PATRIC: A Parallel Algorithm for Counting Triangles in Massive Networks

Shaikh Arifuzzaman[*][†], Maleq Khan[†], and Madhav V. Marathe[*][†]
[*]Department of Computer Science
[†]Network Dynamics and Simulation Science Laboratory
Virginia Tech, Blacksburg, VA 24061, USA
{sm10, maleq, mmarathe}@vbi.vt.edu

ABSTRACT

The recent emergence of massive networks from numerous areas poses significant challenges for network analysts. These networks consists of millions, or even billions, of nodes and edges. Conventional algorithms for measuring various network metrics fail drastically when the size of the network grows to billions of nodes and the network is prohibitively large to fit in the main memory. Finding the number of triangles in a network is an important problem in the analysis of complex networks. Several interesting graph mining applications depend on the number of triangles in the graph of interest. In this paper, we present an efficient MPI-based distributed memory parallel algorithm, called PATRIC, for counting triangles in massive networks. PATRIC scales well to networks with billions of nodes and can compute the exact number of triangles in a network with one billion nodes and 10 billion edges in 16 minutes. Balancing computational loads among processors for a graph problem like counting triangles is a challenging issue. We present and analyze several schemes for balancing load among processors for the triangle counting problem. These schemes achieve a very good load balancing. We also show how our parallel algorithm can adopt an existing edge sparsification technique to approximate the number of triangles with very high accuracy. The adoption of this sparsification technique allows us to work with even larger networks.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph Algorithms*; D.1 [**Programming Techniques**]: Parallel Programming; H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithm, Experimentation, Performance

Keywords

triangle-counting, clustering-coefficient, massive networks, parallel algorithms, social networks, graph mining

1. INTRODUCTION

We study the problem of counting triangles in massive networks that do not fit in the main memory of a single computing node. We present an MPI-based distributed memory parallel algorithm for this problem, which scales well to networks with billions of nodes and edges. Counting triangles in a network is an important and fundamental 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 [11, 12, 23]. Clustering coefficient, transitivity ratio, etc. quantify the triangle density in a network. Existence of triangles and the resulting high clustering coefficient in a social network reflect the common theory of social science where people who have common friends tend to be friends themselves [22]. Also, triangle counting has important applications in graph mining. Recently, it has been used to detect spamming activity and assess content quality in social networks [8], to uncover thematic structure of the web [15], and query planning optimization in databases [5].

Today, data from diverge fields are modeled as graphs because of their convenience in representing underlying relations and structures [12]. Some significant examples are the Web, various social networks, e.g., Facebook, Twitter [20], collaboration and co-authorship networks [24], infrastructure networks (e.g. transportation networks, telephone networks, power networks) and many forms of biological networks [17]. Due to the advancement in technology, we are deluged with data from a wide range of areas such as business and finance [4], computational biology [10] and social science. As Google reported in 2008, the Web graph has over 1 trillion webpages. Most of the social networks, such as Twitter, Facebook, MSN, have millions to billions of users [11, 31]. To analyze these huge data represented by massive networks, faster and parallel algorithms are necessary. Furthermore, such massive networks pose another challenge of dealing with a large memory requirement. These graphs do not fit in the main memory of a single processing unit, and the algorithms must be able to work with a small part of the graph at a time.

The problem of computing the clustering coefficients of the nodes in a graph, and almost equivalently counting triangles, has a rich history [3, 18, 21, 27]. Both exact and approximate algorithms for this problem can be found in the literature. Much of the earlier work focused on improving running time rather than paying attention to memory issues, and these algorithms are mainly based on matrix multiplication and adjacency matrix representation of the network. Among these algorithms, the state of the art is an algorithm due to Alon et al. [3]. These matrix based algorithms [3, 13, 18] are not useful in the analysis of social networks as adjacency matrix representation of network requires $O(n^2)$ memory space, where n is the number of nodes in the network – even for medium sized networks, say a graph with a few hundred thousand nodes, such amount of space can be prohibitively large. As a result, in the last decade the focus has been shifted to algorithms that use adjacency list representation [21, 27], which takes $O(m)$ memory. In a real-world network, m can be much smaller than n^2 as the average degree of a node can be significantly smaller than n, although few of the nodes can have very large degrees. In the last decade, many exact [21, 26, 27] and approximation [16, 25, 28, 31] algorithms have been developed using adjacency list representations.

Although a fairly large volume of work has been done on this problem, much less attention was given, until recently, to the problems associated with massive networks that do not fit in the main memory. Several techniques can be employed to deal with such massive graphs: streaming algorithms, sparsification based algorithms, external-memory algorithms, and distributed memory parallel algorithms. The streaming and sparsification based techniques provide approximation algorithms whereas external-memory and parallel algorithms can be used to find exact solutions.

Streaming algorithms perform a constant number of passes, in some cases $O(\log n)$ passes, over the data streams and make an estimation of the number of triangles. Some streaming-based algorithms can be found in [5,8,19]. Very recently two algorithms based on sparsification are given in [31] and [25]. While loading the graphs to the main memory, these algorithms store only a randomly chosen subset of the edges in the memory. Then the number of triangles in the original network is estimated based on the number of triangles in this sparsed graph. This sparsification technique can alleviate both time and space issues; however, they do so by sacrificing the quality of the solution. Although the estimation for the total triangle counts can be reasonably good for some applications, the accuracy for the local triangle counts and clustering coefficients of the individual nodes can be very poor.

To the best of our knowledge, very few papers have addressed the problems associated with massive networks that do not fit in the main memory and provide an exact solution. A recent paper [11] presents an external-memory algorithm to find the exact number of triangles in a network. Although this algorithm provides an impressive solution to working with massive networks, external-memory algorithms can be very I/O intensive leading to a large running time. Only a parallel algorithm can solve such a problem of a large running time by distributing computing tasks to multiple processors.

Another very recent paper [30] presents a parallel algorithm for exact triangle counts using MapReduce framework [14]. Our parallel algorithm improves the performance, both in time and space, over [30] significantly. A detailed comparison with this algorithm is given in Section 4. Next we summarize our contributions below:

- We present a parallel algorithm called PATRIC, for counting and listing triangles in massive networks. PATRIC scales almost linearly with the number of processors, and is able to process a network with 1 billion nodes and 10 billion edges in 16 minutes using 40 processors. We show the performance of PATRIC by using both artificial and real-world networks.
- We devise and analyze several load balancing schemes for the parallel algorithm. With these schemes, we achieve very good load balancing, even for networks with skewed degree distributions.
- We show how the sparsification technique presented in [31] can be adopted in our parallel algorithm to have a parallel approximation algorithm. Use of such a sparsification technique allows our parallel algorithm to work with even larger networks. Moreover, we show that our parallel sparsification improves the accuracy of the approximation over the sequential sparsificaiton of [31].
- We show, both theoretically and experimentally, a simple modification of a current state of the art sequential algorithm for counting triangles improves its performance and use this modified algorithm in the development of PATRIC.

The rest of the paper is organized as follows. The preliminary concepts, notations and datasets are briefly described in Section 2. In Section 3, we discuss sequential algorithms for counting triangles. We present our parallel algorithm PATRIC and the load balancing schemes in Section 4. The parallelization of the sparsification technique is given in Section 5, and we conclude in Section 6.

2. PRELIMINARIES

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

Notations. The given network is denoted by $G(V, E)$, where V and E are the sets of vertices (nodes) 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 N_v, i.e., $N_v = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |N_v|$.

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 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 as same as the number of edges among the neighbors of v, i.e., $T_v = |\{(u, w) \in E \mid u, w \in N_v\}|$.

The clustering coefficient (CC) of a node $v \in V$, denoted by C_v is the ratio of the number of edges between neighbors of v to the number of all possible edges between neighbors of v. Then, we have, $C_v = \frac{T_v}{\binom{d_v}{2}} = \frac{2T_v}{d_v(d_v-1)}$.

Thus, CC of a node can be computed by simply counting the number of triangles incident on the node, and computing CC is an almost equivalent problem to counting triangles.

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

Datasets. We used both real world networks and artificially generated networks. A summary of all the networks

Table 1: Dataset used in our experiments

Network	**Nodes**	**Edges**	**Source**
Email-Enron	37K	0.36M	SNAP [29]
web-BerkStan	0.69M	13M	SNAP [29]
Miami	2.1M	100M	[7]
LiveJournal	4.8M	86M	SNAP [29]
Twitter	42M	2.4B	[1]
Gnp(n, d)	n	$\frac{1}{2}nd$	Erdős-Réyni
PA(n, d)	n	$\frac{1}{2}nd$	Pref. Attachment

is provided in Table 1. Twitter data set is available at [1], and web-BerkStan, LiveJournal and Email-Enron networks are at SNAP library [29]. Miami is a synthetic, but realistic, social contact network [7] for Miami city: each node is a person from Miami city, and there is an edge between two persons if they "physically" interact with each other within a 24 hour period. Network Gnp(n, d) is generated using the Erdős-Réyni random graph model [9], also known as $G(n, p)$ model, with n nodes and edge probability $p = \frac{d}{n-1}$ so that the expected degree of each node is d. Network PA(n, d) is generated using preferential attachment (PA) model [6] with n nodes and average degree d. PA(n, d) has power-law degree distribution, which is a very skewed distribution.

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.

Experimental Setup. 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. We evaluate the performance of our algorithms using both real-world networks and artificial networks as shown in Table 1. The number of nodes in the real-world networks ranges from 37K to 42M and the number of edges from 0.36M to 2.4B. Note that the web-BerkStan, LiveJournal and Twitter networks have a very skewed degree distribution. For experimenting with even larger networks, we rely on random PA(n, d) networks, with varying n and d, which allow us to experiment with varying network sizes. Since PA(n, d) networks have extremely skewed degree distribution, they make load balancing a challenging task for most parallel graph algorithms and give us a chance to measure the performance of our algorithms in some of the worst case scenarios.

3. SEQUENTIAL ALGORITHMS

In this section, we discuss sequential algorithms for counting triangles using adjacency list representation and show that a simple modification to a state-of-the-art algorithm improves both time and space complexity. Although the modification seems quite simple, and others might have used it previously, our theoretical and experimental analysis of this modification are new. To the best of our knowledge, our analysis is the first to show that such simple modification improves the performance significantly. This modification is also used in our parallel algorithm PATRIC.

```
1: $T \leftarrow 0$        {$T$ stores the count of triangles}
2: for $v \in V$ do
3:     for $u \in N_v$ and $v \prec u$ do
4:         for $w \in N_v$ and $u \prec w$ do
5:             if $(u, w) \in E$ then
6:                 $T \leftarrow T + 1$
```

Figure 1: Algorithm NodeIterator++, where $\prec$ is the degree based ordering of the nodes defined in Equation 1.

A simple but efficient algorithm [26, 30] 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 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. Many algorithms exist [11,21,26,27,30], which provide significant improvement over the above method. A very comprehensive survey of the sequential algorithms can be found in [21, 26]. One of the state of the art algorithms, known as NodeIterator++, as identified in two very recent papers [11, 30], is shown in Figure 1. Both [11] and [30] use this algorithm as a basis of their external-memory algorithm and parallel algorithm, respectively.

This algorithm 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, the algorithm NodeIterator++ incorporates an interesting node ordering based on the degrees of the nodes, with ties broken by node IDs, as defined below:

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

This degree based ordering can improve the running time. Let $\hat{d}_v$ be the number of neighbors u of v such that $v \prec u$. We call $\hat{d}_v$ the*effective degree* of v. Assuming N_vs, for all v, are sorted and a binary search is used to check $(u, w) \in E$, the running time can be shown to be $O(\sum_v \hat{d}_v d_v + \hat{d}_v^2 \log d_{\max})$, where $d_{\max} = \max_v d_v$. This running time is minimized when $\hat{d}_v$ values of the nodes are as close to each other as possible, although, for any ordering of the nodes, $\sum_v \hat{d}_v = m$ is invariant. Notice that in the degree-based ordering, diversity of the $\hat{d}_v$ values are reduced significantly.

We also observe that for the same reason, degree-based ordering of the nodes helps keep the loads among the processors balanced, to some extent, in a parallel algorithm. We use this degree-based ordering in our parallel algorithm PATRIC and discuss this issue in detail in Section 4.

A simple modification of NodeIterator++ is as follow: perform comparison $u \prec v$ for each edge $(u, v) \in E$ in a preprocessing step rather than doing it while counting the triangles. This preprocessing step reduces the total number of $\prec$ comparisons to $O(m)$ and allows us to use an efficient set intersection operation. For each node v, set of neighbors N_v is maintained in the memory; however, for each edge (v, u), u is stored in N_v if and only if $v \prec u$. The modified algorithm NodeIteratorN is presented in Figure 2. All triangles containing node v and any $u \in N_v$ can be found by set intersection $N_u \cap N_v$ (Line 10 in Figure 2). The correctness of NodeIteratorN is proven in Theorem 1.

THEOREM 1. *Algorithm NodeIteratorN counts each triangle in G once and only once.*

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

Figure 2: Algorithm NodeIteratorN, a modification of NodeIterator++.

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 in the preprocessing step, we have $x_2, x_3 \in N_{x_1}$ and $x_3 \in N_{x_2}$. When the loops in Line 8-9 begin with $v = x_1$ and $u = x_2$, node x_3 appears in S (Line 10-11), 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 (in Line 8-9) because $x_1 \notin N_{x_2}$ and $x_1, x_2 \notin N_{x_3}$. $\square$

In NodeIteratorN, $|N_v| = \hat{d}_v$, the effective degree of v. When N_v and N_u are sorted, $N_u \cap N_v$ can be computed in $O(\hat{d}_u + \hat{d}_v)$ time. Then we have $O\left(\sum_{v \in V} d_v \hat{d}_v\right)$ time complexity for NodeIteratorN as shown in Theorem 2, in contrast to $O(\sum_v \hat{d}_v d_v + \hat{d}_v^2 \log d_{\max})$ for NodeIterator++.

THEOREM 2. *The time complexity of algorithm NodeIteratorN is* $O\left(\sum_{v \in V} d_v \hat{d}_v\right)$.

PROOF. Time for the construction of N_v for all v is $O(\sum_v d_v) = O(m)$, and sorting these N_v requires $O(\sum_v \hat{d}_v \log \hat{d}_v)$ time. Now, computing intersection $N_v \cap N_u$ takes $O(\hat{d}_u + \hat{d}_v)$ time. Thus, the time complexity of NodeIteratorN is

$$\begin{aligned}
& O(m) + O(\sum_{v \in V} \hat{d}_v \log \hat{d}_v) + O(\sum_{v \in V} \sum_{u \in N_v} (\hat{d}_u + \hat{d}_v)) \\
= \; & O(\sum_{v \in V} \hat{d}_v \log \hat{d}_v) + O(\sum_{(v,u) \in E} (\hat{d}_u + \hat{d}_v)) \\
= \; & O(\sum_{v \in V} \hat{d}_v \log \hat{d}_v) + O(\sum_{v \in V} d_v \hat{d}_v) = O(\sum_{v \in V} d_v \hat{d}_v).
\end{aligned}$$

The second last step follows from the fact that for each $v \in V$, term $\hat{d}_v$ appears d_v times in this expression. $\square$

Notice that set intersection operation can also be used with NodeIterator++ by replacing Line 4-6 of NodeIterator++ in Figure 1 with the following three lines as shown in [11] (Page 674):

```
1: $S \leftarrow N_v \cap N_u$
2: for $w \in S$ and $u \prec w$ do
3:     $T \leftarrow T + 1$
```

However, with this set intersection operation, the runtime of NodeIterator++ is $O\left(\sum_v d_v^2\right)$ since $|N_v| = d_v$ in NodeIterator++, and computing $N_v \cap N_u$ takes $O(d_u + d_v)$ time. Further, the memory requirement for NodeIteratorN is half of that for NodeIterator++. NodeIteratorN stores $\sum_v \hat{d}_v = m$ elements in all N_v and NodeIterator++ stores $\sum_v d_v = 2m$ elements. Here we would like to note that two algorithms presented in [21, 27] take the same asymptotic time complexity as NodeIteratorN. However, the algorithm in [27] requires three times more memory than NodeIteratorN. The algorithm in [21] requires more than twice the memory as NodeIteratorN, maintains a list of indices for all nodes, and the hidden constant in the runtime can be much larger.

Table 2: Running time for sequential algorithms

Networks	**Runtime (sec.)**		**Triangles**
	NodeItr.++	NodeItr.N	
Email-Enron	0.14	0.07	0.7M
web-BerkStan	3.5	1.4	64.7M
LiveJournal	106	42	285.7M
Miami	46.35	32.3	332M
PA(25M, 50)	690	360	1.3M
Gnp(500K, 20)	1.81	0.6	1308

We also experimentally compare the performance of NodeIteratorN and NodeIterator++ using both real-world and artificial networks. NodeIteratorN is significantly faster than NodeIterator++ for these networks as shown in Table 2.

4. THE PARALLEL ALGORITHM

In this section, we present our parallel algorithm PATRIC for counting triangles and computing clustering coefficients.

4.1 Overview of the Algorithm

We assume that the network is massive and does not fit in the local memory of a single computing node. Locally each processor stores only a part of the network in its memory. Let P be the number of processors used in the computation. The network is partitioned into P partitions, and each processor is assigned one such partition $G_i(V_i, E_i)$ (formally defined below). Processor i performs computation on its partition G_i. The network data is given as input in a single disk file. Each processor, in parallel, reads its own part of the network (the necessary data to construct its own partition G_i) in its local memory. The main steps of PATRIC are given in Figure 3. In the following subsections, we describe the details of these steps and several load balancing schemes.

```
1: for each processor $i$, in parallel, do
2:     $G_i(V_i, E_i) \leftarrow$ READGRAPH$(G, i)$
3:     $T_i \leftarrow$ COUNTTRIANGLES$(G_i, i)$
4: BARRIER
5: Find Sum $T = \sum_i T_i$ using MPIREDUCE
6: return $T$
```

Figure 3: The main steps of PATRIC.

4.2 Partitioning the Network

The memory restriction poses a difficulty where the graph must be partitioned in such a way that the memory required to store a partition is minimized and at the same time the partition contains sufficient information to minimize communications among processors. For the input graph $G(V, E)$, processor i works on $G_i(V_i, E_i)$, which is a subgraph of G

induced by V_i. Each processor i is responsible for counting triangles incident on the nodes in V_i^c, where $V_i^c \subset V_i \subset V$, such that for any i and j, $V_i^c \cap V_j^c = \emptyset$ and $\bigcup_i V_i^c = V$. We call any node $v \in V_i^c$ a *core* node of processor i. Each $v \in V$ is a core node in exactly one partition. How the nodes in V is distributed among the core sets V_i^c for all processors i crucially and significantly affect the load balancing and performance of the algorithm. Later in Section 4.4, we present several load balancing schemes and the details of how sets V_i^c are constructed. Set V_i contains all nodes in V_i^c and $\bigcup_{v \in V_i^c} N_v$. Edge set $E_i \subset E$ is the set of edges (u, v) such that $u, v \in V_i$ and $(u, v) \in E$.

Processor i stores neighbor set N_v of all $v \in V_i$. Notice that for a node $w \in (V_i - V_i^c)$, neighbor set N_w may contain some nodes that are not in V_i. Such neighbors of w, which are not in V_i, can be safely removed from N_w and the number of triangles incident on all $v \in V_i^c$ can be computed correctly. But, the presence of these nodes in N_w does not affect the correctness of the algorithm either. However, we do not store such nodes in N_w to optimize memory usage. Figure 4 shows the differences in memory usage with and without this optimization for two networks: Miami and LiveJournal. As the experimental results show, this optimization saves about 50% of memory space. Figure 4 also demonstrates the memory-scalability of PATRIC: as the more processors are used, each processor consumes less memory space.

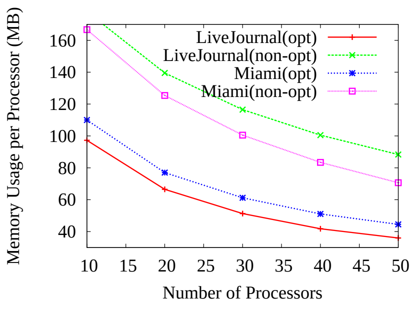


Figure 4: Memory usage with optimized and non-optimized data storing for Miami and LiveJournal networks.

4.3 Counting Triangles

Once each processor i has its partition $G_i(V_i, E_i)$, it uses the modified sequential algorithm *NodeIteratorN* presented in Section 3 to count the triangles in G_i for each *core* node $v \in V_i^c$. Neighbor sets N_w for the nodes $w \in V_i - V_i^c$ only help in finding the edges among the neighbors of the core nodes. To be able to use an efficient intersection operation, N_v for all $v \in V_i$ are sorted. The code executed by processor i is given in Figure 5.

```
1: for $v \in V_i$ do
2:     sort $N_v$ in ascending order
3: $T \leftarrow 0$
4: for $v \in V_i^c$ do
5:     for $u \in N_v$ do
6:         $S \leftarrow N_v \cap N_u$
7:         $T \leftarrow T + |S|$
8: return $T$
```

Figure 5: Algorithm executed by processor i to count triangles in $G_i(V_i, E_i)$.

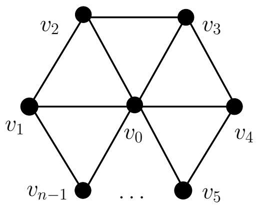


Figure 6: A network with a skewed degree distribution: the degree of v_0 is $n - 1$ while that of the other nodes are 3.

Once all processors complete their counting steps, the counts from all processors are aggregated into a single count by an MPI reduce function, which takes $O(\log P)$ time. Ordering of the nodes, construction of N_v, and disjoint node partitions V_i^c make sure that each triangle in the network appears exactly in one partition G_i. Thus, the correctness of the sequential algorithm *NodeIteratorN* shown in Section 3 ensures that each triangle is counted exactly once.

4.4 Load Balancing in PATRIC

A parallel algorithm is completed when all of the processors complete their tasks. Thus, to reduce the running time of a parallel algorithm, it is desirable that no processor remains idle and all processors complete their executions almost at the same time. Furthermore, to deal with a massive network, it is also desirable that all partitions $G_i(V_i, E_i)$ require almost the same amount of memory space.

In Section 3, we discussed how degree based ordering of the nodes can reduce the running time of the sequential algorithm, and hence it reduces the running time of the local computation in each processor i. We observe that, interestingly, this degree-based ordering also provides load balancing to some extent, both in terms of running time and space, at no additional cost. Consider the example network shown in Figure 6. With an arbitrary ordering of the nodes, $|N_{v_0}|$ can be as much as $n - 1$, and a single processor which contains v_0 as a core node is responsible for counting all triangles incident on v_0. Then the running time of the parallel algorithm can essentially be as same as that of a sequential algorithm. With the degree-based ordering, we have $|N_{v_0}| = 0$ and $|N_{v_i}| \leq 3$ for all $i \neq 0$. Now if the core nodes are equally distributed among the processors, both space and computation time are almost balanced.

Although degree-based ordering helps mitigate the effect of skewness in degree distribution and balance load to some extent, working with more complex networks and highly skewed degree distribution reveals that distributing core nodes equally among processors does not make the load well-balanced in many cases. Figure 7 shows speedup of the parallel algorithm with equal number of core nodes assigned to each processor. The speedup factor due to a parallelization is defined as t_s/t_p, where t_s and t_p are computation time required by a sequential and the parallel algorithm, respectively. As shown in Figure 7, LiveJournal networks show poor speedup, whereas the Miami network shows a relatively better speedup. This poor speedup for LiveJournal network is a consequence of highly imbalanced computation load across the processors as shown in Figure 8. Although most of the processors complete their tasks in less than a second, very few of them take an unusually longer time leading to poor speedup. Unlike Miami network, LiveJournal network has a very skewed degree distribution. In the next section, we present several load balancing schemes that im-

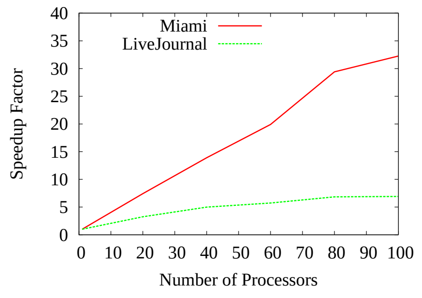


Figure 7: Speedup with equal number of core nodes in all processors.

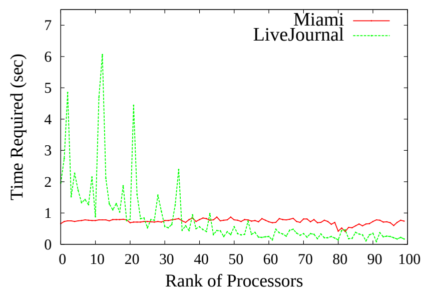


Figure 8: computing load of individual processors with equal number of core nodes.

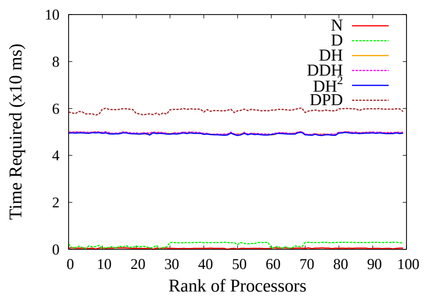


Figure 9: Load balancing cost for LiveJournal network with different schemes.

prove the performance of our algorihtm significantly.

Proposed Load Balancing Schemes

The proposed load balancing schemes require some pre-computation before executing the main steps for counting the triangles. Thus, our parallel algorithm PATRIC works in two phases, as shown below.

1. *Computing balanced load:* This phase computes partitions V_i^c so that the computational loads are well-balanced.
2. *Counting triangle:* This phase counts the triangles following the algorithms in Figure 3 and 5.

Computational cost for phase 1 is referred to as *load-balancing cost*, for phase 2 as *counting cost*, and the total cost for these two phases as *total computational cost*. In order to be able to distribute load evenly among the processors, we need an estimation of computation load for computing triangles. For these purpose, we define a *cost function* $f : V \to \mathbb{R}$, such that $f(v)$ is the computational cost for counting triangle incident on node v (Lines 4-7 in Figure 5). Then, the total cost incurred to processor i is given by $\sum_{v \in V_i^c} f(v)$. To achieve a good load balancing, $\sum_{v \in V_i^c} f(v)$ should be almost equal for all i. Thus, the computation of balanced load consists of the following:

1. **Computing f:** Compute $f(v)$ for each $v \in V$
2. **Computing partitions:** Determine P disjoint partitions V_i^c such that

$$\sum_{v \in V_i^c} f(v) \approx \frac{1}{P} \sum_{v \in V} f(v) \qquad (2)$$

The above computation must also be done in parallel. Otherwise, this computation takes at least $\Omega(n)$ time, which can wipe out the benefit gained from balancing load completely or even have negative effect on the performance. Parallelizing the above computation, especially computing partitions step, is a non-trivial problem. Next, we describe parallel algorithm to perform the above computation.

Computing f:
It might not be possible to exactly compute the value of $f(v)$ before the actual execution of triangle counting takes place. Fortunately, Theorem 2 provides a mathematical formulation of triangle counting cost in terms of the number of vertices, edges, original degree d and effective degree $\hat{d}$. Guided by Theorem 2, we have come up with several approximate cost function $f(v)$ which are listed in Table 3. Each function corresponds to one load balancing scheme. The rightmost column of the table contains short notations used to identify the individual schemes.

Table 3: Cost functions $f(.)$ for load balancing schemes

Node Function	Identifying Notation
$f(v) = 1$	$\mathbb{N}$
$f(v) = d_v$	$\mathbb{D}$
$f(v) = \hat{d}_v$	$\mathbb{DH}$
$f(v) = d_v \hat{d}_v$	$\mathbb{DDH}$
$f(v) = \hat{d}_v^{\,2}$	$\mathbb{DH}^2$
$f(v) = \sum_{u \in N_v} (\hat{d}_v + \hat{d}_u)$	$\mathbb{DPD}$

The input network is given in a file in adjacency list format: adjacency list of the first node followed by that of the second node, and so on. The input file is considered divided by size (number of bytes) into P chunks. Initially, processor i reads in ith chunk from the file in parallel. Thus processor i contains the adjacency lists N_v for the nodes v that are in the ith chunk. If a chunk boundary falls in the middle of an adjacency list, the boundary is shifted so that the entire adjacency list is in only one chunk. (Note that the nodes in ith chunk do not necessarily constitute the core nodes V_i^c for processor i. These chunks are used only for the purpose of computing balanced load and finding the actual partitions V_i^c.) Thus, the degree d_v of each node v in ith chuck is known to processor i. Every processor i computes $f(v)$ for all nodes v in the ith chunk in parallel. Computation of $f(v)$ for different schemes are given below.

- $f(v) = 1$: This function requires no computation and assigns an equal number of core vertices to each processor.
- $f(v) = d_v$: This function also does not require any computation as d_v is already known to processor i. This scheme assigns an equal number of edges to each processor.
- $f(v) = \hat{d}_v$: Processor i needs degrees of the neighbors of v to compute $\hat{d}_v$. Some nodes $u \in N_v$ might reside in some other processors. Processor i communicates with those processors to find d_u.
- $f(v) = d_v \hat{d}_v$: The computation of $d_v \hat{d}_v$ requires the computation of d_v and $\hat{d}_v$, which have been discussed above.
- $f(v) = \hat{d}_v^{\,2}$: It requires computation of $\hat{d}_v$, which is

done as described above.

- $f(v) = \sum_{u \in N_v} (\hat{d_v} + \hat{d_u})$: This function gives the best estimation of the counting cost. However, computing this function requires two levels of communications, as described below.
  i. Computing $\hat{d_v}$: discussed above.
  ii. Computing $\hat{d_u}$ for nodes $u \in N_v$: once $\hat{d_v}$ is computed for all v in ith chunk by all processor i, this processor communicates with processors j to obtain $\hat{d_u}$ where u is in the jth chunk.

Computing partitions:
Given that each processor i knows $f(v)$ for all v in ith chunk as described above, our goal is to partition V into P disjoint subsets V_i^c such that $\sum_{v \in V_i^c} f(v) \approx \frac{1}{P} \sum_{v \in V} f(v)$. Assuming the nodes in V are labeled as $0, 1, 2, \ldots, n-1$ in this order, first the cumulative sum $g(v) = \sum_{k=0}^{v} f(k)$ for each $v \in V$ is computed using a parallel algorithm given in [2] and summarized below:

i. Let the nodes in the ith chunk are $n_i, n_i+1, \ldots, n_{i+1}-1$. Each processor i finds *local sum* $S_i = \sum_{v=n_i}^{n_{i+1}-1} f(v)$.
ii. The processors compute cumulative local sums R_i as follow: processor 0 sets $R_0 = S_0$ and sends R_0 to Processor 1; any other processor i waits until it receives R_{i-1} from processor $i-1$. Once R_{i-1} is received, it computes $R_i = R_{i-1} + S_i$ and sends R_i to processor $i+1$.
iii. Each processor i computes cumulative sum $g(v)$ as follow: $g(n_i) = R_{i-1} + f(n_i)$ and $g(v) = g(v-1) + f(v)$ for $n_i < v < n_{i+1}$.

Next we show how the partitions V_i^c can be computed from the cumulative sums $g(v)$ for all $v \in V$. Notice that $R_{P-1} = \sum_{v \in V} f(v)$. Processor $(P-1)$ computes $\alpha = \frac{1}{P} \sum_{v \in V} f(v) = \frac{1}{P} R_{P-1}$ and broadcast α to all other processors. Each processors i finds the boundary nodes x_j in its chunk: node x_j is the jth boundary node if and only if $g(x_i - 1) \leq i\alpha \leq g(x_i)$. Processor i can find the boundary nodes in the ith chunk by simply scanning the cumulative sum $g(v)$ for the nodes in the ith chunk. Notice that some chunks may have multiple boundary nodes and some chunks may not have any. For each boundary node x_j found in the ith chunk, processor i sends messages containing $x_j - 1$ and x_j to processor $j-1$ and j, respectively. Processor j receives two messages containing x_j and $x_{j+1} - 1$. Then partition V_j^c is the set of nodes $\{x_j, x_j+1, \ldots, x_{j+1}-1\}$.

Since scheme $\mathbb{DPD}$ requires two levels of communication for computing $f(.)$, it has the largest cost among the load balancing schemes . Computing $f(.)$ for $\mathbb{DPD}$ requires $O(\frac{m}{P} + P \log P)$ time. Computing partitions has a runtime complexity of $O(\frac{m}{P} + P)$. Therefore, the load balancing cost of $\mathbb{DPD}$ is given by $O(\frac{m}{P} + P \log P)$. Figure 9 shows an experimental result of the load balancing cost for different schemes on the LiveJournal network. Scheme $\mathbb{N}$ has the lowest cost and $\mathbb{DPD}$ the highest. Schemes $\mathbb{DH}$, $\mathbb{DH}^2$, and $\mathbb{DDH}$ have a quite similar load balancing cost.

4.5 Performance Analysis

In this section, we present the experimental results evaluating the performance of PATRIC and the load balancing schemes given in Section 4.4. We also compare the performance of PATRIC with the only other known distributed-memory parallel algorithm [30] for counting triangles.

4.5.1 Strong Scaling

Strong scaling of a parallel algorithm shows how much speedup a parallel algorithm gains as the number of processors increases. Figure 10 shows strong scaling of PATRIC on LiveJournal, Miami and Twitter networks with different load balancing schemes. The speedup factors of all load balancing schemes are almost equal on Miami network. Schemes $\mathbb{N}$ and $\mathbb{D}$ have a little better speedup than the others. On the contrary, for LiveJournal and Twitter networks, speedup factors for different load balancing schemes vary quite significantly. Schemes $\mathbb{DPD}$ and $\mathbb{DH}^2$ achieve better speedup than the other schemes for these networks. As discussed before, Miami is a network with an almost even degree distribution. Thus, all load balancing schemes, even simpler schemes like $\mathbb{N}$ and $\mathbb{D}$, distribute loads equally among processors (Figure 11). This produces an almost same speedup on Miami network with all load balancing schemes. A lower load balancing cost of schemes $\mathbb{N}$ and $\mathbb{D}$ (as shown in Figure 9) yields a little higher speedup. Unlike Miami network, LiveJournal and Twitter have a very skewed degree distribution. As a result, partitioning the network based on number of nodes ($\mathbb{N}$) or degree ($\mathbb{D}$) do not provide good load balancing. The other schemes, especially $\mathbb{DPD}$, capture the computational load more precisely and produce a very even load distribution among processors (Figure 11). In fact, scheme $\mathbb{DPD}$ provides the best speedup for LiveJournal and Twitter networks. Our subsequent results will be based on the scheme $\mathbb{DPD}$ since it performs better than other schemes on real world networks with skewed degree distribution.

4.5.2 Scaling with Network Size

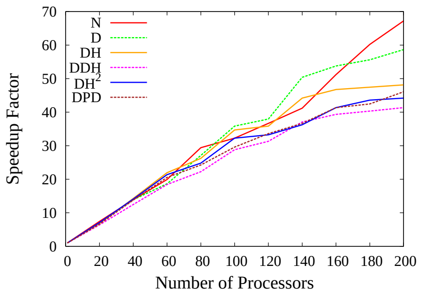
The load-balancing cost of our algorithm, as shown in Section 4.4, is $O(m/P + P \log P)$. For the algorithm given in Figure 5, the counting cost is $O(\sum_{v \in V_i^c} \sum_{u \in N_v} (\hat{d_u} + \hat{d_v}))$. Thus, the total computational cost of our algorithm is,

$$\begin{aligned} F(P) &= O(m/P + P \log P + \max_i \sum_{v \in V_i^c} \sum_{u \in N_v} (\hat{d_u} + \hat{d_v})) \\ &\approx c_1 m/P + c_2 P \log P + c_3 \max_i \sum_{v \in V_i^c} \sum_{u \in N_v} (\hat{d_u} + \hat{d_v}), \end{aligned}$$

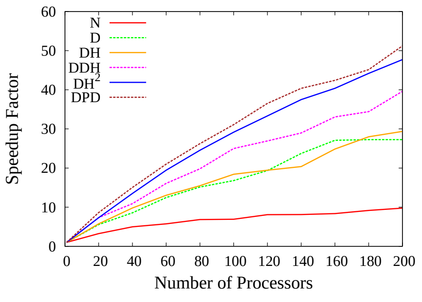
where c_1, c_2, and c_3 are constants. Now, quantity denoting computation cost, $(c_1 m/P + c_3 \sum_{v \in V_i^c} \sum_{u \in N_v} (\hat{d_u} + \hat{d_v}))$, decreases with the increase of P, but communication cost $P \log P$ increases with P. Thus, initially when P increases, the overall runtime decreases (hence the speedup increases). But, for some large value of P, the term $P \log P$ becomes dominating, and the overall runtime increases with the addition of further processors, as seen for $PA(5M, 50)$ network in Figure 12.

Notice that communication cost $P \log P$ is independent of network size. Therefore, when networks grow larger, computation cost increases, and hence they scale to a higher number of processors, as shown in Figure 12. This is, in fact, a highly desirable behavior of our parallel algorithm which is designed for real world massive networks. We need large number of processors when the network size is large and computation time is high.

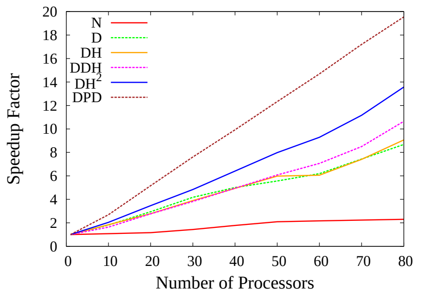
Consequently, there is an optimal value of P (P_{opt}) for which the total time $F(P)$ drops to its minimum and the speedup reaches to maximum. To have an estimation of P_{opt}, we replace d and $\hat{d}$ with average degree $\bar{d}$ and $\bar{d}/2$, re-



(a) Miami network

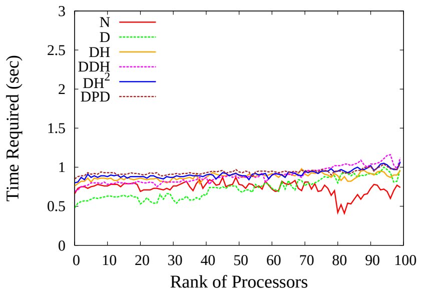


(b) LiveJournal network

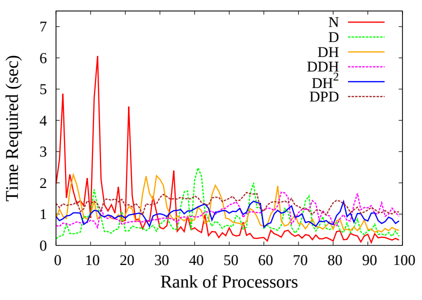


(c) Twitter network

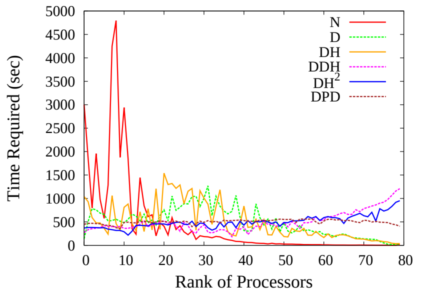
Figure 10: Speedup gained from different load balancing schemes for LiveJournal, Miami and Twitter networks.



(a) Miami network



(b) LiveJournal network



(c) Twitter network

Figure 11: Load distribution among processors for LiveJournal, Miami and Twitter networks by different schemes.

spectively, and have $F(P) \approx c_1 n\bar{d}/P + c_2 P \log P + c_3 n\bar{d}^2/P$. At the minimum point, $\frac{\mathrm{d}}{\mathrm{d}P}(F(P)) = 0$, which gives the following relationship of P_{opt}, n and $\bar{d}$: $P^2(1+\log P) = \frac{n}{c_2}(c_3\bar{d}^2 + c_1\bar{d})$. Thus, P_{opt} has roughly a linear relationship with $\sqrt{n}$ and $\bar{d}$. If a network with number of nodes n' and average degree $\bar{d}'$ experimentally shows an optimal P of P'_{opt}, then another network with n nodes and an average degree d has an approximate $P_{opt} \approx P'_{opt}\frac{\bar{d}}{\bar{d}'}\sqrt{\frac{n}{n'}}$. Thus, if we compute P'_{opt} experimentally by trial and error for a smaller network, we can estimate P_{opt} for all other networks.

4.5.3 *Weak Scaling*

Weak scaling of a parallel algorithm shows the ability of the algorithm to maintain constant computation time when the problem size grows proportionally with the increasing number of processors. We use PA(n, m) networks for this experiment, and for P processors, we use network PA$(P/10*1M, 50)$. The weak scaling of PATRIC is shown in Figure 13. Counting cost remains almost constant, whereas the load-balancing cost increases slowly with the increase of processors. Although the problem size per processor remains same, the addition of processors causes communication overhead to increase. Taking this fact into account, the weak scaling of PATRIC is very good.

4.5.4 *Comparison with Previous Algorithms*

The runtime of PATRIC on the mentioned networks are shown in Table 4. We compare the running of PATRIC with a very recent distributed-memory parallel algorithm for counting triangles given in [30]. We select three of the five networks used in [30]. Twitter and LiveJournal are the two largest among the networks used in [30]. We also selected web-BerkStan which is a bad behaving network with a very skewed degree distribution. No artificial network is used in [30]. For all of these three networks, PATRIC is more than 45 times faster than the algorithm in [30]. This huge improvement of PATRIC over [30] is due to the fact that their algorithm generates a huge volume of intermediate data, which are all possible 2-paths centered at each node. The amount of such intermediate data can be significantly larger than the original network. For example, for the Twitter network, 300B 2-paths are generated while there are only 2.4B edges in the network. The algorithm in [30] shuffles and regroups these 2-paths, which take significantly larger time and also memory. To deal with the memory issue, they proposed a complicated partitioning scheme, which improves the memory issue to some extent; however, it does not improve the running time (see [30] for details).

Table 4: Runtime Performance of PATRIC using 200 processors and the algorithm in [30].

Networks	**Runtime (sec.)**		**Triangles**
	PATRIC	[30]	
Twitter	9.4m	423m	34.8B
web-BerkStan	0.10s	1.70m	65M
LiveJournal	0.8s	5.33m	286M
Miami	0.6s	–	332M
PA(1B, 20)	15.5m	–	0.403M

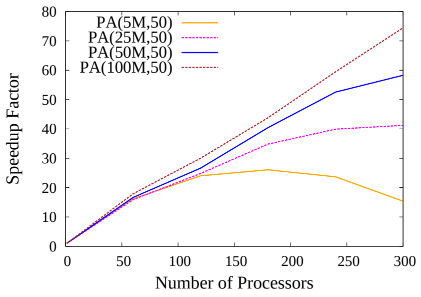


Figure 12: Increase of scalability with increase of network size.

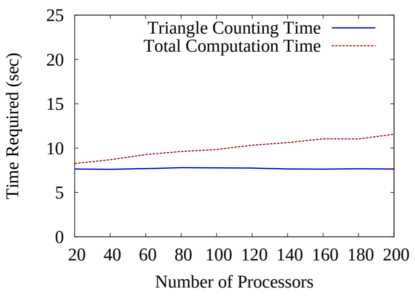


Figure 13: Weak scaling on PA($P/10*1M,50$) networks.

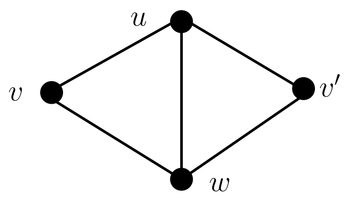


Figure 14: Two triangles (v,u,w) and (v',u,w) with an overlapping edge.

Table 5: Accuracy of our parallel sparsification algorithm and DOULION [31] with $p=0.1$. Our parallel algorithm was run with 100 processors. Variance, max error and average error are calculated from 25 independent runs for each of the algorithms.

Networks	**Variance**		**Avg. error (%)**		**Max error (%)**	
	Our Alg.	DOULION	Our Alg.	DOULION	Our Alg.	DOULION
web-BerkStan	1.287	2.027	0.389	0.392	1.02	1.08
LiveJournal	1.770	1.958	1.46	1.86	3.88	4.75

5. A SPARSIFICATION-BASED PARALLEL APPROXIMATION ALGORITHM

In this section, we integrate a sparsification technique, called DOULION, proposed in [31] with our parallel algorithm. Our adopted version of DOULION provides more accuracy than DOULION. Sparsification of a network is a sampling technique where some randomly chosen edges are retained and the rest are deleted, and then computation is performed in the sparsified network. Sparsification of a network saves both computation time and memory space and provides an approximate result.

Let $G(V,E)$ and $G'(V,E'\subset E)$ be the networks before and after sparsification, respectively. Network $G'(V,E')$ is obtained from $G(V,E)$ by retaining each edge, independently, with probability p and removing with probability $1-p$. Now any algorithm can be used to find the exact number of triangles in G'. Let $T(G')$ be the number of triangles in G'. The estimated number of triangles in G is given by $\frac{1}{p^3}T(G')$, which is an unbiased estimation since $E\left[\frac{1}{p^3}T(G')\right]=T(G)$.

As shown in [31], the variance of the estimated number of triangles is

$$\text{Var} = \left(\frac{1}{p^3}-1\right)T(G)+2k\left(\frac{1}{p}-1\right), \tag{3}$$

where k is the number of pairs of triangles in G with an overlapping edge (see Figure 14).

In our parallel algorithm, sparsification is done as follows: each processor i independently performs sparsification on its partition $G_i(V_i,E_i)$. While loading partition G_i into its local memory, it retains each edge $(u,v)\in E_i$ with probability p and discards it with probability $1-p$ as shown Figure 15. If T' is the number of triangles obtained after sparsification, $\frac{1}{p^3}T'$ is the estimated number of triangles in G.

Notice that the sparsification of our algorithm is not exactly the same as that of DOULION. Consider two triangles (v,u,w) and (v',u,w) with an overlapping edge (u,w) as shown in Fig. 14. In DOULION, if edge (u,w) is not retained, none of the two triangles survive, and as a result, survivals of (v,u,w) and (v',u,w) are not independent events. Now, in our case, if v and v' are core nodes in two different partitions G_i and G_j, processor i may retain edge (u,w) while processor j discards (u,w), and vice versa. As processor i and j perform sparsification independently, survivals of triangles (v,u,w) and (v',u,w) are independent events.

```
1: for $v \in V_i$ do
2:     for $(v,u) \in E$ do
3:         if $v \prec u$ then
4:             toss a biased coin with success prob. $p$
5:             if success then
6:                 store $u$ to $N_v$
7: $T_i \leftarrow$ count of triangles
8: Find Sum $T' = \sum_i T_i$ using MpiReduce
9: $T \leftarrow \frac{1}{p^3} \times T'$
```

Figure 15: Triangle counting with parallel sparsification

However, our estimation is also unbiased, and in fact, this difference (with DOULION) improves the accuracy of the estimation by our parallel algorithm. Since the probability of survival of any triangle is still exactly $\frac{1}{p^3}$, we have $E\left[\frac{1}{p^3}T'\right]=T$. To calculate variance of the estimation, let k'_i be the number of pairs of triangles with an overlapping edge such that both triangles are in partition G_i, and $k'=\sum_i k'_i$. Let k'' be the number of pairs of triangles (v,u,w) and (v',u,w) with an overlapping edge (u,w) (as shown in Fig. 14) and v and v' are core nodes in two different partitions. Then clearly, $k'+k''=k$ and $k'\le k$. Now following the same steps as in [31], one can show that the variance of our estimation is

$$\text{Var}' = \left(\frac{1}{p^3}-1\right)T(G)+2k'\left(\frac{1}{p}-1\right). \tag{4}$$

Comparing Equations 3 and 4, if $k''>0$, we have $k'<k$ and reduced variance leading to improved accuracy. This

Table 6: Comparison of our parallel sparsification algorithm and DOULION [31] on LiveJournal network with 100 processors.

Metrics	p	0.1	0.2	0.3	0.4	0.5
Accuracy	Our Alg.	99.9914	99.9917	99.9924	99.9936	99.9971
	DOULION	99.6310	99.7544	99.8392	99.9121	99.9584
Speedup	Our Alg.	57.88	24.36	11.04	6.19	4.0
	DOULION	30.96	11.96	6.71	4.31	3.03

observation is verified by experimental results on two real-world networks as shown in Table 5. This observation also suggests that accuracy can be improved with a larger number of processors.

In [31], it was shown that due to sparsification with parameter p, the computation can be faster as much as $1/p^2$ times. However, in practice the speed up is typically smaller than $1/p^2$ but it always seems to be larger than $1/p$. Table 6 shows the accuracy and speedup factor with varying p ranging from 0.1 to 0.5 for the LiveJournal network. The speedup factor, due to sparsification, of our algorithm is better than that of DOULION. For the LiveJournal network, DOULION shows a speedup of 31 with $p = 0.1$ while our algorithm has a speedup of 58.

6. CONCLUSION

We presented a parallel algorithm, called PATRIC, for counting triangles in a massive network. This parallel algorithm can work with networks that have billions of nodes and edges. Such capability of PATRIC will enable various types of analysis of massive real-world networks, networks that otherwise do not fit in the main memory of a single processor. PATRIC shows very good scalability with both the number of processors and the problem size and performs well on both real-world and artificial networks. PATRIC has been able to count triangles of a massive network with $1B$ nodes and $10B$ edges in 16 minutes using 40 processors. We presented several load balancing schemes and showed that such schemes provide very good balancing. Further, we have adopted the sparsification approach of DOULION in our parallel algorithm with improved accuracy. This adoption will allow us to deal with even larger networks.

7. REFERENCES

[1] Twitter Data. `http://an.kaist.ac.kr/~haewoon/release/twitter_social_graph`, 2010. [Online].

[2] M. Alam and M. Khan. Efficient algorithms for generating massive random networks. Technical Report 13-064, NDSSL at Virginia Tech, May 2013.

[3] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.

[4] C. Apte, B. Liu, E. Pednault, and P. Smyth. Business applications of data mining. *Commun. ACM*, 45(8):49–53, Aug. 2002.

[5] 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.

[6] A. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[7] C. Barrett, R. Beckman, et al. Generation and analysis of large synthetic social contact networks. In *WSC*, 2009.

[8] 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.

[9] B. Bollobas. *Random Graphs.* Cambridge Univ. Press, 2001.

[10] J. Chen and S. Lonardi. *Biological Data Mining.* Chapman & Hall/CRC, 1st edition, 2009.

[11] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Proc. of KDD*, 2011.

[12] F. Chung and L. Lu. *Complex Graphs and Networks.* American Mathematical Society, Aug. 2006.

[13] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proc. of STOC*, 1987.

[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI*, 2004.

[15] J. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proc. Natl. Acad. of Sci. USA*, 99(9):5825–5829, 2002.

[16] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[17] M. Girvan and M. Newman. Community structure in social and biological networks. *Proc. Natl. Acad. of Sci. USA*, 99(12):7821–7826, June 2002.

[18] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. In *Proc. of STOC*, 1977.

[19] M. Kolountakis, G. Miller, R. Peng, and C. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. In *Proc. of WAW*, 2010.

[20] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proc. of WWW*, 2010.

[21] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407:458–473, 2008.

[22] M. McPherson, L. Smith-Lovin, and J. Cook. Birds of a feather: Homophily in social networks. *Annual Rev. of Soc.*, 27(1):415–444, 2001.

[23] R. Milo, S. Shen-Orr, et al. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, October 2002.

[24] M. Newman. Coauthorship networks and patterns of scientific collaboration. *Proc. Natl. Acad. of Sci. USA*, 101(1):5200–5205, April 2004.

[25] R. Pagh and C. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112(7):277–281, 2012.

[26] T. Schank. *Algorithmic Aspects of Triangle-Based Network Analysis.* PhD thesis, University of Karlsruhe, 2007.

[27] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proc. of Exp. and Efficient Algorithms*, 2005.

[28] C. Seshadhri, A. Pinar, and T. Kolda. Triadic measures on graphs: the power of wedge sampling. In *Proc. of SDM*, 2013.

[29] SNAP. Stanford network analysis project. `http://snap.stanford.edu/`, 2012.

[30] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proc. of WWW*, 2011.

[31] C. Tsourakakis, U. Kang, G. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proc. of KDD*, 2009.