Parallel Algorithms for Counting Triangles in Networks with Large Degrees

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 is an important problem in the analysis of complex networks. The number of triangles also has important applications in data mining. Existing distributed memory parallel algorithms for counting triangles are either Map-Reduce based or message passing interface (MPI) based and work with overlapping partitions of the given network. These algorithms are designed for very sparse networks and do not work well when the degrees of the nodes are relatively larger. For networks with larger degrees, Map-Reduce based algorithm generates prohibitively large intermediate data, and in MPI based algorithms with overlapping partitions, each partition can grow as large as the original network, wiping out the benefit of partitioning the network.**

In this paper, we present two efficient MPI-based parallel algorithms for counting triangles in massive networks with large degrees. The first algorithm is a space-efficient algorithm for networks that do not fit in the main memory of a single compute node. This algorithm divides the network into non-overlapping partitions. The second algorithm is for the case where the main memory of each node is large enough to contain the entire network. We observe that for such a case, computation load can be balanced dynamically and present a dynamic load balancing scheme which improves the performance significantly. Both of our algorithms scale well to large networks and to a large number of processors.

***Index Terms*—triangle-counting, parallel algorithms, massive networks, social networks, graph mining.**

I. Introduction

Counting triangles in a network is a fundamental and important algorithmic problem in the analysis of complex networks, and its solution can be used in solving many other problems such as the computation of clustering coefficient, transitivity, and triangular connectivity [1], [2]. Existence of triangles and the resulting high clustering coefficient in a social network reflect some common theories of social science, e.g., *homophily* where people become friends with those similar to themselves and *triadic closure* where people who have common friends tend to be friends themselves [3]. Further, triangle counting has important applications in graph mining such as detecting spamming activity and assessing content quality [4], uncovering the thematic structure of the web [5], and query planning optimization in databases [6].

Network is a powerful abstraction for representing underlying relations in large unstructured datasets. Examples include the web graph [7], various social networks, e.g., Facebook, Twitter [8], collaboration networks [9], infrastructure networks (e.g., transportation networks, telephone networks) and biological networks [10]. In the present world of technological advancement, we are deluged with data from a wide range of areas such as business and finance [11], computational biology [12] and social science. Many social networks have millions to billions of users [2], [13]. This motivates the need for space efficient parallel algorithms.

Counting triangles and related problems such as computing clustering coefficients has a rich history [13]–[17]. Much of the earlier algorithms are mainly based on matrix multiplication and adjacency matrix representation of the network. Matrix based algorithms [14] are not useful in the analysis of massive networks for their prohibitively large memory requirements. In the last decade many algorithms [15], [16], [18] have been developed using adjacency list representations. 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 graphs: streaming algorithms [19], [20], sparsification based algorithms [13], [18], external-memory algorithms [2], and parallel algorithms [17], [20]–[22]. 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 solve such a problem of a large running time by distributing computing tasks to multiple processors. Over the last couple of years, several parallel algorithms, either shared memory or distributed memory (MapReduce or MPI) based, have been proposed.

A shared memory parallel algorithm is proposed in [20] for counting triangles in a streaming setting, which is an approximation algorithm. In [17], two parallel algorithms for exact triangle counting using the MapReduce framework are presented. The first algorithm generates huge volumes of intermediate data, which are all possible 2-paths centered at each node. Shuffling and regrouping these 2-paths require a significantly large amount of time and memory. The second algorithm suffers from redundant counting of triangles. An improvement of the second algorithm is given in a very recent paper [23]. Although this algorithm reduces the redundant counting to some extent, the redundancy is not entirely eliminated. A MapReduce based parallelization of a wedge-based sampling technique [18] is proposed in [22], which is an approximation algorithm. MapReduce framework provides several advantages such as fault tolerance, abstraction of parallel computing mechanisms, and ease of developing a quick prototype or program. However, the overhead for doing so results in a larger runtime. On the other hand, MPI based systems provide the advantages of defining and controlling parallelism from a granular level, implementing application specific optimizations such as load balancing, memory and message optimization.

A recent paper [21] 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. Although this algorithm works very well on very sparse networks, it is not suitable for networks with large degrees. The size of the partitions grows quadratically with the increase of the degrees of the nodes. As a result, for a network with large degrees, the partitions can grow as large as the whole network, wiping out the benefit of partitioning the network.

We present two efficient MPI-based parallel algorithms for finding the exact number of triangles in networks with large degrees. The first algorithm is a space efficient algorithm for massive networks that do not fit in the memory of a single computing machine. This algorithm divides the network into non-overlapping partitions. Not only this algorithm is suitable for networks with large degrees, even for networks with smaller degrees, it can work on larger networks than that of the algorithm in [21] as the non-overlapping partitioning scheme leads to significantly smaller partitions. The second algorithm is for the case where the memory of each machine is large enough to contain the entire network. For such a case, we present a parallel algorithm with a dynamic load balancing scheme, which improves the performance significantly. Both of our algorithms scale well to large networks and to a large number of processors.

The rest of the paper is organized as follows. The preliminary concepts, notations and datasets are briefly described in Section II. In Section III, we discuss some background work on counting triangles. We present our parallel algorithms in Section IV and V and conclude in Section VI.

II. Preliminaries

Below are the notations, definitions, datasets, and experimental setup used in this paper.

Basic 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 graph is undirected. If $(u,v) \in E$, we say u and v are neighbors of 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 containing node v (in other words, triangles incident on v) is denoted by T_v. Notice that the number of triangles containing node v is same as the number of edges among the neighbors of v, i.e.,

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

We use K, M and B to denote thousands, millions and billions, respectively; e.g., 1B stands for one billion.

TABLE I
Dataset used in our experiments

Network	Nodes	Edges	Source
web-Google	0.88M	5.1M	SNAP [25]
web-BerkStan	0.69M	13M	SNAP [25]
Miami	2.1M	100M	[26]
LiveJournal	4.8M	86M	SNAP [25]
Twitter	42M	2.4B	[24]
PA(n, d)	n	$\frac{1}{2}nd$	Pref. Attachment

Datasets. We used both real world and artificially generated networks. A summary of all the networks is provided in Table I. Twitter data set is available at [24], and web-BerkStan, LiveJournal and web-Google networks are at SNAP library [25]. Miami [26] is a synthetic, but realistic, social contact network for Miami city. Note that the web-BerkStan, web-Google, LiveJournal and Twitter networks have very skewed degree distribution, i.e, some nodes have very large degrees. Artificial network PA(n, d) is generated using preferential attachment (PA) model [27] with n nodes and average degree d. PA(n, d) has power-law degree distribution, which is a very skewed distribution. Networks having some nodes with high degrees 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.

Computation Model. We develop parallel algorithms for message passing interface (MPI) based distributed-memory parallel systems, where each processor has its own local memory. The processors do not have any shared memory; one processor cannot access the local memory of another processor, and the processors communicate via exchanging messages using MPI.

We perform our experiments using a computing cluster (Dell C6100) with 30 computing nodes and 12 processors (Intel Xeon X5670, 2.93GHz) per node. The memory per processor is 4GB, and the operating system is SLES 11.1.

III. A Background on Counting Triangles

Our parallel algorithms are based on the state-of-the-art sequential algorithm for counting triangles. In this section, we describe the sequential algorithm and some background of our parallel algorithms.

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 edges among its neighbors, i.e., the number of pairs of neighbors

```
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.

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. The algorithms presented in [15], [16] uses a total ordering $\prec$ of the nodes to avoid duplicate counts of the same triangle. Any arbitrary ordering of the nodes, e.g., ordering the nodes based on their IDs, makes sure each triangle is counted exactly once – counts only one permutation among the six possible permutations. However, algorithms in [15], [16] incorporate an interesting node ordering based on the degrees of the nodes, with ties broken by node IDs, as defined as follows: $u \prec v \iff d_u < d_v$ or $(d_u = d_v$ and $u < v)$.

These algorithms are further improved in a recent paper [21] by a simple modification. The algorithm [21] defines $N_x \subseteq \mathcal{N}_x$ as the set of neighbors of x having a higher order $\prec$ than x. For an edge (u,v), the algorithm stores v in N_u if $u \prec v$, and u in N_v, otherwise, i.e., $N_v = \{u : (u,v) \in E, v \prec u\}$. Then the triangles containing node v and any $u \in N_v$ can be found by set intersection $N_u \cap N_v$. Now let, $\hat{d_v} = |N_v|$ be 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. The above state-of-the-art sequential algorithm is presented in Fig. 1. Our parallel algorithms are based on this sequential algorithm.

B. Related Parallel Algorithms

In Section I we discussed a few parallel algorithms [17], [21], [22] which deal with massive networks. The most relevant to our work is the parallel algorithm presented in [21]. The algorithm [21] divides the input graph into a set of overlapping partitions. Let P be the number of processors used in the computation. A partition (subgraph) G_i is constructed as follows. Set of nodes V is partitioned into P disjoint subsets $V_0^c, V_1^c, \ldots, V_{P-1}^c$, such that, for any j and k, $V_j^c \cap V_k^c = \emptyset$ and $\bigcup_k V_k^c = V$. Then, a set V_i is constructed containing all nodes in V_i^c and $\bigcup_{v \in V_i^c} N_v$. The partition G_i is a subgraph of G induced by V_i. Each processor i works on partition G_i. The node set V_i^c and N_v for $v \in V_i^c$ constitute the disjoint (non-overlapping) portion of the partition G_i. The node set $V_i - V_i^c$ and N_u for $u \in V_i - V_i^c$ constitute the overlapping portion of the partition G_i.

This scheme works very well for sparse networks; however, when the average degrees of networks increase, each partition can be very large. Assuming an average degree $\bar{d}$ of the network, the algorithm in [21] has a space requirement of $\Omega(n\bar{d}/P)$ for storing disjoint portion of the partition. The space needed for storing the whole partition is $\Omega(xn\bar{d}/P)$ where $1 \le x \le \bar{d}$ which can be very large. In many real world networks average degrees are large, e.g., Facebook users have an average of 190 friends [28].

We observe that even for a sparse network with small average degree, if there are few nodes with large degrees, say $O(n)$, some partitions can be very large. For example, consider a node v with degree $n-1$, where n is the number nodes in the network, the partition containing node v will be equal to the whole network. Some real networks have very skewed degree distributions where some nodes have very large degrees.

In the first algorithm presented in this paper, we divide the input networks into non-overlapping partitions. Each partition is almost equal and has approximately m/P edges, which can be significantly smaller (in some cases, as much as $\bar{d}$ times smaller) than the overlapping partition in [21]. As a result, our algorithm can work with networks with large degrees. Since the partitions are significantly smaller, even for a network with smaller degree the algorithm can work on larger networks than [21]. Table II shows the space requirement (in MB) of our algorithm and the algorithm in [21].

TABLE II
MEMORY USAGE OF OUR ALGOIRTHM AND [21] FOR STORING THE LARGEST PARTITION. NUMBER OF PARTITIONS USED IS 100.

Networks	**Memory (MB)**		**Avg. Degree**
	Our algo.	Algo. in [21]	
Miami	10.63	36.56	47.6
web-Google	1.49	5.65	11.6
LiveJournal	9.41	22.15	18
Twitter	265.82	6876.25	57.14
PA(10M, 100)	121.11	2120.94	100

Now consider the case that the size of an overlapping partition is equal (or almost equal) to the whole network and each computing machine has enough space available for storing the partition, and consequently the whole network. For such a case, we observe that a dynamic load balancing scheme can make the computation even faster and present an efficient parallel algorithm with dynamic load balancing, which is significantly faster than the algorithm in [21]. We present our parallel algorithms in the following sections.

IV. A SPACE EFFICIENT PARALLEL ALGORITHM

In this section, we present our space-efficient parallel algorithm for counting triangles. At first, we present the overview of the algorithm. The detailed description of the algorithm follows thereafter.

A. Overview of the Algorithm

Let P be the number of processors used in the computation. The algorithm partitions the input network $G(V,E)$ into a set of P subgraphs G_i. Informally, the subgraph G_i is constructed as follows: set of nodes V is partitioned into P mutually disjoint subsets V_ks, for $0 \le k \le P-1$, such that, $\bigcup_k V_k = V$. Node set V_i, along with N_v for all $v \in V_i$, constitutes the subgraph G_i (Definition 1). Each processor is assigned one such subgraph (partition) G_i. Now, to count triangles incident on $v \in V_i$, processor i needs N_u for all $u \in N_v$. If $u \in V_i$, processor i counts triangles incident on (v, u) by computing $N_u \cap N_v$. However, if $u \in V_j$, $j \neq i$, then N_u resides in processor j. In such a case, processor i and j exchange message(s) to count triangles adjacent to edge (v, u). There are several non-trivial issues involving this exchange of messages, which can crucially affect the performance of the algorithm. We devise an efficient communication scheme to reduce the communication overhead and improve the performance of the algorithm. Once all processors complete the computation associated to respective partitions, the counts from all processors are aggregated.

Definition 1. ***A non-overlapping partition:*** *Given a graph $G(V, E)$ and an integer $P \geq 1$ denoting the number of partitions, a non-overlapping partition for our algorithm, denoted by $G_i(V'_i, E'_i)$, is a subgraph of G which is constructed as follows.*

- *V is partitioned into P disjoint subsets V_is of consecutive nodes, such that, for any j and k, $V_j \cap V_k = \emptyset$ and $\bigcup_k V_k = V$*
- *$V'_i = V_i \cup \{v : v \in N_u, u \in V_i\}$*
- *$E'_i = \{(u, v) : u \in V_i, v \in N_u\}$*

The subgraphs (partitions) G_is are non-overlapping– each edge $(u, v) \in E$ resides in one and only one partition. For any j and 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.

Our algorithm exchanges two types of messages– data message and completion notifier. A message is denoted by $\langle t, X\rangle$ where $t \in \{data, completion\}$ is the type of the message and X is the actual data associated with the message. For a data message ($t = data$), X refers to a neighbor list, whereas for a completion notifier ($t = completion$), the value of X is disregarded. We describe the details of our algorithm in the following subsections.

B. Computing Partitions

While constructing partitions G_i, set of nodes V is partitioned into P disjoint subsets V_is (Definition 1). How the nodes in V are distributed among the sets V_i for all processors i crucially affect the performance of the algorithm. 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 the cost for counting triangles incident on a node $v \in V$(cost for executing Line 7-10 in Fig. 1). We need to compute P disjoint partitions of node set V such that for each partition V_i,

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

We realize that the parallel algorithm for computing balanced partitions of V proposed in [21] is applicable to our problem. The paper [21] proposed several estimations for $f(v)$, among which $f(v) = \sum_{u \in N_v} (\hat{d_v} + \hat{d_u})$ is 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 different function $f(v)$ to estimate the computational cost of our algorithm more precisely (in Section IV-F). We use this function to compute balanced partitions. Once all P partitions are computed, each processor is assigned one such partition.

C. Counting Triangles with An Efficient Communication Scheme

As we discussed in Section IV-A, processor i stores N_v for all $v \in V_i$. However, to compute triangles incident on $v \in V_i$, N_u for all $u \in N_v$ are also required. Now, if $u \in V_j$, $j \neq i$, then N_u resides in processor j. A simple approach to resolve the issue is as follows.

(The Direct Approach) *Processor i requests processor j for N_u. Upon receiving the request, processor j sends N_u to processor i. Processor i counts triangles incident on the edge (v, u) by computing $N_v \cap N_u$.*

The direct approach has a high communication overhead due to exchanging redundant messages. Assume $u \in N_v$ and $u \in N_w$ for $v, w \in V_i$. Then processor i sends two separate requests for N_u to processor j while computing triangles incident on v and w, respectively. In response to the above requests, processor j sends same message N_u to processor i twice. Such redundant messages increases the communication overhead leading to poor performance of the algorithm.

One way to eliminate redundant messages is that instead of requesting N_us multiple times, processor i stores them in memory after fetching them for the first time. Before sending a request, processor i performs a lookup into the already fetched lists of N_us. A new request for N_u is made only when it is not already fetched. However, the space requirement for storing all N_us along with G_i is same as that of storing an overlapping partition. This diminishes our original goal of a space-efficient algorithm.

Another way to eliminate message redundancy is as follows. When a neighbor list N_u is fetched, processor i scans through G_i to find all nodes $v \in V_i$ such that $u \in N_v$. Processor i, then, performs all computations related to N_u (i.e., $N_v \cap N_u$). Once these computations are done, the neighbor list N_u is never needed again and can be discarded. However, scanning through the whole partition G_i for each fetched list N_u might be very expensive, which is even more expensive than the direct approach with redundant messages.

Since all of the above techniques compromise either runtime or space efficiency, we introduce another communication scheme for counting triangles involving nodes $v \in V_i$ and $u \in V_j$, such that $u \in N_v$.

(The Surrogate Approach) *Processor i sends N_v to processor j. Processor j scans N_v to find all nodes $u \in N_v$ such that $u \in V_j$. For all such nodes u, processor j counts triangles incident on edge (u, v) by performing the operation $N_v \cap N_u$.*

The surrogate approach eliminates the exchange of redundant messages: while counting triangles incident on a node

$v \in V_i$, processor i may find multiple nodes $u \in N_v$ such that $u \in V_j$. Processor i sends N_v to processor j when such a node u is encountered for the first time. Since processor j counts triangles incident on edge (u, v) for all such nodes u, processor i does not send N_v again to processor j.

To implement the above strategy for eliminating redundant messages, processor i needs to keep record of which processors it has already sent N_v to, for a node $v \in V_i$. This is done using a single variable *LastProc* which records the last processor a neighbor list N_v is sent to. The variable is initialized to a negative value. When processor i encounters a node $u \in N_v$ such that $u \in V_j$, it checks the value of *LastProc*. If *LastProc* $\neq j$, processor i sends N_v to processor j and set *LastProc* $= j$. Otherwise, the node u is ignored, meaning it would be redundant to send N_v. Once all nodes $u \in N_v$ are checked, the variable *LastProc* is again reset to a negative value. It is easy to see that since V_i is a set of consecutive nodes, and all neighbor lists N_v are sorted, all nodes $u \in N_v$ such that $u \in V_j$ reside in N_v in consecutive positions. Thus, using the variable *LastProc* redundant messages are detected correctly and eliminated without compromising any of execution and space efficiency. This capability of surrogate approach is crucial in the runtime performance of the algorithm, as shown experimentally in Section IV-H.

The pesudocode for counting triangles for an incoming 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 processor i to count triangles for incoming message $\langle data, X\rangle$.

D. Termination

Once a processor i completes the computation associated with all $v \in V_i$, it broadcasts a completion notifier $\langle completion, X\rangle$. However, it cannot terminate its execution since other processors might send it data messages $\langle data, X\rangle$ for counting triangles (as in Fig. 2). When processor i receives completion notifiers from all other processors, aggregation of counts from all processors is performed using MPI aggregation function, and the execution terminates.

The complete pseudocode of our space efficient parallel algorithm for counting triangles using surrogate approach is presented in Fig. 3.

E. 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.*

```
1: $T_i \leftarrow 0$   //$T_i$ is processor $i$'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 proc. $j$ 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 $\langle completion, X\rangle$
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 i executes Line 1-22. After that, they are synchronized and the aggregation is performed (Line 24-26).

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 as shown below.

1. *Case $x_1, x_2 \in V_i$:*
   Nodes x_1 and x_2 are in the same partition i. Processor 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 i attempts to count the triangle executing the loop in Line 2-6 with $v = x_1$ and $u = x_2$. However, since $x_2 \notin V_i$, processor i sends N_{x_1} to processor j (Line 8). Processor 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. □

F. Computing An Estimation for $f(v)$

As discussed in Section IV-B, computing balanced partitions requires an estimation of the cost $f(v)$ for counting triangles incident on node v. We compute a new function for estimating $f(v)$ which captures the computing cost of our algorithm more precisely, as follows.

Set of neighbors $\mathcal{N}_v$ and N_v are defined in Section II and III, respectively, as $\mathcal{N}_v = \{u : (u,v) \in E\}$ and $N_v = \{u : (u,v) \in E, v \prec u\}$. It is easy to see, $u \in \mathcal{N}_v - N_v \Leftrightarrow v \in N_u$.

To estimate the cost for counting triangles incident on node $v \in V_i$, we consider the cost for counting triangles incident on 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 $v \in N_u$. There might be two sub-cases:
   - If $u \in V_j$ for $j \neq i$, processor j sends N_u to processor i, and processor i counts triangle by computing $N_u \cap N_v$ (Fig. 2).
   - If $u \in V_i$, processor 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 processor i computes triangles incident on (v,u). All such nodes u impose a computation cost of $\sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)$ on processor i for node v.
2. *Case $u \in N_v$:*
   This case implies $v \in \mathcal{N}_u - N_u$ 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 estimated cost attributed to node v for counting triangles on all edges (v,u), such that $u \in \mathcal{N}_v$, is $\sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)$. This 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 estimation function presented in [21] with ours in Section IV-H.

G. Runtime and Space Complexity

The runtime and space complexity of our algorithm are presented below.

Runtime Complexity: The runtime complexity of our algorithm is the sum of costs for computing partitions, counting triangles, exchanging N_vs, and aggregating counts from all processors. Computing balanced partition takes $O(m/P + P \log P)$ time using the scheme presented in [21]. For the algorithm given in Fig. 3, the worst case cost for counting triangles is $O(\sum_{v \in V_i} \sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_u + \hat{d}_v))$. Further, the communication cost incurred on a processor is $O(m/P)$ in the worst case. The aggregation of counts require $O(\log P)$ time using MPI aggregation function. Thus, the time complexity of our parallel algorithm is,

$$O(m/P + P \log P + \max_i \sum_{v \in V_i} \sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_u + \hat{d}_v))$$

Space Complexity: The space complexity of our algorithm depends on the size of the partitions and messages exchanged among processors. The size of the partition is $O(\max_i\{|V_i'| + |E_i'|\})$. Now our algorithm stores only a single incoming or outgoing message at a time, and each data message contains N_v. Thus, the space requirement for storing messages is $O(\max_{v \in V} |N_v|) = O(d_{max})$. Taking those two into account, the overall space complexity is $O(\max_i\{|V_i'| + E_i'\} + d_{max})$.

H. Performance

In this section, we present the experimental evaluation of the performance our space-efficient parallel algorithm.

Strong Scaling: Strong scaling of a parallel algorithm shows how much speedup a parallel algorithm gains as the number of processors increases. Fig. 4 shows strong scaling 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. The high communication overhead of direct approach due to redundant messages leads to poor speedup whereas the surrogate approach eliminates message redundancy without introducing any computational overhead leading to better performance.

Effect of Estimation for f(v): We show the performance of our algorithm with new estimation function (computed in Section IV-F), $f(v) = \sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)$, and the best estimation function $f(v) = \sum_{u \in N_v} (\hat{d}_v + \hat{d}_u)$ of [21]. As Fig. 5 shows, our algorithm with new estimation function provides better speedup than that with [21]. Miami network has a comparable performance with both functions since it has a relatively even degree distribution and both functions provide almost the same estimation. However, for networks with skewness in degrees (LiveJournal and web-BerkStan), our new function estimates the computational cost more precisely and provides significantly better speedup.

Scaling with Processors and Network Size: We show how our algorithm scales with increasing network size and number of processors, and compare the results with the algorithm in [21]. Our algorithm scales to a higher number of processors when networks grow larger, as shown in Fig. 14. This is, in fact, a highly desirable behavior of our parallel algorithm since we need a large number of processors when the network size is large and computation time is high. Now as compared to [21], the speedup and scaling of our algorithm are a little smaller since our algorithm has a higher communication overhead. However, this difference in scaling is very small and both algorithms perform comparably.

Memory Scaling: We compare the space requirement of our algorithm and [21] with networks with increasing average degrees. For this experiment, we use $PA(10M, d)$ networks with average degree d varying from 10 to 100. As shown in Fig. 7, our algorithm shows a very linear (and slow) increase of space requirement whereas for [21] the space requirement increases very rapidly. Our algorithm divides the network into non-overlapping partitions and hence has a much smaller space

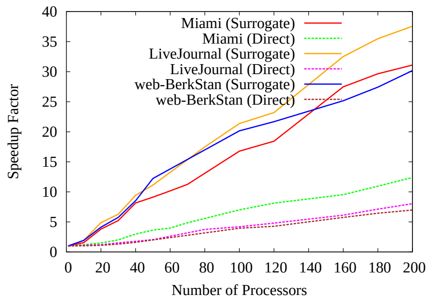

Fig. 4. Speedup factors of our algorithm with both direct and surrogate approaches.

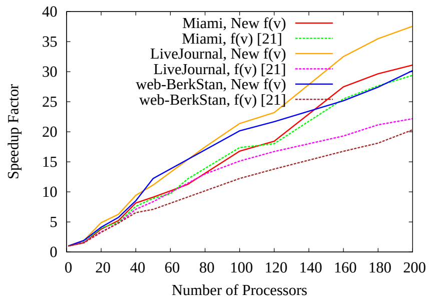

Fig. 5. Speedup factors of our algorithm with new estimation function and the best function of [21].

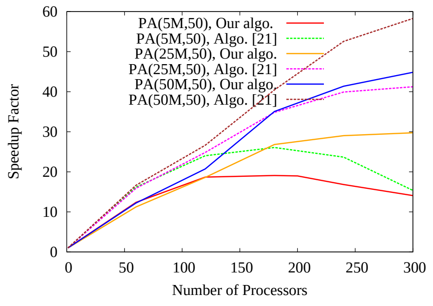

Fig. 6. Improved scalability of our algorithm with increasing network size.

requirement as discussed in Section III. For the same reason, space requirement of our algorithm for storing a partition reduces rapidly with additional processors, as shown in Fig. 8.

Comparison of Runtime with Previous Algorithms: We present a comparison of runtime of our algorithm with the algorithm in [21] in Table III. Since our algorithm exchanges messages for counting triangles, it has a higher runtime than [21]. Runtime with the direct approach is relatively high due to message redundancy. However, our algorithm with surrogate approach improves the runtime quite significantly, and the performance is quite comparable to [21].

TABLE III
RUNTIME PERFORMANCE OF OUR ALGORITHM AND THE ALGORITHM IN [21]. WE USED 200 PROCESSORS FOR THIS EXPERIMENT.

Networks	**Runtime**			**Triangles**
	[21]	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

Weak Scaling: Weak scaling of a parallel algorithm measures its ability to maintain constant computation time when the problem size grows proportionally with the number of processors. The weak scaling of our algorithm is shown in Fig. 9. Although the problem size per processor remains same, the addition of processors causes the overhead for exchanging messages to increase. Thus with increasing number of processors, runtime of our algorithm increases slowly. Since the change in runtime is not very drastic, rather slow, the weak scaling of the algorithm is good.

V. A FAST PARALLEL ALGORITHM WITH DYNAMIC LOAD BALANCING

In this section, we present our parallel algorithm for counting triangles with an efficient dynamic load balancing scheme. First, we provide an overview of the algorithm, and then a detailed description follows.

A. Overview of the Algorithm

We assume that each computing node has enough memory to store the whole network. The computation of counting triangles in the network is distributed among processors. We refer the computation assigned to and performed by a processor as a task. For the convenience of future discussion, we present the following definitions related to computing tasks.

Definition 2. ***Task:*** *Given a graph $G = (V, E)$, a task denoted by $\langle v, t\rangle$, refers to counting triangles incident on nodes $v \in \{v, v+1, \ldots, v+t-1\} \subseteq V$. The task referring to counting triangles in the whole network is $\langle 0, n\rangle$.*

Definition 3. ***An atomic task:*** *A task $\langle v, 1\rangle$ referring to counting triangles incident on a single node v is an atomic task. An atomic task can not be further divided.*

Definition 4. ***Task size:*** *Let, $f : V \to \mathcal{R}$ be a cost function such that $f(v)$ denotes some measure of the cost for counting triangles on node v. We define the size $S(v, t)$ of a task $\langle v, t\rangle$ as follows.*

$$S(v, t) = \sum_{i=0}^{t-1} f(v + i).$$

We consider the cost functions $f(v) = 1$ and $f(v) = d_v$ since those are known for all $v \in V$ and have no computational overhead. Using a computationally expensive function for representing the cost for counting triangles might lead to poor performance of the algorithm.

Now in a static load balancing scheme, each processor works on a pre-computed partition. Since the partitioning is based on estimated computing cost which might not equal to the actual computing cost, some processors will remain idle after finishing computation ahead of others. Our algorithm employs a dynamic load balancing scheme to reduce idle time of processors leading to improved performance. The algorithm divides the total computation into several tasks and assign them dynamically. How and when to assign a task require communication among processors. The scheme for Communication and decision about task granularity are crucial to the performance of our algorithm.

In the following subsection, we describe an efficient algorithm for dynamic load balancing.

B. An Efficient Dynamic Load Balancing Scheme

We design a dynamic load balancing scheme with a dedicated processor for coordinating balancing decisions. We distinguish this processor as the *coordinator* and the rest

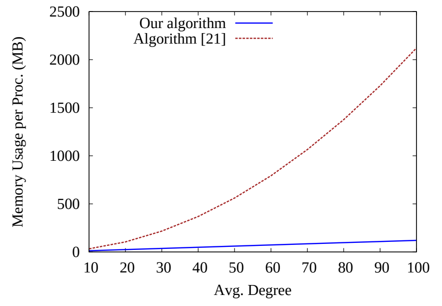

Fig. 7. Increase of space requirement of our algorithm and algorithm [21] with increasing average degree of networks.

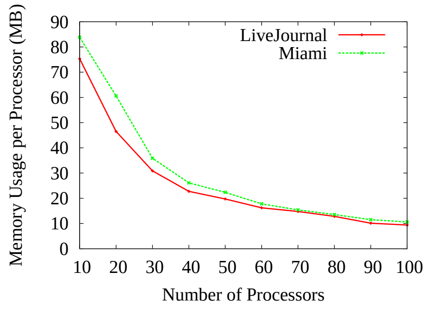

Fig. 8. Memory scalability of our algorithm with increasing number of processors for Miami and LiveJournal networks.

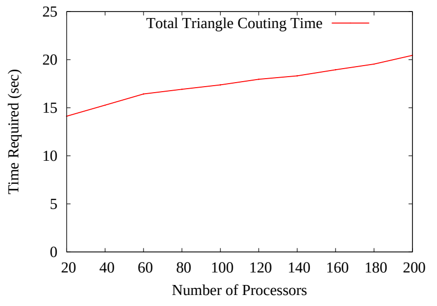

Fig. 9. Weak scaling of our algorithm. The experiment is performed on PA($P/10*1M, 50$) networks where P is the number of processors used.

as *workers*. The *coordinator* assigns tasks, receives notifications and re-assigns tasks to idle workers, and *workers* are responsible for actually performing *tasks*. At the beginning, each worker is assigned an initial task. Once any worker i completes its current task, it sends a request to *coordinator* for an additional task. From the available un-assigned tasks, *coordinator* assigns a new task to worker i.

The coordinate may divide the computation into tasks of equal size and assign them dynamically. However, the size of tasks is a crucial determinant of the performance of the algorithm. Assume time required by some worker to compute the last completed task is q. The amount of time a worker remains idle, denoted by a continuous random variable X, can be assumed to be uniformly distributed over the interval $[0, q]$, i.e., $X \sim U(0, q)$. Since $E[X] = q/2$, a worker remains idle for $q/2$ amount of time on average. If the size $S(v, t)$ of tasks $\langle v, t\rangle$ is large, time q required to complete the last task becomes large, and consequently, idle time $q/2$ also grows large. In contrast, if $S(v, t)$ decreases, the idle time is expected to decrease. However, if $S(v, t)$ is very small, total number of tasks becomes large, which increases communication overhead for task requests and re-assignments.

Therefore, instead of keeping the size of tasks $S(v, t)$ constant throughout the execution, our algorithm adjusts $S(v, t)$ dynamically, initially assigning large tasks and then gradually decreasing them. In particular, initially half of the total computation $\langle 0, n\rangle$ is assigned among the workers in tasks of almost equal sizes. That is, a total of $\langle 0, t'\rangle$ task, such that $S(0, t') = \frac{1}{2} S(0, n)$, is assigned initially, and the remaining computations $\langle t', n - t'\rangle$ are assigned dynamically with the granularity of tasks decreasing gradually. Next, we describe the steps of our dynamic load balancing scheme in detail.

Initial Assignment. The set of $(P - 1)$ initial tasks corresponds to counting triangles on nodes $v \in \{0, 1, \ldots, t' - 1\}$ such that $S(0, t') \approx S(t', n - t')$. Thus we need to find node t' which divides the set of nodes V into two disjoint subsets in such a way that $\sum_{v=0}^{t'-1} f(v) \approx \sum_{v=t'}^{n-1} f(v)$, given $f(v)$ for each $v \in V$. Now if we compute sequentially, it takes $O(n)$ time to perform the above computations. However, we observe that a parallel algorithm for computing balanced partitions of V proposed in [21] can be used to perform the above computation which takes $O(n/P + \log P)$ time. Once t' is determined, the task $\langle 0, t'\rangle$ is divided into $(P-1)$ tasks $\langle v, t\rangle$, one for each worker, in almost equal sizes.

$$S(v, t) = \frac{1}{P - 1} \sum_{v \in 0}^{t'-1} f(v). \tag{1}$$

That is, set of nodes $\{0, 1, \ldots, t' - 1\}$ is divided into $(P - 1)$ subsets such that for each subset $\{v, v + 1, \ldots, t - 1\}$, $\sum_{i=0}^{t-1} f(v + i) \approx \frac{1}{P-1} \sum_{v \in 0}^{t'-1} f(v)$. This computation can also be done using the parallel algorithm [21] mentioned above.

Note that all P processors work in parallel to determine initial tasks. Since the initial assignment is deterministic, workers pick their respective tasks $\langle v, t\rangle$ without involving the coordinator.

Dynamic Re-assignment. Once any worker completes its current task and becomes idle, the *coordinator* assigns it a new task dynamically. This re-assignment is done in the following steps.

- The coordinator divides the un-assigned computations $\langle t', n - t'\rangle$ into several tasks and stores them in a queue W. How the coordinator decides on the size $S(v, t)$ of each task $\langle v, t\rangle$ will be described shortly.
- When any worker i finishes its current task and becomes idle, it sends a task request $\langle i\rangle$ to the coordinator.
- If $W \neq \emptyset$, the coordinator picks a task $\langle v, t\rangle \in W$, and assigns it to worker i.

Our algorithm decreases the size $S(v, t)$ of each dynamically assigned tasks gradually for the reasons discussed at the beginning of this subsection. Let, V' be the set of nodes remaining to be assigned as tasks. Since at every new assignment V' decreases our algorithm uses V' to dynamically adjust task sizes. This is done using the following equation.

$$S(v, t) = \frac{1}{P - 1} \sum_{v \in V'} f(v). \tag{2}$$

Note that the size $S(v, t)$ of a dynamically assigned task $\langle v, t\rangle$ decreases at every new assignment. By the definition of atomic task (in definition 3) we have a finite number of tasks. When the coordinator has no more unassigned tasks, i.e., $W = \emptyset$, it sends a special termination message $\langle terminate\rangle$ to the requesting worker. Once the coordinator completes sending termination messages to all workers, it aggregates counts of triangles from all workers, and the algorithm terminates.

C. Counting Triangles

Once a processor i has an assigned task $\langle v, t \rangle$, it uses the algorithm presented in Fig. 10 to count the triangles incident on nodes $v \in \{v, v+1, \ldots, v+t-1\}$.

```
1: Procedure CountTriangles($\langle v, t \rangle$) :

2: $T \leftarrow 0$    //$T$ is the count of triangles
3: for $v \in \{v, v+1, \ldots, v+t-1\}$ do
4:     for $u \in N_v$ do
5:         $S \leftarrow N_v \cap N_u$
6:         $T \leftarrow T + |S|$
7: return $T$
```

Fig. 10. A procedure executed by processor i to count triangles corresponding to the task $\langle v, t \rangle$.

The complete pseudocode of our algorithm for counting triangles with an efficient dynamic load balancing scheme is presented in Fig. 11.

```
1: All processors initially do the following:
2: Determine initial tasks (see discussion of Eqn. 1)
3:
4: The coordinator does the following:
5: $W \leftarrow \emptyset$
6: for all remaining tasks $\langle v, t \rangle$ do
7:     EnQueue ($W$, $\langle v, t \rangle$ )
8: while $W$ is not $\emptyset$ do
9:     Receive task requests $\langle i \rangle$
10:    $\langle v, t \rangle \leftarrow$ DeQueue ($W$)
11:    Send message $\langle v, t \rangle$ to worker $i$
12: Send $\langle terminate \rangle$ to proc. $i$ for requests $\langle i \rangle$
13:
14: Each worker $i$ does the following:
15: $T_i \leftarrow 0$
16: $T_i \leftarrow T_i +$ CountTriangles$(v, t)$   //for initial task

17: while worker $i$ is idle do
18:    Send message $\langle i \rangle$ to coordinator
19:    Receive message $M$ from coordinator
20:    if $M$ is $\langle terminate \rangle$ then
21:        Stop execution
22:    else if $M$ is a task $\langle v, t \rangle$ then
23:        $T_i \leftarrow T_i +$ CountTriangles$(v, t)$
24:
25: MpiBarrier
26: Find Sum $T \leftarrow \sum_i T_i$ using MpiReduce
27: return $T$
```

Fig. 11. An algorithm for counting triangles with dynamic load balancing.

D. Correctness of the Algorithm

We establish the correctness of our algorithm as follows. Consider a triangle (x_1, x_2, x_3) with $x_1 \prec x_2 \prec x_3$, without the loss of generality. Now, the triangle is counted only when $x_1 \in \{v, v+1, \ldots, v+t-1\}$ for some task $\langle v, t \rangle$. The triangle is never counted again since $x_1 \notin N_{x_2}$ and $x_1, x_2 \notin N_{x_3}$ by the construction of N_x (Line 1-3 in Fig. 1).

E. Performance

In this section, we present the experimental evaluation of our parallel algorithm for counting triangles with dynamic load balancing.

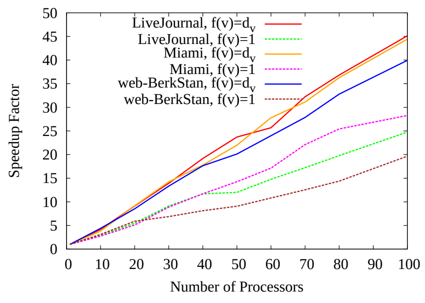

Fig. 12. Speedup factors of our algorithm on Miami, LiveJournal and web-BerkStan networks with both $f(v) = 1$ and $f(v) = d_v$ cost functions.

Strong Scaling: We present the strong scaling of our algorithm on Miami, LiveJournal, and web-BerkStan networks with both cost functions $f(v) = 1$ and $f(v) = d_v$ in Fig. 12. Speedup factors are significantly higher with the function $f(v) = d_v$ than with $f(v) = 1$. The function $f(v) = 1$ refers to equal cost of counting triangles for all nodes whereas the function $f(v) = d_v$ relates the cost to the degree of v. Distributing tasks based on the sum of degrees of nodes (Eqn. 1 and 2) reduces the effect of skewness of degrees and makes tasks more balanced leading to higher speedups. Our next experiments will be based on cost function $f(v) = d_v$.

Comparison with Previous Algorithms: We compare the runtime of our parallel algorithm with the algorithm in [21] on a number of real and artificial networks. As shown in Table IV, our algorithm is is more than 2 times faster than [21] for all these networks. The algorithm in [21] is based on static partitioning whereas our algorithm employs a dynamic load balancing scheme to reduce idle time of processors leading to improved performance.

TABLE IV
Runtime performance of our algorithm and algorithm [21].

Networks	Runtime		Triangles
	[21]	Our algo.	
web-BerkStan	0.10s	0.041s	65M
LiveJournal	0.8s	0.384s	286M
Miami	0.6s	0.301s	332M
PA(20M, 50)	11.85s	5.241s	0.028M

Effect of Dynamic Adjustment of Task Granularity: We show how the granularity of tasks affects idle time of worker processors for Miami and LiveJournal networks. As Fig. 13 shows, with tasks of static size, some processors have very large idle times. However, when task granularity is dynamically adjusted, idle times of processors become very small leading to balanced load among processors. This consequently improves the runtime performance of the algorithm.

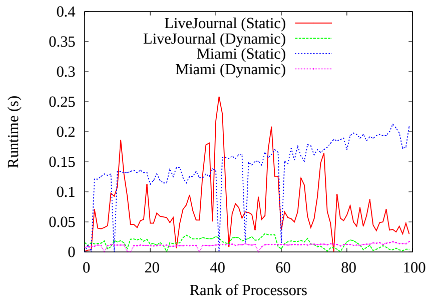

Fig. 13. Idle time of processors with both static and dynamic task granularity. When task granularity is adjusted dynamically idle times decrease significantly leading to smaller runtime.

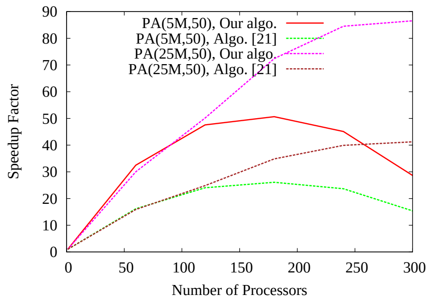

Fig. 14. Improved scalability of our algorithm with increasing network size. Further, our algorithm achieves higher speedups than [21].

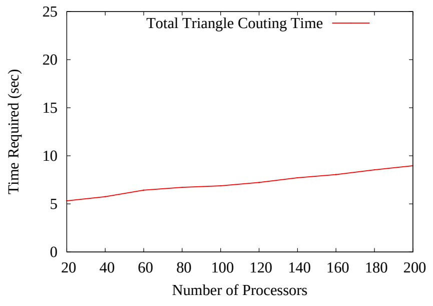

Fig. 15. Weak scaling of our algorithm. A very slowly increasing runtime suggests a good weak scaling of the algorithm.

Scaling with Processors and Network Size: Our algorithm scales to a higher number of processors with increasing size of networks, as shown in Fig. 14. Scaling of our algorithm with number of processors is very comparable to that of [21]. However, our algorithm achieves significantly higher speedup factors than [21].

Weak Scaling: The weak scaling of our algorithm is shown in Fig. 15. With the addition of processors, communication overhead increases since idle workers exchange messages with the coordinator for new tasks. However, since the overhead for requesting and assigning tasks is very small, the increase of runtime with additional processors is very slow. Thus, the weak scaling of our algorithm is very good.

VI. Conclusion

We present two parallel algorithms for counting triangles in networks with large degrees. Our space-efficient algorithm works on networks that do not fit into the main memory of a single computing machine. The algorithm partitions the network into non-overlapping subgraphs and reduces the memory requirement significantly leading to the ability to work with even larger networks. This algorithm is not only useful for networks with large degrees, it is equally useful for networks with small degrees when it comes to working with massive networks. We believe that for emerging massive networks, this algorithm will prove very useful. When the main memory of each computing machine is large enough to store the whole network, our parallel algorithm with dynamic load balancing can be used for faster analysis of the network.

References

[1] R. Milo, S. Shen-Orr *et al.*, "Network motifs: simple building blocks of complex networks." *Science*, vol. 298, no. 5594, pp. 824–827, October 2002.
[2] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *Proc. of KDD*, 2011.
[3] M. McPherson, L. Smith-Lovin, and J. Cook, "Birds of a feather: Homophily in social networks," *Annual Rev. of Soc.*, vol. 27, no. 1, pp. 415–444, 2001.
[4] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proc. of KDD*, 2008.
[5] 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.
[6] Z. Bar-Yosseff, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proc. of SODA*, 2002.
[7] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Computer Networks*, vol. 33, no. 16, pp. 309 – 320, 2000.
[8] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proc. of WWW*, 2010.
[9] M. Newman, "Coauthorship networks and patterns of scientific collaboration," *Proc. Natl. Acad. of Sci. USA*, vol. 101, no. 1, pp. 5200–5205, April 2004.
[10] 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.
[11] C. Apte, B. Liu, E. Pednault, and P. Smyth, "Business applications of data mining," *Commun. ACM*, vol. 45, no. 8, pp. 49–53, Aug. 2002.
[12] J. Chen and S. Lonardi, *Biological Data Mining*, 1st ed. Chapman & Hall/CRC, 2009.
[13] C. Tsourakakis, U. Kang, G. Miller, and C. Faloutsos, "Doulion: counting triangles in massive graphs with a coin," in *Proc. of KDD*, 2009.
[14] N. Alon, R. Yuster, and U. Zwick, "Finding and counting given length cycles," *Algorithmica*, vol. 17, pp. 209–223, 1997.
[15] T. Schank, "Algorithmic aspects of triangle-based network analysis," Ph.D. dissertation, University of Karlsruhe, 2007.
[16] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theor. Comput. Sci.*, vol. 407, pp. 458–473, 2008.
[17] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proc. of WWW*, 2011.
[18] C. Seshadhri, A. Pinar, and T. Kolda, "Triadic measures on graphs: the power of wedge sampling," in *Proc. of SDM*, 2013.
[19] M. Jha, C. Seshadhri, and A. Pinar, "A space efficient streaming algorithm for triangle counting using the birthday paradox," in *Proc. of KDD*, 2013.
[20] K. Tangwongsan, A. Pavan, and S. Tirthapura, "Parallel triangle counting in massive streaming graphs," in *Proc. of CIKM*, 2013.
[21] S. Arifuzzaman, M. Khan, and M. Marathe, "PATRIC: A parallel algorithm for counting triangles in massive networks," in *Proc. of CIKM*, 2013.
[22] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task, "Counting triangles in massive graphs with mapreduce," *CoRR*, vol. abs/1301.5887, 2013.
[23] H.-M. Park and C.-W. Chung, "An efficient mapreduce algorithm for counting triangles in a very large graph," in *Proc. of CIKM*, 2013.
[24] "Twitter Data," http://an.kaist.ac.kr/∼haewoon/release/twitter_social_graph, 2010, [Online].
[25] SNAP. (2012) Stanford network analysis project. http://snap.stanford.edu/. [Online]. Available: http://snap.stanford.edu/
[26] C. Barrett, R. Beckman *et al.*, "Generation and analysis of large synthetic social contact networks," in *WSC*, 2009.
[27] A. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, pp. 509–512, 1999.
[28] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The anatomy of the facebook social graph," *CoRR*, vol. abs/1111.4503, 2011.