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 \( O(\bar{d}) \)-factor 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 [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 the paper [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 still counts a particular type of triangles (triangles whose vertices lie in the same partition) \((P - 1)\) times. 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 \( \Omega (m^{3/2} \sqrt{\log m}) \) (Theorem 3 in [18]) which almost equals to the whole network. Another MapReduce based approximation algorithm is proposed in [16], which is based on parallelization of a wedge sampling technique given in [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 faster 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 nodes. 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, as results show, achieves a space efficiency of up to 25 times 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 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 \(O(d)\)-factor 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 experimental setup 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 of each other. The set of all neighbors of \(v\) in \(E\) is denoted by \(N_v\), i.e., \(N_v = \{u \in V | (u, v) \in E\}\). The degree of \(v\) is \(\deg_v = |N_v|\). 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.,

\[T_v = |\{(u, w) \in E | u, w \in N_v\}|.\]

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 and Machine Specifications. We develop parallel algorithms for MPI based distributed-memory parallel systems where each processor has its own local memory, and processors communicate via exchanging messages. 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.

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

<table>
<thead>
<tr>
<th>Network</th>
<th>Nodes</th>
<th>Edges</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>web-Google</td>
<td>0.85M</td>
<td>5.1M</td>
<td>SNAP [23]</td>
</tr>
<tr>
<td>web-BerkStan</td>
<td>0.69M</td>
<td>6.5M</td>
<td>SNAP [23]</td>
</tr>
<tr>
<td>Miami</td>
<td>2.1M</td>
<td>50M</td>
<td>[21]</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>4.8M</td>
<td>43M</td>
<td>SNAP [23]</td>
</tr>
<tr>
<td>Twitter</td>
<td>42M</td>
<td>2.4B</td>
<td>[24]</td>
</tr>
<tr>
<td>PA((n, d))</td>
<td>(n)</td>
<td>(\frac{n}{2})</td>
<td>Pref. Attachment</td>
</tr>
</tbody>
</table>
six permutations of \( u, v, \) and \( w. \) A total ordering \( v \) 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 broken by node IDs, as defined as follows:

\[
u \prec v \iff d_u < d_v \text{ or } (d_u = d_v \text{ and } u < v).
\]

These algorithms are further improved in a recent paper \([15]\) by a simple modification. The algorithm \([15]\) defines \( N_u \subseteq N_v \) as the set of neighbors of \( v \) having a higher order \( \prec \) than \( v, \)

\[
N_v = \{ u : (u, v) \in E, v \prec u \}.
\]

That is, for an edge \((u, v)\), the algorithm stores \( u \) in \( N_u \) if \( v \prec u \) and consequently \( u \in N_v \iff v \notin N_u. \) Then the triangles containing node \( v \) and any \( u \in N_v \) can be found by set intersection \( N_u \cap N_v. \) Now let, \( d_v = |N_v| \) be the effective degree of node \( v \). The cost for computing \( N_u \cap N_v \) requires \( O(d_v + d_u) \) time when \( N_u \) and \( N_u \) are sorted. 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 \( u, w \in N_v. \) 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 \( \bigcup_{0 \leq k < P} V_k^c = V. \) Then, a set \( V_i \) is constructed as \( V_i = V_i^c \cup \{ u \in V : u \notin V_i^c \}. \) 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 \( d \) of the network, the algorithm has a space requirement of \( \Omega(mP) \) for storing disjoint portion of the partition. Storing the whole partition requires \( \Omega(Pm^2) \) or \( \Omega(Pm) \) space, where \( 1 \leq x \leq d, \) which can be as large as the whole network \( O(m). \)

Our space-efficient parallel algorithm divides the input networks into non-overlapping partitions. Each partition is almost equal and has approximately \( \frac{m}{P} \) edges leading to a space complexity of \( O(Pm) \). The sizes of all partitions sum up to the size of the network. This partitioning offers as much as \( d \) times saving over the overlapping partitions and thus allows to work on larger networks. Table II shows the space requirement of our algorithm which is up to 25× smaller than that of \([15]\).

![Fig. 1. The state-of-the-art sequential algorithm for counting triangles.](image)

**TABLE II** MEMORY USAGE OF OUR ALGORITHM AND \([15]\) FOR STORING THE LARGEST PARTITION. NUMBER OF PARTITIONS USED IS 100.

<table>
<thead>
<tr>
<th>Networks</th>
<th>Memory (MB)</th>
<th>Ratio</th>
<th>( d )</th>
<th>( d_{max} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>web-Google</td>
<td>1.49</td>
<td>11.4</td>
<td>7.85</td>
<td>11.6</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>9.41</td>
<td>110.75</td>
<td>11.75</td>
<td>18</td>
</tr>
<tr>
<td>Miami</td>
<td>10.63</td>
<td>109.58</td>
<td>10.32</td>
<td>47.6</td>
</tr>
<tr>
<td>Twitter</td>
<td>265.82</td>
<td>4254.18</td>
<td>16.004</td>
<td>57.1</td>
</tr>
<tr>
<td>PA(10M, 100)</td>
<td>121.11</td>
<td>2126.94</td>
<td>17.5</td>
<td>100</td>
</tr>
<tr>
<td>PA(1M, 1000)</td>
<td>138.20</td>
<td>3427.36</td>
<td>24.8</td>
<td>1000</td>
</tr>
</tbody>
</table>

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 \( 0 \leq j, k \leq P - 1 \) and \( j \neq k, V_j \cap V_k = \emptyset \) 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 $0 \leq j, k \leq P - 1$ and $j \neq k$, $E_j \cap E_k = \emptyset$ 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 $u \in V_j$, information of both $N_u$ and $N_v$ 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 $u \in V_j$, $j \neq i$, $N_u$ 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 on 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 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 $V_i$,

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

(3)

Several estimations for $f(v)$ were proposed in [15] among which $f(v) = \sum_{u \in N_v} (d_u + d_v)$ 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).

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 $\frac{in_i}{P}$ to $\frac{(i+1)n_i}{P} - 1$. This computation takes $O(\frac{n_i}{P} + P)$ 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 $V_i = \{n_i, n_{(i+1)} - 1\}$ using the algorithms [15].

Once all $P$ partitions are computed, each processor is assigned one such partition.

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 $p_j$ counts triangles incident on the edge $(v,u)$ by computing $N_u \cap N_v$. 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 $v_1, v_2, \ldots, v_k$. In response to those requests, $p_j$ sends same $N_u$ 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 $N_u$; finds all $k$ nodes $v \in V_i$ such that $u \in N_v$. It then performs all $k$ computations $N_u \cap N_v$ involving $N_u$ and discards $N_u$. Now, since $u \in N_{v_1} \cap N_{v_2} \cap \cdots \cap N_{v_k}$, for $v_1, v_2, \ldots, v_k \in V_i$. Then $p_i$ 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 a different approach which exploits an inherent property of our data structure $N_u$ 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 $p_i$ instead of computing $N_u \cap N_v$ by itself delegating it to $p_j$? In particular, we consider an alternate approach to address
the above issue: \( p_i \) sends \( N_u \) to \( p_j \) instead of fetching \( N_u \). Processor \( p_j \) counts triangles incident on edge \((u, v)\) by performing the operation \( N_u \cap N_v \).

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 \( N_u \cap N_v \). 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 \( O(\sum_{v \in V_j} d_v) \) runtime of direct approach for the similar purpose). In fact, this extraction can be done while computing triangles \( N_u \cap N_v \) 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 \( N_u \) 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 \( O(|V_i| \cdot P) \). Now, note the following simple yet useful property of \( N_v \): Since \( V_j \) is a set of consecutive nodes, and all neighbor lists \( N_u \) 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, \( \text{LastProc} \)) it has sent \( N_v \) to. When \( p_i \) encounters \( u \in N_v \) such that \( u \in V_j \), it checks \( \text{LastProc} \). If \( \text{LastProc} \neq p_j \), then \( p_i \) sends \( N_v \) to \( p_j \) and set \( \text{LastProc} = p_j \). Otherwise, the node \( u \) is ignored, meaning it would be redundant to send \( N_v \). Resetting a single variable \( \text{LastProc} \) 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 \( t \in \{ \text{data, notifier} \} \) is the type and \( X \) is the actual data associated with the message. For a data message \( \langle t = \text{data} \rangle \), \( X \) refers to a neighbor list \( N_x \) whereas for a completion notifier \( \langle t = \text{notifier} \rangle \), \( X = 0 \). The pseudocode for counting triangles for an incoming data message \( \langle \text{data}, X \rangle \) is given in Fig. 2.

Once a processor \( p_i \) completes the computation on all \( v \in V_t \), it broadcasts a completion notifier \( \langle \text{notifier}, X \rangle \). However, it cannot terminate execution until it receives \( \langle \text{notifier}, X \rangle \) 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.

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 \( x_1 < x_2 < x_3 \). 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 $X = N_{x_2}$, and node $x_3$ appears in $S = N_{x_2} \cap N_{x_3}$ (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 \( O \left( \sum_{u \in N_{v} - N_{u}} (d_u + d_u) \right) \).

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

\[
u \in N_v - N_u \Leftrightarrow v \in N_u.
\]

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 $u \in N_v$. There might be two cases:

1. **Case $u \in N_v - N_u$:** 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 $v$.

Thus for both sub-cases $p_i$ computes triangles incident on $(v, u)$, all such nodes $u$ impose a computation cost of \( O \left( \sum_{u \in N_{v} - N_{u}} (d_u + d_u) \right) \) on $p_i$ for node $v$.

2. **Case $u \in N_u$:** This case implies $v \in N_u - N_v$ (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 $u \in N_v$, is \( O \left( \sum_{u \in N_{v} - N_{u}} (d_u + d_u) \right) \).

Theorem 2 gives us the intended function $f(v) = \sum_{u \in N_{v} - N_{u}} (d_u + d_u)$ 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 $u \in V_j, j \neq i$, i.e., $(v, u)$ is a cut edge. Next let $X_i = \bigcup_{u \in V_i} C_v$ 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 $W_{\text{dir}}$ of exchanging messages by processor $p_i$ is given by.

\[
O \left( \sum_{v \in V_i} \sum_{0 \leq j \leq P - 1} \left[ \sum_{u \in N_v} (d_u + O(1)) + \sum_{u \in N_u} (d_u + O(1)) \right] \right).
\]

**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 $|N_u| = d_u$ of receiving $N_u$ (for $v < u$) or the cost $|N_v| = d_v$ of sending $N_v$ (for $u < v$) and the second term $O(1)$ for the cost of a request message.

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

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

\[
O \left( \sum_{v \in V_i} \sum_{0 \leq j \leq P - 1} \left[ \sum_{u \in N_v} \frac{d_u}{l_{ij}} + \sum_{u \in N_u} \frac{d_u}{l_{ij}} \right] \right).
\]

**Proof:** For node $v \in V_i$, all cut edges $(v, u)$ with $u \in V_j$ and $v < u$, $N_u$ is sent to $p_j$ at most once instead of $l_{ij}$ times. Thus each such $(v, u)$ is attributed to $\frac{1}{l_{ij}}$-fraction of the sending cost $O(d_u)$, which is accounted by the first innermost summation term. For all edges $(v, u)$ with $u < v$, $p_i$ receives $N_u$ from $p_j$ at most once instead of $l_{ij}$ 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 $d_u$, for all $v \in V_i$, by the average degree $\bar{d}$ and number of cut edges from $v$ to partition $j$ with $v < u$, $l_{ij}$, by $l$ (an average over all $l_{ij}$). Then we get, the communication cost for Surrogate approach, $W_{\text{sur}}$: 

\[
O \left( \sum_{v \in V_i} \sum_{0 \leq j \leq P - 1} \left[ \sum_{u \in N_v} \frac{d_u}{l} + \sum_{u \in N_u} \frac{d_u}{l} \right] \right) = O \left( \frac{\sum_{v \in V_i} (|X_i|d_v)}{l} \right) = O \left( \frac{x_i d}{l} \right).
\]
The second last step follows from $X_i = \bigcup_{v \in V_i} C_v$. Similarly, we get, communication cost for Direct approach, $W_{\text{dir}}$:

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

Since $\frac{x_i}{x} > l$, 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>
<thead>
<tr>
<th>Networks</th>
<th># of Messages</th>
<th>Ratio</th>
<th>$t$ (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miami</td>
<td>16,321,478</td>
<td>3,987,871</td>
<td>4.09</td>
</tr>
<tr>
<td>Web-Google</td>
<td>492,488</td>
<td>99,221</td>
<td>4.97</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>23,138,824</td>
<td>4,002,575</td>
<td>5.78</td>
</tr>
<tr>
<td>Twitter</td>
<td>247,821,246</td>
<td>25,341,984</td>
<td>9.78</td>
</tr>
<tr>
<td>PA(10M, 100)</td>
<td>99,436,823</td>
<td>8,092,340</td>
<td>12.29</td>
</tr>
</tbody>
</table>

D. Complexity of the Algorithm.

Runtime Complexity. Computing balanced partition takes $O\left( \frac{n}{p} + P \right)$ time using an adaptation of [15]. The worst case cost for counting triangles is $O\left( \sum_{v \in V} \sum_{u \in N_v} \left( d_u + \hat{d}_u \right) \right)$ (Fig. 3). Further, the communication cost incurred on a processor is $O\left( x_i \bar{d} / l \right)$ (Sec. V-C). The summing up of counts require $O\left( \log P \right)$ time using MPI aggregation function. Thus, the time complexity of our parallel algorithm is, $O\left( \frac{m}{p} + P + \max_i x_i \bar{d} / l + \max_i \sum_{v \in V} \sum_{u \in N_v} \left( d_u + \hat{d}_u \right) \right)$.

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 $N_i$ requires a space of $O(\max_{v \in V} \{|V_i| + |E_i|\}) = O(\bar{d})$. Thus, the total space complexity of our algorithm is $O(\max_i \{ |V_i| + |E_i| \})$ which approximately equals to $O\left( \frac{m}{p} \right)$ with the non-overlapping balanced partitions. A comparison of space complexity of other related algorithms is provided in Table IV. Our algorithm achieves up to $O(\bar{d})$-factor space efficiency over MapReduce based algorithms [12], [17], [18] and up to $O(\bar{d})$-factor over the algorithm with overlapping partitioning [15].

VI. EXPERIMENTAL EVALUATION

The experimental evaluation of the performance our space-efficient parallel algorithm is presented below.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Space complexity</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suri et al. [12]</td>
<td>$O(mP)$</td>
<td>size of Map output</td>
</tr>
<tr>
<td>Park et al. [17]</td>
<td>$O(mP)$</td>
<td>size of Map output</td>
</tr>
<tr>
<td>Park et al. [18]</td>
<td>$\Omega\left( \frac{m^2}{P^2} \right)$, 1 $\leq x \leq d$</td>
<td>for each processor</td>
</tr>
<tr>
<td>PATRIC [15]</td>
<td>$\Omega\left( \frac{m^2}{P^2} \right)$</td>
<td>for each processor</td>
</tr>
<tr>
<td>Our algo.</td>
<td>$O\left( \frac{m^2}{P^2} \right)$</td>
<td>for each processor</td>
</tr>
</tbody>
</table>

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 $\sim$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 $35 \times$ faster than [12], $17 \times$ than [17], $7 \times$ than [18], and almost as fast as [15].

<table>
<thead>
<tr>
<th>Networks</th>
<th>Runtime</th>
<th>Triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>web-BerkStan</td>
<td>0.10s</td>
<td>65M</td>
</tr>
<tr>
<td>Miami</td>
<td>0.6s</td>
<td>332M</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>0.8s</td>
<td>290M</td>
</tr>
<tr>
<td>Twitter</td>
<td>9.4m</td>
<td>34.8B</td>
</tr>
<tr>
<td>PA(1B, 20)</td>
<td>15.5m</td>
<td>0.43M</td>
</tr>
</tbody>
</table>

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.

Effect of Estimation for $f(v)$. We show the performance of our algorithm with new weight function (computed in Section V-B), $f(v) = \sum_{u \in N_v} (d_v + \hat{d}_u)$, 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. 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].

![Graph](image_url)

Fig. 8. Weak scaling of our algorithm, experiment performed on PA\((t\times10^9, 50)\) 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.

**REFERENCES**


