A Space-efficient Parallel Algorithm for Counting Exact Triangles in Massive Networks

Shaikh Arifuzzaman*†, Maleq Khan*, and Madhav Marathe*†
*Network Dynamics & Simulation Science Laboratory, Virginia Bioinformatics Institute
†Department of Computer Science
Virginia Tech, Blacksburg, Virginia 24061 USA
Email: {sm10, maleq, mmarathe}@vbi.vt.edu

***Abstract*—Finding the number of triangles in a network (graph) is an important problem in mining and analysis of complex networks. Massive networks emerging from numerous application areas pose a significant challenge in network analytics since these networks consist of millions, or even billions, of nodes and edges. Such massive networks necessitate the development of efficient parallel algorithms. There exist several MapReduce and an only MPI (Message Passing Interface) based distributed-memory parallel algorithms for counting triangles. MapReduce based algorithms generate prohibitively large intermediate data. The MPI based algorithm can work on quite large networks, however, the overlapping partitions employed by the algorithm limit its capability to deal with very massive networks.**

In this paper, we present a space-efficient MPI based parallel algorithm for counting *exact* number of triangles in massive networks. The algorithm divides the network into non-overlapping partitions. Our results demonstrate up to 25-fold space saving over the algorithm with overlapping partitions. This space efficiency allows the algorithm to deal with networks which are 25 times larger. We present a novel approach that reduces communication cost drastically (up to 90%) leading to both a space- and runtime-efficient algorithm. Our adaptation of a parallel partitioning scheme by computing a novel weight function adds further to the efficiency of the algorithm. Denoting average degree of nodes and the number of partitions by $\bar{d}$ and P, respectively, our algorithm achieves up to $O(P^2)$-factor space efficiency over existing MapReduce based algorithms and up to $\bar{d}$-factor (approx.) over the algorithm with overlapping partitioning.

***Keywords*-counting triangles, parallel algorithms, massive networks, social networks, graph mining, space efficiency.**

I. Introduction

Counting triangles in a network is a fundamental algorithmic problem in the analysis of complex networks. It has many important applications such as computing clustering coefficient, transitivity, and triangular connectivity of networks [1]. Further, counting triangles has been used in detecting spamming activity and assessing content quality of networks [2], uncovering the thematic structure of the web [3], query optimization in database [4], and detecting communities or clusters in social and information networks [5].

Network is a powerful abstraction for representing underlying relations in large unstructured datasets. Examples include the web graph [6], various social networks [7], biological networks [8], and many other information networks. In the era of big data, the emerging network data is also very large. Social networks such as Facebook and Twitter have millions to billions of users [1], [9]. Such massive networks motivate the need for space-efficient parallel algorithms.

Existing Algorithms. The problem of counting triangles has a rich history [2], [10]–[12]. Despite the fairly large volume of work addressing this problem, only recently has attention been given to the problems associated with massive networks. Several techniques can be employed to deal with such massive networks: streaming algorithms [2], [13], sparsification based algorithms [14], external-memory algorithms [1], and parallel algorithms [12], [15], [16]. The streaming and sparsification based algorithms are approximation algorithms. External memory algorithms can be very I/O intensive leading to a large runtime. Efficient parallel algorithms can achieve running efficiency by distributing computing tasks among multiple processors.

Over the last couple of years, several parallel algorithms have been proposed. Two parallel algorithms for exact triangle counting using the MapReduce framework are presented in [12]. The first algorithm generates huge volumes of intermediate data by enumerating all possible 2-paths which require a large amount of time and memory while shuffling and regrouping. The second algorithm suffers from redundant counting of triangles. An improvement of the second algorithm is given in [17]. Though this scheme reduces the redundant count of [12] to some extent, the redundancy is not entirely eliminated. In fact, for P partitions, the algorithm over-counts (P-1 times) triangles whose vertices lie in the same partition. Further, the expected size of the output from all Map instances is $O(mP)$ (m is the number of edges) which is significantly larger than the size of the original network.

In a recent work [18], Park et al. propose a randomized MapReduce algorithm for triangle enumeration. To achieve a runtime complexity of the optimal serial algorithm, each reducer requires a space of $\Omega\{m^{3/4}\sqrt{\log m}\}$ (Theorem 3 in [18]), even though it gives an approximate count. Another MapReduce based approximation algorithm is proposed in [16], which is based on wedge sampling technique [14].

A recent paper [15] proposes an MPI based parallel algorithm for counting the *exact* number of triangles in massive networks. The algorithm employs an overlapping partitioning scheme and a novel load balancing scheme. The overlapping partitions eliminate the need for message exchanges leading to a fast algorithm. However, the overlapping partitions pose significant space overhead since those might grow as large as the whole network wiping out the benefit of partitioning.

Although there exists a couple of standard parallel graph partitioning algorithms such as Parmetis and Zoltan [19], those might not work well for our problem. Those algorithms strive to minimize cut edges, which help reduce communication overhead, however, we also require the computation cost to be well-balanced among processors. We need to estimate weights of nodes (based on triangle counting cost) in parallel in the partitioning procedure which is not readily available in standard algorithms. Hence we adapt a parallel partitioning scheme [15] which considers the actual triangle counting cost incurred at nodes and thus helps in balancing computation loads.

Our Contributions. In this paper, we present a space-efficient MPI based parallel algorithm for counting *exact* number of triangles in massive networks. The algorithm divides the network into non-overlapping partitions and achieves a space efficiency of up to 25 times over the algorithm with overlapping partitions for the networks we experimented on. This space efficiency allows the algorithm to deal with networks which are 25 times larger. We present a novel approach that reduces approx. 70% to 90% of communication cost without requiring additional space, which leads to both a space- and runtime-efficient algorithm. Our adaptation of a parallel partitioning scheme by computing a novel weight function offers additional runtime efficiency to the algorithm. Our algorithm achieves up to $O(P^2)$-factor space saving over existing MapReduce based algorithms and up to $\bar{d}$-factor (approx.) over the algorithm with overlapping partitioning.

Remarks. Note that unlike approximation algorithms which provide an overall (global) estimate of number of triangles in the graph, this paper presents an exact algorithm which can be used to count triangles incident on individual nodes (local triangles). Such *local* counts facilitate computing clustering coefficient of nodes and finding vertex neighborhood and community seeds [20]. To the best of our knowledge, among all exact algorithms, our algorithm has the lowest space complexity, without even compromising its runtime efficiency.

II. Preliminaries

Below are the notations, definitions, datasets, and computation model used in this paper.

Notation and Definitions. The given network is denoted by $G(V, E)$, where V and E are the sets of vertices and edges, respectively, with $m = |E|$ edges and $n = |V|$ vertices labeled as $0, 1, 2, \ldots, n-1$. We use the words *node* and *vertex* interchangeably. We assume that the input network is undirected. If $(u, v) \in E$, we say u and v are neighbors to each other. The set of all neighbors of $v \in V$ is denoted by $\mathcal{N}_v$, i.e., $\mathcal{N}_v = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |\mathcal{N}_v|$. A triangle is a set of three nodes $u, v, w \in V$ such that there is an edge between each pair of these three nodes, i.e., $(u, v), (v, w), (w, u) \in E$. The number of triangles incident on v, denoted by T_v, is same as the number of edges among the neighbors of v, i.e.,

TABLE I
Dataset used in our experiments. K, M and B denote thousands, millions and billions, resp.

Network	Nodes	Edges	Source
web-Google	0.88M	5.1M	SNAP [23]
web-BerkStan	0.69M	6.5M	SNAP [23]
Miami	2.1M	50M	[21]
LiveJournal	4.8M	43M	SNAP [23]
Twitter	42M	2.4B	[24]
PA(n, d)	n	$\frac{1}{2}nd$	Pref. Attachment

$$T_v = |\{(u, w) \in E : u, w \in \mathcal{N}_v\}|.$$

Let P be the number of processors used in the computation, which we denote by $p_0, p_1, \ldots, p_{P-1}$ where each subscript refers to the rank of a processor.

Datasets. We use both real world and artificially generated networks for our experiments. A summary of all the networks is provided in Table I. Miami [21] is a synthetic, but realistic, social contact network for Miami city. Twitter, LiveJournal, web-BerkStan, and web-Google are real-world networks. Artificial network PA(n, d) is generated using preferential attachment (PA) model [22] with n nodes and average degree d. Both real-world and PA(n, d) networks have very skewed degree distributions. Networks having such distributions create difficulty in partitioning and balancing loads and thus give us a chance to measure the performance of our algorithms in some of the worst case scenarios. Note that in our experiments we consider edges of the input graph to be undirected– we ignore the original directionality of edges for web-Google, web-BerkStan, and LiveJournal networks.

Computation Model. We develop parallel algorithms for MPI based distributed-memory parallel systems where each processor has its own local memory. The processors do not have any shared memory, and they communicate via exchanging messages.

III. A Background on Counting Triangles

First, we describe the state-of-the-art sequential algorithm which our parallel algorithm is based on. Space complexity of other related parallel algorithms is discussed thereafter.

A. *Efficient Sequential Algorithm.*

A naïve approach to count triangles in a graph $G(V, E)$ is to check, for all possible triples (u, v, w), $u, v, w \in V$, whether (u, v, w) forms a triangle; i.e., check if $(u, v), (v, w), (u, w) \in E$. There are $\binom{n}{3}$ such triples, and thus this algorithm takes $\Omega(n^3)$ time. A simple but efficient algorithm for counting triangles is: for each node $v \in V$, find the number of pairs of neighbors that complete a triangle with vertex v. In this method, each triangle (u, v, w) is counted six times – all six permutations of u, v, and w. A total ordering $\prec$ of the nodes (e.g., ordering based on node IDs or any arbitrary ordering) makes sure each triangle is counted exactly once. However, algorithms in [10], [11] incorporate an interesting node ordering based on the degrees of the nodes, with ties

```
1: for each edge $(u, v)$ do
2:     if $u \prec v$, store $v$ in $N_u$
3:     else store $u$ in $N_v$
4: for $v \in V$ do
5:     sort $N_v$ in ascending order
6: $T \leftarrow 0$        {$T$ is the count of triangles}
7: for $v \in V$ do
8:     for $u \in N_v$ do
9:         $S \leftarrow N_v \cap N_u$
10:        $T \leftarrow T + |S|$
```

Fig. 1. The state-of-the-art sequential algorithm for counting triangles.

broken by node IDs, as defined as follows:

$$u \prec v \iff d_u < d_v \text{ or } (d_u = d_v \text{ and } u < v). \qquad (1)$$

These algorithms are further improved in a recent paper [15] by a simple modification. The algorithm [15] defines $N_v \subseteq \mathcal{N}_v$ as the set of neighbors of v having a higher order $\prec$ than v,

$$N_v = \{u : (u, v) \in E, v \prec u\}. \qquad (2)$$

That is, for an edge (u, v), the algorithm stores u in N_v if $v \prec u$, and consequentially, $u \in N_v \iff v \notin N_u$. Then the triangles containing node v and any $u \in N_v$ can be found by set intersection $N_u \cap N_v$. Now we call $\hat{d_v} = |N_v|$ as the *effective degree* of node v. The cost for computing $N_u \cap N_v$ requires $O(\hat{d_v} + \hat{d_u})$ time when N_v and N_u are sorted leading to a total runtime of $O\left(\sum_{v \in V} d_v \hat{d_v}\right)$ (Theorem 2 in [15]). The above state-of-the-art sequential algorithm is presented in Fig. 1.

B. Space Complexity of Related Parallel Algorithms.

Among the parallel algorithms discussed in Section I, there are several MapReduce based algorithms [12], [17] and an MPI based algorithm [15] that count *exact* number of triangles. The MapReduce based algorithm proposed in [12] works in two rounds of Map and Reduce phases. In Map phases, the algorithm generates a huge amount of intermediate data which are all possible 2-paths w-v-u centered around each node $v \in V$, such that $u, w \in \mathcal{N}_v$. The algorithm then check whether such 2-paths are closed by an edge, i.e. if $(w, u) \in E$. Since the number of these 2-paths is very large, even larger than the network size, shuffling and regrouping these data requires a large runtime and enormous memory. As instance, for Twitter network, $300B$ 2-paths are generated whereas the network has only $2.4B$ edges. The space requirement becomes prohibitively excessive for very large networks. Even for smaller networks, if there are few nodes with high degrees, say $O(n)$, this algorithm generates $O(n^2)$ 2-paths centered at those nodes, which is quite unmanageable. Many real networks demonstrate power-law degree distributions where some nodes have very large degrees (See d_{max} in Table II).

Another MapReduce algorithm proposed in [12], the partition-based algorithm, has a space requirement of $O(mP)$ for the Map phase (with P partitions), which is P times larger than the network size. The algorithm in [17] improves the runtime of the partition-based algorithm of [12], however the space requirement still remains same.

The MPI based algorithm in [15] divides the input graph into a set of P overlapping partitions as follows. First, V is partitioned into P disjoint subsets V_i^c, such that $\bigcup_{0 \le k < P} V_k^c = V$. Then, a set V_i is constructed as $V_i = V_i^c \cup \left(\bigcup_{v \in V_i^c} N_v\right)$. Now, set of edges E_i, defined as $E_i = \{(u, v) | u \in V_i, v \in N_u\}$, constitutes the i-th overlapping partition which p_i works on. Note that edges in $E_i^c = \{(u, v) | u \in V_i^c, v \in N_u\}$ constitute the *disjoint* (non-overlapping) portion of the partition i. Rest of the edges $(u, v) \in E_i - E_i^c$ overlaps across multiple partitions. Now, the overlapping partitions allow the algorithm to count triangles without any communication among processors leading to faster computation. However, overlapping partitions have a significant space overhead. Assuming an average degree $\bar{d}$ of the network, the algorithm has a space requirement of $\Omega(\frac{n\bar{d}}{P})$ for storing *disjoint* portion of the partition. Storing the whole partition requires $\Omega(\frac{xn\bar{d}}{P})$ or $\Omega(\frac{xm}{P})$ space, where $1 \le x \le \bar{d}$, which can be as large as the whole network $O(m)$.

Our space-efficient parallel algorithm divides the input networks into non-overlapping partitions. The load balancing procedure makes sure that the computational cost is almost equal in each partition. We also observe experimentally that the largest partition has approximately $\frac{m}{P}$ edges. The sizes of all partitions sum up to the size of the network. This partitioning offers as much as $\bar{d}$ times saving over the overlapping partitions and thus allows to work on larger networks. Table II shows the space requirement of our algorithm for several real and artificial networks, which is up to $25\times$ smaller than that of [15].

TABLE II
MEMORY USAGE OF OUR ALGOIRTHM AND [15] FOR STORING THE LARGEST PARTITION. NUMBER OF PARTITIONS USED IS 100.

Networks	Memory (MB)		Ratio	$\bar{d}$	d_{max}
	Our algo.	[15]			
web-Google	1.49	11.3	7.85	11.6	6332
LiveJournal	9.41	110.75	11.75	18	20333
Miami	10.63	109.58	10.32	47.6	425
Twitter	265.82	4254.18	16.004	57.1	1001159
PA(10M, 100)	121.11	2120.94	17.5	100	25068
PA(1M, 1000)	138.20	3427.36	24.8	1000	19255

We present our parallel algorithm in the following section.

IV. A SPACE EFFICIENT PARALLEL ALGORITHM

First, we present an overview of the algorithm. A detailed description follows thereafter.

A. Overview of the Algorithm.

Our algorithm partitions the input network $G(V, E)$ into a set of P partitions constructed as follows: set of nodes V is partitioned into P disjoint subsets V_i, such that, for $0 \le j, k \le P - 1$ and $j \ne k$, $V_j \cap V_k = \emptyset$ and $\bigcup_k V_k = V$. Edge set E_i, constructed as $E_i = \{(u, v) : u \in V_i, v \in N_u\}$, constitutes the i-th partition. Note that this partition is non-overlapping– each edge $(u, v) \in E$ resides in one and only one partition. For $0 \le j, k \le P-1$ and $j \ne k$, $E_j \cap E_k = \emptyset$ and $\bigcup_k E_k = E$. The sum of space required to store all partitions equals to the space required to store the whole network. Processor p_i works on the i-th partition and is responsible for having all triangles incident on nodes $v \in V_i$ counted.

Now, to count triangles incident on $v \in V_i$, p_i needs N_u for all $u \in N_v$ (Lines 7-10, Fig. 1). If $u \in V_i$, information of both N_v and N_u is available in the i-th partition, and p_i counts triangles incident on (v, u) by computing $N_u \cap N_v$. However, if $u \in V_j$, $j \ne i$, N_u resides in partition j. Processor p_i and p_j exchange message(s) to count triangles incident on such (v, u). This exchanging of messages introduces a communication overhead which is a crucial factor for the performance of the algorithm. We devise an efficient approach to reduce the communication overhead drastically and improve the performance significantly. Once all processors complete the computation associated with respective partitions, the counts from all processors are aggregated.

B. Computing Partitions.

While constructing partitions i, set of nodes V is partitioned into P disjoint subsets V_i of consecutive nodes. How the nodes in V are distributed among the sets V_i for all partitions i crucially affect the performance of the algorithm. Distributing equal number of nodes for each partition might not make computational load even among the processors. Ideally, the set V should be partitioned in such a way that the cost for counting triangles is almost equal for all partitions. Let, $f(v)$ be a *weight function* referring to the cost for counting triangles incident on $v \in V$(cost for executing Line 7-10, Fig. 1). We need to compute P disjoint partitions of V such that for each partition V_i,

$$\sum_{v \in V_i} f(v) \approx \frac{1}{P} \sum_{v \in V} f(v). \tag{3}$$

Several estimations for $f(v)$ were proposed in [15] among which $f(v) = \sum_{u \in N_v} (\hat{d_v} + \hat{d_u})$ was shown experimentally as the best. Since our algorithm employs a different communication scheme for counting triangles, none of those estimations corresponds to the cost of our algorithm. Thus, we compute a new weight function $f(v)$ to estimate the computational cost of our algorithm more precisely in Section V-B (Theorem 2).

Once $f(v)$ is computed for all $v \in V$, we compute cumulative sum $F(t) = \sum_{v=0}^{t} f(v)$ in parallel by using a parallel prefix sum algorithm [25]. Processor p_i computes and stores $F(t)$ for nodes t, where t starts from $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. This computation takes $O(\frac{n}{P} + P)$ time. Now, let $V_i = \{n_i, n_i + 1 \ldots, n_{(i+1)} - 1\}$ for some node $n_i \in V$. We call n_i the *start* or *boundary* node of partition i. Now, V_i is computed in such a way that the sum $\sum_{v \in V_i} f(v)$ becomes almost equal ($\frac{1}{P} \sum_{v \in V} f(v)$) for all partitions i. At the end of this execution, each processor p_i knows boundary nodes n_i and $n_{(i+1)}$. We adapt the algorithm presented in [15] to compute V_i for our problem by using our newly computed cost function $f(v)$. In summary, computing partitions has the following main steps.

- *Step 1:* Compute a new cost function $f(v)$ which corresponds to the triangle counting cost of our algorithm (Section V-B).
- *Step 2:* Compute cumulative sum $F(v)$ by a parallel prefix sum algorithm [25].
- *Step 3:* Compute boundary nodes n_i for every subset $V_i = \{n_i, \ldots, n_{(i+1)} - 1\}$ using the algorithms in [15].

Once all P partitions are computed, processor p_i is assigned the partition V_i.

C. Counting Triangles with An Efficient Communication Approach.

As discussed in the overview of our algorithm, processor p_i and p_j require to exchange messages for counting triangles incident on (v, u) where $v \in V_i$ and $u \in N_v \cap V_j$. A straightforward approach for this communication might be very inefficient. For example, in a simple way, such triangles can be counted as follows: p_i *requests* p_j *for* N_u. *Upon receiving the request,* p_j *sends* N_u *to* p_i. *Processor* p_i *counts triangles incident on the edge* (v, u) *by computing* $N_v \cap N_u$. For further reference, we call this approach as 'Direct approach'.

We observe that this approach has a high communication overhead due to exchanging a large number of redundant messages leading to a large runtime. Assume $u \in N_{v_1} \cap N_{v_2} \cap \cdots \cap N_{v_k}$, for $v_1, v_2, \ldots, v_k \in V_i$. Then p_i sends k separate requests for N_u to p_j while computing triangles incident on v_1, v_1, $\ldots$, v_k. In response to those requests, p_j sends same message N_u to p_i for k times.

One seemingly obvious way to eliminate redundant messages is that instead of requesting N_u multiple times, p_i stores it in memory for subsequent use. However, space requirement for storing all N_u along with the partition i itself is same as that of storing an overlapping partition. This diminishes our original goal of a space-efficient algorithm.

Another way of eliminating message redundancy is as follows. When N_u is fetched, p_i completes all computation that requires N_u: p_i finds all k nodes $v \in V_i$ such that $u \in N_v$. It then performs all k computations $N_v \cap N_u$ involving N_u and discards N_u. Now, since $u \in N_v \iff v \notin N_u$, p_i cannot extract such k nodes v from the message N_u. Instead, p_i requires to scan through its whole partition to find such nodes v where $u \in N_v$. This *scanning* is very expensive– $O(\sum_{v \in V_i} d_v)$ in the worst case for each message– which might even be slower than the direct approach with redundant messages.

All the above techniques to improve the efficiency of *Direct* approach introduce additional space or runtime overhead. Next we propose an efficient approach which exploits an inherent property of our data structure N_v to reduce message exchanges drastically without adding further overhead.

Reduction of messages. As discussed before, p_i cannot count triangles on (v, u) for $v \in V_i$ and $u \in N_v \cap V_j$ without fetching N_u from partition j. Now, we take a different viewpoint: we ask the question, what is the implication for p_i delegating the computation $N_v \cap N_u$ to p_j instead of doing by itself? In particular, we consider the following approach: p_i

sends N_v to p_j instead of fetching N_u. Processor p_j counts triangles incident on edge (u, v) by performing the operation $N_v \cap N_u$.

We call this approach as *Surrogate* approach. On a surface, this might seem to be a simple modification from *Direct* approach. However, notice the following implication which is very significant to the algorithm: once p_j receives N_v, it can extract the information of all nodes u, such that u is in both N_v and V_j, by just scanning N_v. For all such nodes u, p_j counts triangles incident on edge (u, v) by performing the operation $N_v \cap N_u$. Processor p_j then discard N_v since it is no longer needed. Note that extracting all u such that $u \in N_v$ and $u \in V_j$ requires $O(d_v)$ time (compare this to $O(\sum_{v \in V_i} d_v)$ runtime of direct approach for the similar purpose). In fact, this extraction can be done while computing triangles $N_v \cap N_u$ for first such u. This saves from any additional overhead.

As we noticed, if delegated, p_j can count triangles on multiple edges (u, v) from a single message N_v, where $v \in V_i$ and $u \in N_v \cap V_j$. Thus p_i does not require to send N_v to p_j multiple times for each such u. However, to avoid multiple sending, p_i needs to keep track of which processors it has already sent N_v to. This *message tracking* is also crucial since any additional space or runtime overhead might compromise the efficiency of the overall approach.

It is easy to see that one can perform the above tracking by maintaining P *flag* variables, one for each processor. Before sending N_v to a particular processor p_j, p_i checks j-th flag to see if it is already sent. All flags are initially reset to zero. This implementation is conceptually simple but cost for resetting flags for each $v \in V_i$ sums to a significant cost of $O(|V_i|.P)$. Now, note the following simple yet useful property of N_v: *Since V_j is a set of consecutive nodes, and all neighbor lists N_v are sorted, all nodes $u \in N_v \cap V_j$ reside in N_v in consecutive positions.*

The above property enable each p_i to track messages by only recording the last processor (say, *LastProc*) it has sent N_v to. When p_i encounters $u \in N_v$ such that $u \in V_j$, it checks *LastProc*. If *LastProc* $\neq p_j$, then p_i sends N_v to p_j and set *LastProc* $= p_j$. Otherwise, the node u is ignored, meaning it would be redundant to send N_v. Resetting a single variable *LastProc* for all computation involving N_v does not introduce any additional overhead.

Thus surrogate approach detects and eliminates message redundancy and allows multiple computation from a single message, without even compromising execution or space efficiency. The efficiency gained from this capability is shown both theoretically and experimentally in Section V and VI, respectively.

D. Pseudocode for Counting Triangles.

We denote a message by $\langle t, X \rangle$ where $t \in \{data, notifier\}$ is the type and X is the actual data associated with the message. For a data message ($t = data$), X refers to a neighbor list N_x whereas for a completion notifier ($t = notifier$), $X = \emptyset$. The pesudocode for counting triangles for an incoming data message $\langle data, X \rangle$ is given in Fig. 2.

```
1: Procedure SurrogateCount($X, i$) :
2: $T \leftarrow 0$   //$T$ is the count of triangles
3: for all $u \in X$ such that $u \in V_i$ do
4:     $S \leftarrow N_u \cap X$
5:     $T \leftarrow T + |S|$
6: return $T$
```

Fig. 2. A procedure executed by p_i to count triangles for the received message $\langle data, X \rangle$ from some p_j in accordance to surrogate approach.

Once a processor p_i completes the computation on all $v \in V_i$, it broadcasts a completion notifier $\langle notifier, X \rangle$. However, it cannot terminate execution until it receives $\langle notifier, X \rangle$ from all other processors since other processors might send data messages for surrogate computation. Finally, p_0 sums up counts from all processors using MPI aggregation function. The complete pseudocode of our algorithm using surrogate approach is presented in Fig. 3.

```
1: $T_i \leftarrow 0$   //$T_i$ is $p_i$'s count of triangles
2: for $v \in V_i$ do
3:     for $u \in N_v$ do
4:         if $u \in V_i$ then
5:             $S \leftarrow N_v \cap N_u$
6:             $T_i \leftarrow T_i + |S|$
7:         else if $u \in V_j$ then
8:             Send $\langle data, N_v \rangle$ to $p_j$ if not sent already
9:
10:     Check for incoming messages $\langle t, X \rangle$:
11:     if $t = data$ then
12:         $T_i \leftarrow T_i +$ SurrogateCount($X, i$)
13:     else
14:         Increment completion counter
15:
16: Broadcast $\langle notifier, X \rangle$
17: while completion counter < P-1 do
18:     Check for incoming messages $\langle t, X \rangle$:
19:     if $t = data$ then
20:         $T_i \leftarrow T_i +$ SurrogateCount($X, i$)
21:     else
22:         Increment completion counter
23:
24: MpiBarrier
25: Find Sum $T \leftarrow \sum_i T_i$ using MpiReduce
26: return $T$
```

Fig. 3. An algorithm for counting triangles using surrogate approach. Each processor p_i executes Line 1-22. After that, they are synchronized and the aggregation is performed (Line 24-26).

V. Theoretical Analysis

We present the theoretical justification for efficiency and correctness of our algorithm in this section.

A. Correctness of The Algorithm.

The correctness of our space efficient parallel algorithm is formally presented in the following theorem.

Theorem 1. *Given a graph $G = (V, E)$, our space efficient parallel algorithm correctly counts exact number of triangles in G.*

Proof: Consider a triangle (x_1, x_2, x_3) in G, and without the loss of generality, assume that $x_1 \prec x_2 \prec x_3$. By the constructions of N_x (Line 1-3 in Fig. 1), we have $x_2, x_3 \in N_{x_1}$ and $x_3 \in N_{x_2}$. Now, there might be two cases:

1. *Case $x_1, x_2 \in V_i$:*
   Nodes x_1 and x_2 are in the same partition i. Processor p_i executes the loop in Line 2-6 (Fig. 3) with $v = x_1$ and $u = x_2$, and node x_3 appears in $S = N_{x_1} \cap N_{x_2}$, and the triangle (x_1, x_2, x_3) is counted once. But this triangle cannot be counted for any other values of v and u because $x_1 \notin N_{x_2}$ and $x_1, x_2 \notin N_{x_3}$.
2. *Case $x_1 \in V_i, x_2 \in V_j, i \neq j$:*
   Nodes x_1 and x_2 are in two different partitions, i and j, respectively, without the loss of generality. Processor p_i attempts to count the triangle executing the loop in Line 2-6 with $v = x_1$ and $u = x_2$. However, since $x_2 \notin V_i$, p_i sends N_{x_1} to p_j (Line 8). Processor p_j counts the triangle while executing the loop in Line 10-12 with $X = N_{x_1}$, and node x_3 appears in $S = N_{x_2} \cap N_{x_1}$(Line 2 in Fig. 2). This triangle can never be counted again in any processor, since $x_1 \notin N_{x_2}$ and $x_1, x_2 \notin N_{x_3}$.

Thus, in both cases, each triangle in G is counted once and only once. This completes our proof. ■

B. Computing An Estimation for Weight Function $f(v)$.

Our computation of balanced partitions in Section IV-B requires an estimation of the cost $f(v)$ which we compute from the following theorem.

Theorem 2. *The cost for counting triangles attributed to node $v \in V_i$ is given by $O\left(\sum_{u \in \mathcal{N}_v - N_v} (\hat{d_v} + \hat{d_u})\right)$.*

Proof: We have the following definitions from Section II and III, respectively: $\mathcal{N}_v = \{u : (u,v) \in E\}$ and $N_v = \{u : (u,v) \in E, v \prec u\}$. Then, it is easy to see,

$$u \in \mathcal{N}_v - N_v \Leftrightarrow v \in N_u. \tag{4}$$

To estimate the cost for counting triangles incident on node $v \in V_i$, consider the cost for counting triangles incident on all edges (v,u) such that $u \in \mathcal{N}_v$. There might be two cases:

1. *Case $u \in \mathcal{N}_v - N_v$:* This case implies $v \in N_u$ (by Eqn. 4). There might be two sub-cases:
   - If $u \in V_j$ for $j \neq i$, p_j sends N_u to p_i, and p_i counts triangle by computing $N_u \cap N_v$ (Fig. 2).
   - If $u \in V_i$, p_i counts triangle by computing $N_u \cap N_v$ while executing the loop in Line 2-6 in Fig. 3 for node u.

   Thus for both sub-cases p_i computes triangles incident on (v,u). All such nodes u impose a computation cost of $O\left(\sum_{u \in \mathcal{N}_v - N_v} (\hat{d_v} + \hat{d_u})\right)$ on p_i for node v.
2. *Case $u \in N_v$:* This case implies $v \in \mathcal{N}_u - N_u$ (by Eqn. 4) which is same as case 1 with u and v interchanged. By a similar argument of case 1, the imposed computation cost for such (v,u) is attributed to node u.

Thus the cost attributed to node v for counting triangles on all edges (v,u), for $u \in \mathcal{N}_v$, is $O\left(\sum_{u \in \mathcal{N}_v - N_v} (\hat{d_v} + \hat{d_u})\right)$. ■

Theorem 2 gives us the intended function $f(v) = \sum_{u \in \mathcal{N}_v - N_v} (\hat{d_v} + \hat{d_u})$ which we use in our partitioning step. We present an experimental evaluation comparing the best function presented in [15] with ours in Section VI.

C. Cost of Message Passing in Direct and Surrogate Approaches.

For $v \in V_i$, assume C_v is the set of edges $(v,u) \in E$ such that $u \notin V_i$, i.e., (v,u) is a cut edge. Next let $X_i = \bigcup_{v \in V_i} C_v$ is the set of all cut edges emanating from partition i, and $x_i = |X_i|$.

We present the communication cost incurred by *Direct* approach in the following lemma.

Lemma 1. *For Direct approach, the cost W_{dir} of exchanging messages by processor p_i is given by,*

$$O\left(\sum_{v \in V_i} \sum_{\substack{0 \le j \le P-1, \\ j \neq i}} \left[\sum_{\substack{u: u \in N_v, \\ u \in V_j}} (\hat{d_u} + 1) + \sum_{\substack{u: v \in N_u, \\ u \in V_j}} (\hat{d_v} + 1)\right]\right).$$

Proof: For a cut edge (v,u) with $v \in V_i$, the first term in each of the innermost summations accounts for the cost $O(|N_u|) = O(\hat{d_u})$ of receiving N_u (for $v \prec u$) or the cost $O(|N_v|) = O(\hat{d_v})$ of sending N_v (for $u \prec v$) and the second term for the cost $O(1)$ of a *request* message. ■

Let l_{vj} is the number of cut edges emanating from node v to some nodes u in partition j with $v \prec u$. Now the following lemma states the communication cost incurred by *Surrogate* approach.

Lemma 2. *For Surrogate approach, the cost W_{sur} of exchanging messages by processor p_i is given by,*

$$O\left(\sum_{v \in V_i} \sum_{\substack{0 \le j \le P-1, \\ j \neq i}} \left[\sum_{\substack{u: u \in N_v, \\ u \in V_j}} \frac{\hat{d_v}}{l_{vj}} + \sum_{\substack{u: v \in N_u, \\ u \in V_j}} \frac{\hat{d_u}}{l_{ui}}\right]\right).$$

Proof: For node $v \in V_i$, all cut edges (v,u) with $u \in V_j$ and $v \prec u$, N_v is sent to p_j at most once instead of l_{vj} times. Thus each such (v,u) is attributed to $\frac{1}{l_{vj}}$-fraction of the sending cost $O(\hat{d_v})$, which is accounted by the first innermost summation term. For all edges (v,u) with $u \prec v$, p_i receives N_u from p_j at most once instead of l_{ui} times. This *receiving* cost incurred on p_i for each such edge (v,u) is accounted by the second innermost summation term. ■

Comparison of costs. To get a crude estimate of how these two quantities (in Lemma 1 and 2) compare, we replace degrees $\hat{d_v}$, for all $v \in V_i$, by the average degree $\bar{d}$ and number of cut edges from v to partition j with $v \prec u$, l_{vj}, by l (an average over all l_{vj}). Then we get, the communication cost for *Surrogate approach*, W_{sur}:

$$\begin{aligned} & O\left(\sum_{v \in V_i} \sum_{\substack{0 \le j \le P-1, \\ j \neq i}} \left[\sum_{\substack{u: u \in N_v, \\ u \in V_j}} \frac{\bar{d}}{l} + \sum_{\substack{u: v \in N_u, \\ u \in V_j}} \frac{\bar{d}}{l}\right]\right) \\ = \; & O\left(\sum_{v \in V_i} \sum_{u:(v,u) \in C_v} \frac{\bar{d}}{l}\right) \text{[by the Defn. of } C_v\text{]} \\ = \; & O\left(\sum_{v \in V_i} \frac{|C_v|\bar{d}}{l}\right) = O\left(\frac{|X_i|\bar{d}}{l}\right) = O\left(\frac{x_i \bar{d}}{l}\right). \end{aligned}$$

The second last step follows from $X_i = \bigcup_{v \in V_i} C_v$. Similarly, we get, communication cost for *Direct approach*, W_{dir}:

$$O\Big(\sum_{(v,u)\in X_i} (\bar{d}+1) \Big) = O\left(|X_i|\bar{d} + |X_i|\right) = O\left(x_i\bar{d} + x_i\right).$$

Since $\frac{W_{dir}}{W_{sur}} > l$, the surrogate approach has at least l times smaller communication cost than that of the direct approach. As shown in Table II, the values of l range approx. from 4 to 10 for the networks we experimented on, and the surrogate approach reduces approx. 70% to 90% of messages.

TABLE III
NUMBER OF MESSAGES EXCHANGED IN DIRECT AND SURROGATE APPROACHES.

Networks	# of Messages		*Ratio*	l(avg)
	Direct	Surrogate		
Miami	$16,321,478$	$3,987,871$	4.09	3.89
web-Google	$493,488$	$99,221$	4.97	5.01
LiveJournal	$23,138,824$	$4,002,575$	5.78	5.43
Twitter	$247,821,246$	$25,341,984$	9.78	5.78
PA(10M, 100)	$99,436,823$	$8,092,340$	12.29	5.92

D. *Complexity of the Algorithm.*

Runtime Complexity. Computing balanced partition takes $O(\frac{m}{P} + P)$ time using an adaptation of the partitioning algorithm given in [15]. The worst case cost for counting triangles is $O(\sum_{v\in V_i}\sum_{u\in\mathcal{N}_v - N_v}(\hat{d}_u + \hat{d}_v))$ (Fig. 3). Let w_i be the communication cost incurred on p_i (as shown in Lemma 2). The summing up of counts require $O(\log P)$ time using MPI aggregation function. Thus, the time complexity of our parallel algorithm is, $O\big(\frac{m}{P} + P + \max_i w_i + \max_i \sum_{v\in V_i}\sum_{u\in\mathcal{N}_v - N_v}(\hat{d}_u + \hat{d}_v)\big)$.

Space Complexity. The size of the largest partition is $O(\max_i\{|V_i| + |E_i|\})$. Further, to store a single incoming or outgoing message containing N_v requires a space of $O(\max_{v\in V}|N_v|) = O(\hat{d}_{max})$. Thus, the total space complexity of our algorithm is $O(\max_i\{|V_i| + |E_i|\} + \hat{d}_{max})$. With the non-overlapping partitioning along with the load balancing scheme, we observe experimentally that the largest partition has approx. $\frac{m}{P}$ edges. A comparison of space complexity of other related algorithms is provided in Table IV. Our algorithm achieves up to $O(P^2)$-factor space efficiency over MapReduce based algorithms [12], [17], [18] and approx. $\bar{d}$-factor over the algorithm with overlapping partitioning [15].

VI. EXPERIMENTAL EVALUATION

We perform our experiments using a high performance computing cluster with 64 computing nodes (QDR InfiniBand interconnect), 16 processors (Sandy Bridge E5-2670, 2.6GHz) per node, memory 4GB/processor, and operating system CentOS Linux 6. The experimental evaluation of the performance our space-efficient parallel algorithm is presented below.

TABLE IV
A COMPARISON OF SPACE COMPLEXITY AMONG RELATED ALGORITHMS.

Algorithms	Space complexity	Remarks
Suri et al. [12]	$O(mP)$	size of Map output
Park et al. [17]	$O(mP)$	size of Map output
PATRIC [15]	$\Omega(\frac{xm}{P}), 1 \le x \le d$	for each processor
Our algo.	$O(\max_i\{|V_i| + |E_i|\} + \hat{d}_{max})$	$\approx \frac{m}{P}$ edges/proc.

Comparison with Previous Algorithms. The algorithm in [15] employs an overlapping partitioning and thus doesn't require message passing for counting triangles leading to a very fast algorithm (Table V). The non-overlapping partitioning employed by our algorithm achieves huge space saving over [15] (Table II), albeit requiring message passing for counting triangles. Our proposed communication approach (surrogate) reduces communication cost quite significantly leading to an almost similar runtime efficiency to [15]. In fact, our algorithm loses only ~20% runtime efficiency for the gain of a significant space efficiency of up to 2500%, thus allowing to work on larger networks.

A runtime comparison among other related algorithms for counting triangles in Twitter network is given in Fig. 4. Our algorithm is 35× faster than [12], 17× than [17], 7× than [18], and almost as fast as [15].

TABLE V
RUNTIME PERFORMANCE OF OUR ALGORITHM AND THE ALGORITHM IN [15]. WE USED 200 PROCESSORS FOR THIS EXPERIMENT.

Networks	Runtime			Triangles
	[15]	Direct	Surrogate	
web-BerkStan	0.10s	3.8s	0.14s	65M
Miami	0.6s	4.79s	0.79s	332M
LiveJournal	0.8s	5.12s	1.24s	286M
Twitter	9.4m	35.49m	12.33m	34.8B
PA(1B, 20)	15.5m	78.96m	20.77m	0.403M

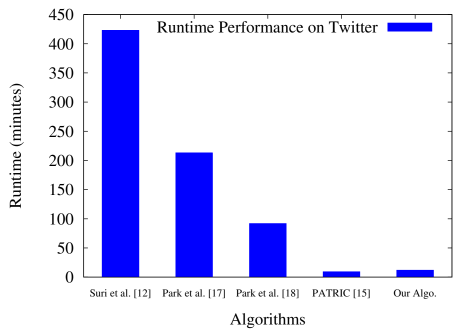

Fig. 4. Runtime reported by various algorithms for counting triangles in Twitter network.

Strong Scaling. Fig. 5 shows strong scaling (speedup) of our algorithm on Miami, LiveJournal, and web-BerkStan networks with both direct and surrogate approaches. Speedup factors with the surrogate approach are significantly higher than that of the direct approach due to its capability to reduce communication cost drastically. Our algorithm demonstrates an almost linear speedup to a large number of processors.

Further, our algorithm scales to a higher number of processors when networks grow larger, as shown in Fig. 6. This is, in fact, a highly desirable behavior since we need a large number of processors when the network size is large and computation time is high.

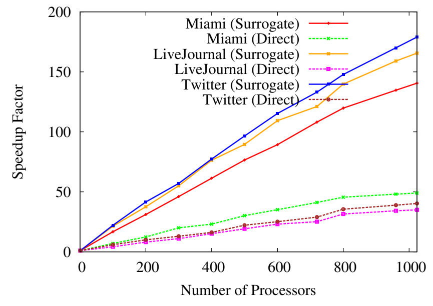

Fig. 5. Speedup factors of our algorithm with both direct and surrogate approaches.

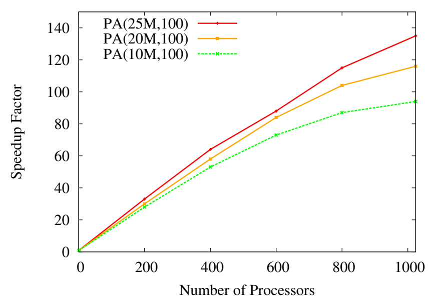

Fig. 6. Improved scalability of our algorithm with increasing network size.

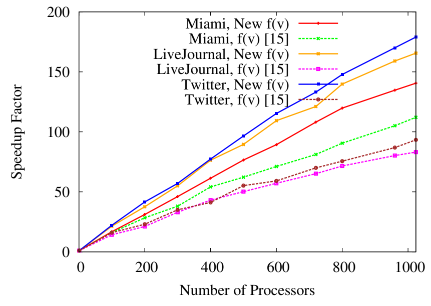

Fig. 7. Comparison of new estimation function of our algorithm and the best function of [15].

Effect of Estimation for f(v). We show the performance of our algorithm with new weight function (computed in Section V-B), $f(v) = \sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)$, and the best function $f(v) = \sum_{u \in N_v} (\hat{d}_v + \hat{d}_u)$ reported in [15]. As Fig. 7 shows, our algorithm with new weight function provides better speedup than that of [15]. Our new $f(v)$ estimates the computational cost more precisely and helps compute balanced partitions (Eqn. 3), which leads to better speedup.

Weak Scaling. Weak scaling of a parallel algorithm measures its ability to maintain constant computation time when the problem size grows proportionally with processors. The weak scaling of our algorithm is shown in Fig. 8. Since the addition of processors causes the overhead for exchanging messages to increase, the runtime of the algorithm increases slowly. However, as the change in runtime is rather slow (not drastic), our algorithm demonstrates a reasonably good weak scaling.

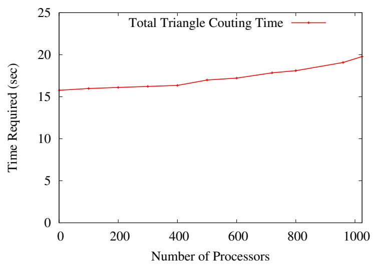

Fig. 8. Weak scaling of our algorithm, experiment performed on PA($t/10 * 1M, 50$) networks, t = number of processors used.

VII. Conclusion

We present a space-efficient parallel algorithms for counting *exact* number of triangles in massive networks. The algorithm employs non-overlapping partitions and reduces the space requirement significantly leading to the ability to work on larger networks. An efficient communication approach reduces message passing drastically to provide a fast algorithm. Our computation of a novel weight function for a parallel partitioning scheme adds further to the efficiency of the algorithm. We also provide a comprehensive theoretical analysis to justify the performance of the algorithm. We believe that for emerging massive networks, this algorithm will prove very useful.

Acknowledgments. This work has been partially supported by DTRA Grant HDTRA1-11-1-0016, DTRA CNIMS Contract HDTRA1-11-D-0016-0001, NSF NetSE Grant CNS-1011769, NSF SDCI Grant OCI-1032677, and NSF DIBBs Grant ACI-1443054.

References

[1] S. Chu and J. Cheng, “Triangle listing in massive networks and its applications,” in *KDD*, 2011.
[2] L. Becchetti *et al.*, “Efficient semi-streaming algorithms for local triangle counting in massive graphs,” in *KDD*, 2008.
[3] J. Eckmann and E. Moses, “Curvature of co-links uncovers hidden thematic layers in the world wide web,” *Proc. Natl. Acad. of Sci. USA*, vol. 99, no. 9, pp. 5825–5829, 2002.
[4] Z. Bar-Yosseff *et al.*, “Reductions in streaming algorithms, with an application to counting triangles in graphs,” in *Proc. of SODA*, 2002.
[5] A. Prat-Pérez *et al.*, “Shaping communities out of triangles,” in *CIKM*, 2012.
[6] A. Broder *et al.*, “Graph structure in the web,” *Computer Networks*, vol. 33, no. 16, pp. 309 – 320, 2000.
[7] H. Kwak *et al.*, “What is twitter, a social network or a news media?” in *WWW*, 2010.
[8] M. Girvan and M. Newman, “Community structure in social and biological networks,” *Proc. Natl. Acad. of Sci. USA*, vol. 99, no. 12, pp. 7821–7826, Jun. 2002.
[9] J. Ugander *et al.*, “The anatomy of the facebook social graph,” *CoRR*, vol. abs/1111.4503, 2011.
[10] T. Schank, “Algorithmic aspects of triangle-based network analysis,” Ph.D. dissertation, University of Karlsruhe, 2007.
[11] M. Latapy, “Main-memory triangle computations for very large (sparse (power-law)) graphs,” *Theor. Comput. Sci.*, vol. 407, pp. 458–473, 2008.
[12] S. Suri and S. Vassilvitskii, “Counting triangles and the curse of the last reducer,” in *WWW*, 2011.
[13] M. Jha *et al.*, “A space efficient streaming algorithm for triangle counting using the birthday paradox,” in *KDD*, 2013.
[14] C. Seshadhri, A. Pinar, and T. Kolda, “Triadic measures on graphs: the power of wedge sampling,” in *SDM*, 2013.
[15] S. Arifuzzaman, M. Khan, and M. Marathe, “PATRIC: A parallel algorithm for counting triangles in massive networks,” in *CIKM*, 2013.
[16] T. G. Kolda *et al.*, “Counting triangles in massive graphs with mapreduce,” *CoRR*, vol. abs/1301.5887, 2013.
[17] H.-M. Park and C.-W. Chung, “An efficient mapreduce algorithm for counting triangles in a very large graph,” in *CIKM*, 2013.
[18] H.-M. Park *et al.*, “Mapreduce triangle enumeration with guarantees,” in *CIKM*, 2014.
[19] Zoltan. http://www.cs.sandia.gov/zoltan/.
[20] D. Gleich and C. Seshadri, “Vertex neighborhoods, low conductance cuts, and good seeds for local community methods,” in *KDD*, 2012.
[21] C. Barrett *et al.*, “Generation and analysis of large synthetic social contact networks,” in *WSC*, 2009.
[22] A. Barabasi and R. Albert, “Emergence of scaling in random networks,” *Science*, vol. 286, pp. 509–512, 1999.
[23] Snap. http://snap.stanford.edu/.
[24] Twitter. http://an.kaist.ac.kr/~haewoon/release/twitter_social_graph.
[25] S. Aluru, “Teaching parallel computing through parallel prefix,” in *SC*, 2012.