Unbiasedness of Tri-Fly). *At any time, the expected values of the estimates given by* Tri-Fly *are equal to the true global and local triangle counts. That is, in Algorithm 1,*

$$\mathbb{E}[\bar{c}^{(t)}] = |\mathcal{T}^{(t)}|, \ \forall t \geq 1, \quad \textit{and} \quad \mathbb{E}[c^{(t)}[u]] = |\mathcal{T}^{(t)}[u]|, \ \forall u \in \mathcal{V}^{(t)}, \ \forall t \geq 1.$$

Proof Sketch. Consider a triangle $(u, v, w) \in \mathcal{T}^{(t)}$. Let $d_i[uvw]$ be the contribution of (u, v, w) to $\bar{c}^{(t)}$ by each worker $i \in \mathcal{W}$. Then, by the definition of $p_i[uvw]$, and

Table 3: Time and space complexities of Tri-Fly for processing the first t edges in the input graph stream.

	Masters (Total)	Workers (Each)	Workers (Total)	Aggregators (Total)
Time	$O(t \cdot \lvert\mathcal{W}\rvert)$	$O(t \cdot \min(t,k))$	$O(\lvert\mathcal{W}\rvert \cdot t \cdot \min(t,k))$	$O(\lvert\mathcal{W}\rvert \cdot t \cdot \min(t,k))$*
Space	$O(\lvert\mathcal{M}\rvert)$	$O(\min(t,k))$	$O(\lvert\mathcal{W}\rvert \cdot \min(t,k))$	$O(\lvert\mathcal{V}^{(t)}\rvert)$

* can be reduced by lazy aggregation

lines 11 and 20 of Algorithm 1, $d_i[uvw] = 1/(|\mathcal{W}| \cdot p_i[uvw])$ with probability $p_i[uvw]$, and $d_i[uvw] = 0$ with probability $(1 - p_i[uvw])$. Therefore, $\mathbb{E}[d_i[uvw]] = 1/|\mathcal{W}|$. Then, $\mathbb{E}[d_i[uvw]] = 1/|\mathcal{W}|$ and linearity of expectation imply

$$\mathbb{E}[\bar{c}^{(t)}] = \mathbb{E}\Bigg[\sum_{i\in\mathcal{W}} \sum_{(u,v,w)\in\mathcal{T}^{(t)}} d_i[uvw]\Bigg] = \sum_{i\in\mathcal{W}} \sum_{(u,v,w)\in\mathcal{T}^{(t)}} \mathbb{E}[d_i[uvw]] = |\mathcal{T}^{(t)}|, \forall t \geq 1.$$

See [1] for a full proof with the unbiasedness of the other estimates. ■

Theorem 2 presents the result of our variance analysis given in the supplementary document [1]. The variance of each $c^{(t)}[u]$ can be analyzed in the same manner considering only the triangles with node u.

Theorem 2 (Variance of Tri-Fly). *The variance of the estimate $\bar{c}^{(t)}$ in* Tri-Fly *is inversely proportional to the number of workers. Let $r^{(t)}$ be the number of triangle pairs in $\mathcal{T}^{(t)}$ where (a) an edge is shared and (b) the shared edge is not last to arrive in any of the two triangles. Let $z^{(t)}$ be* $\max\big(0, |\mathcal{T}^{(t)}|\big(\frac{(t-1)(t-2)}{k(k-1)} - 1\big) + r^{(t)}\big(\frac{t-1-k}{k}\big)\big)$. *Then, Eq.* (2) *holds in Algorithm 1.*

$$Var[\bar{c}^{(t)}] \leq \frac{z^{(t)}}{|\mathcal{W}|}, \ \forall t \geq 1. \tag{2}$$

Proof Sketch. For each worker $i \in \mathcal{W}$, let $\bar{c}_i^{(t)}$ be the global triangle count sent from the worker by time t. Then, $\bar{c}^{(t)} = \sum_{i\in\mathcal{W}} \bar{c}_i^{(t)}/|\mathcal{W}|$ (line 20 of Algorithm 1). Eq. (2) follows from $Var[\bar{c}_i^{(t)}] \leq z^{(t)}$ for each $i \in \mathcal{W}$ (Lemma 1 in [1]) and independence between $\bar{c}_i^{(t)}$ and $\bar{c}_j^{(t)}$ for $i \neq j$. See [1] for a full proof. ■

4.4 Time and Space Complexity Analyses

We summarize the time and space complexities of Tri-Fly in Table 3. Detailed analyses with proofs are given in the supplementary document [1]. Notice that, with a fixed storage budget k, the time complexity of Tri-Fly is linear in the number of edges, as confirmed empirically in Section 5.2. The results also suggest that reducing storage budget k and using more masters and aggregators need to be considered if the input stream is too fast to be processed.

Table 4: Summary of real-world and synthetic graph streams.

Name	# Nodes	# Edges	Summary
BerkStan	685,230	6,649,470	Web
Patent	3,774,768	16,518,947	Citation
Flickr	2,302,925	22,838,276	Friendship
FriendSter	65,608,366	1,806,067,135	Friendship
Random **(800GB)**	1,000,000	1,000,000,000 − 100,000,000,000	Synthetic

5 Experiments

In this section, we conduct experiments to answer the following questions:

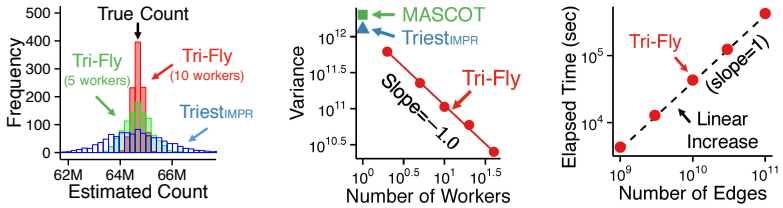
- **Q1. Illustration of Theorems**: Does Tri-Fly give unbiased estimates? How rapidly do their variances drop as the number of workers is scaled up? How does Tri-Fly scale with the size of the input stream?
- **Q2. Performance**: How fast and accurate is Tri-Fly compared to the best single-machine streaming algorithms?

5.1 Experimental Settings

Machines: All experiments were conducted on a cluster of 40 machines with 3.47GHz Intel Xeon X5690 CPUs and 32GB RAM.
Datasets: The graph datasets used in the paper are summarized in Table 4. The self loops, duplicated edges, and the directions of edges were ignored.
Implementations: We implemented Tri-Fly, Triest$_{\text{IMPR}}$ [7] (single-machine) and Mascot [11] (single-machine) in C++ and MPICH 3.1. In them, sampled edges were stored in main memory in the adjacency list format. For Tri-Fly, we used 1 master and 1 aggregator. We used lazy aggregation (see the last paragraph of Section 4.2) and aggregated all counts once at the end of each input stream. We simulated graph streams by streaming edges in a random order from the disk of machines that host the master of Tri-Fly or single-machine algorithms.
Evaluation Metrics: We evaluated the accuracy of each algorithm at the end of each input stream. Let x be the true global triangle count, and $\hat{x}$ be its estimate obtained by an evaluated algorithm. Likewise, for each node $u \in \mathcal{V}$, let $x[u]$ be the true local triangle count of u and $\hat{x}[u]$ be its estimate. We used *global error*, defined as $\frac{|x-\hat{x}|}{1+x}$, and *RMSE*, defined as $\sqrt{\frac{1}{|\mathcal{V}|}\sum_{u\in\mathcal{V}}(x[u]-\hat{x}[u])^2}$, to evaluate the accuracy of global triangle counting and local triangle counting, respectively.

5.2 Q1. Illustration of Our Theorems (Figure 2)

Illustration of Unbiasedness (Theorem 1). Figure 2(a) shows the distributions of 1,000 estimates of the global triangle count in the BerkStan dataset obtained by Tri-Fly and Triest$_{\text{IMPR}}$. We set storage budget k so that each



(a) Unbiasedness (b) Variances drop (c) Scalable

Fig. 2: **Theoretical properties of Tri-Fly.** (a) Tri-Fly gives unbiased estimates. (b) The variances of the estimates drop inversely proportional to the number of workers. (c) Tri-Fly scales linearly with the size of the input stream.

worker stored up to 5% of the edges. The averages of the estimates given by Tri-Fly were close to the true triangle count, as expected from Theorem 1.

Illustration of Variance Decrease (Theorem 2). Under the same experimental settings, Figure 2(b) shows the variances of the estimates of the global triangle count obtained by different algorithms. We measured the sample variance of $1,000$ estimates in each setting. The variance in Tri-Fly dropped inversely proportional to the number of workers, as expected in Theorem 2.

Illustration of Linear Scalability (Section 4.4) We measured the running time of Tri-Fly while varying the size of the input stream. We used 30 workers and fixed the storage budget k to 10^7. To measure the scalability independently of the speed of input streams, we measured the time taken to process edges, ignoring the time taken by the master to wait for the arrival of edges in input streams. Figure 2(c) shows the results with random graph streams with 1 million nodes and different numbers of edges. The largest one was **800GB** with **100 billion edges**. Tri-Fly scaled linearly with the size of the input stream, as expected in Section 4.4. We obtained similar results when graph streams with realistic structure were used (see the supplementary document[1]).

5.3 Q2. Performance (Figure 3)

Since Tri-Fly is the first distributed streaming algorithm for triangle counting, there is **no direct competitor** of Tri-Fly. As baselines, we used Triest$_{\text{IMPR}}$ [7] and Mascot [11], which are the state-of-the-art single-machine streaming algorithms estimating both global and local triangle counts (see Table 1).

We measured the speeds and accuracies of the considered algorithms with different storage budgets. To compare their speeds independently of the speed of input streams, we measured the time taken by each algorithm to process edges, ignoring the time taken to wait for the arrival of edges in input streams. All evaluation metrics and running times were averaged over 10 trials in the Friendster dataset and over 100 trials in the other datasets.

As seen in Figure 3, Tri-Fly showed the best performance in every dataset. Specifically, Tri-Fly was up to **8.8× faster** than baselines with similar accu-

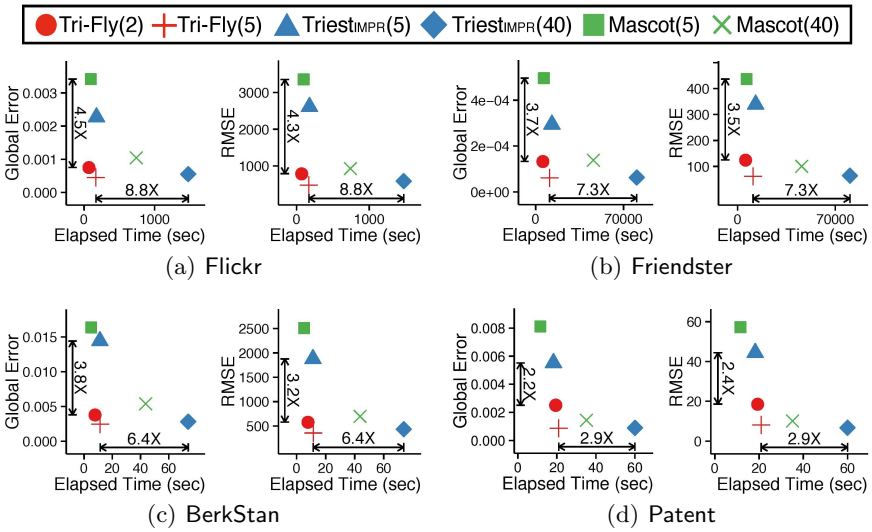


Fig. 3: **Tri-Fly achieves both speed and accuracy.** In each plot, points represent the speeds and errors of different algorithms (the numbers in parentheses indicate the percentage of edges that can be stored in each worker). Tri-Fly was up to **4.5× more accurate** than single-machine baselines with similar speeds, and it was up to **8.8× faster** than those with similar accuracies.

racies. In terms of global error and RMSE, Tri-Fly was up to **4.5× and 4.3× more accurate** than baselines with similar speeds, respectively.

6 Conclusion

In this work, we propose Tri-Fly, the first distributed streaming algorithm estimating the counts of global and local triangles with the following strengths:

- **Accurate**: Tri-Fly yields up to *4.5× and 4.3× smaller estimation errors* for global and local triangle counts than similarly fast baselines (Figure 3).
- **Fast**: Tri-Fly is up to *8.8× faster* than similarly accurate baselines (Figure 3). Tri-Fly scales linearly with the size of the stream (Figure 2(c)).
- **Theoretically sound**: Tri-Fly yields unbiased estimates whose variances drop as the the number of machines is scaled up (Theorems 1 and 2).

Reproducibility: The code and datasets used in the paper are available at `http://www.cs.cmu.edu/~kijungs/codes/trifly/`.

Acknowledgements. This material is based upon work supported by the National Science Foundation under Grants No. CNS-1314632 and IIS-1408924. Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053. This publication was made possible by

NPRP grant # 7-1330-2-483 from the Qatar National Research Fund (a member of Qatar Foundation). Shin was supported by KFAS Scholarship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, or other funding parties. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

1. Supplementary document. Available online: `http://www.cs.cmu.edu/~kijungs/codes/trifly/supple.pdf` (2018)
2. Ahmed, N.K., Duffield, N., Neville, J., Kompella, R.: Graph sample and hold: A framework for big-graph analytics. In: KDD (2014)
3. Arifuzzaman, S., Khan, M., Marathe, M.: Patric: A parallel algorithm for counting triangles in massive networks. In: CIKM (2013)
4. Bar-Yossef, Z., Kumar, R., Sivakumar, D.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: SODA (2002)
5. Becchetti, L., Boldi, P., Castillo, C., Gionis, A.: Efficient algorithms for large-scale local triangle counting. TKDD 4(3), 13 (2010)
6. Cohen, J.: Graph twiddling in a mapreduce world. Computing in Science & Engineering 11(4), 29–41 (2009)
7. De Stefani, L., Epasto, A., Riondato, M., Upfal, E.: Triest: Counting local and global triangles in fully-dynamic streams with fixed memory size. In: KDD (2016)
8. Eckmann, J.P., Moses, E.: Curvature of co-links uncovers hidden thematic layers in the world wide web. PNAS 99(9), 5825–5829 (2002)
9. Jha, M., Seshadhri, C., Pinar, A.: A space efficient streaming algorithm for triangle counting using the birthday paradox. In: KDD (2013)
10. Kutzkov, K., Pagh, R.: On the streaming complexity of computing local clustering coefficients. In: WSDM (2013)
11. Lim, Y., Kang, U.: Mascot: Memory-efficient and accurate sampling for counting local triangles in graph streams. In: KDD (2015)
12. Park, H.M., Myaeng, S.H., Kang, U.: Pte: Enumerating trillion triangles on distributed systems. In: KDD (2016)
13. Pavan, A., Tangwongan, K., Tirthapura, S.: Parallel and distributed triangle counting on graph streams. Technical report, IBM, Tech. Rep. (2013)
14. Pavan, A., Tangwongsan, K., Tirthapura, S., Wu, K.L.: Counting and sampling triangles from a graph stream. PVLDB 6(14), 1870–1881 (2013)
15. Shin, K.: Wrs: Waiting room sampling for accurate triangle counting in real graph streams. In: ICDM (2017)
16. Shin, K., Eliassi-Rad, T., Faloutsos, C.: Patterns and anomalies in k-cores of real-world graphs with applications. Knowl. Inf. Syst. 54(3), 677–710 (2018)
17. Suri, S., Vassilvitskii, S.: Counting triangles and the curse of the last reducer. In: WWW (2011)
18. Tsourakakis, C.E., Kang, U., Miller, G.L., Faloutsos, C.: Doulion: counting triangles in massive graphs with a coin. In: KDD (2009)
19. Wang, J., Cheng, J.: Truss decomposition in massive networks. PVLDB 5(9), 812–823 (2012)
20. Wasserman, S., Faust, K.: Social network analysis: Methods and applications, vol. 8. Cambridge university press (1994)