Fast Parallel Algorithms for Counting and Listing Triangles in Big Graphs

SHAIKH ARIFUZZAMAN, University of New Orleans
MALEQ KHAN, Texas A&M University–Kingsville
MADHAV MARATHE, University of Virginia

Big graphs (networks) arising in numerous application areas pose significant challenges for graph analysts as these graphs grow to billions of nodes and edges and are prohibitively large to fit in the main memory. Finding the number of triangles in a graph is an important problem in the mining and analysis of graphs. In this article, we present two efficient MPI-based distributed memory parallel algorithms for counting triangles in big graphs. The first algorithm employs overlapping partitioning and efficient load balancing schemes to provide a very fast parallel algorithm. The algorithm scales well to networks with billions of nodes and can compute the exact number of triangles in a network with 10 billion edges in 16 minutes. The second algorithm divides the network into non-overlapping partitions leading to a space-efficient algorithm. Our results on both artificial and real-world networks demonstrate a significant space saving with this algorithm. We also present a novel approach that reduces communication cost drastically leading the algorithm to both a space- and runtime-efficient algorithm. Further, we demonstrate how our algorithms can be used to list all triangles in a graph and compute clustering coefficients of nodes. Our algorithm can also be adapted to a parallel approximation algorithm using an edge sparsification method.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; • **Computing methodologies → Massively parallel algorithms**; • **Information systems** → *Data mining*;

Additional Key Words and Phrases: Triangle-counting, clustering-coefficient, massive networks, parallel algorithms, social networks, graph mining

ACM Reference format:
Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. 2019. Fast Parallel Algorithms for Counting and Listing Triangles in Big Graphs. *ACM Trans. Knowl. Discov. Data* 14, 1, Article 5 (December 2019), 34 pages.
https://doi.org/10.1145/3365676

This work has been partially supported by DTRA CNIMS Contract HDTRA1-11-D-0016-0001, DTRA Grant HDTRA1-11-1-0016, DTRA NSF NetSE Grant CNS-1011769 and NSF SDCI Grant OCI-1032677. In addition, Dr. S. Arifuzzaman was also partially supported by Louisiana Board of Regents RCS Grant LEQSF(2017-20)-RDA-25. Most of this work was done and a preliminary version of this article was prepared when the authors were at the Biocomplexity Institute of Virginia Tech. Additionally, Shaikh Arifuzzaman and Madhav Marathe were with the Department of Computer Science, Virginia Tech, Blacksburg, VA 24060. Some preliminary results of the work presented in this article have appeared in the proceedings of CIKM 2013 [8] and HPCC 2015 [9]. A link of triangle counting code of this article can be found on: https://www.cs.uno.edu/~arif/research.html.
Authors' addresses: S. Arifuzzaman, Computer Science Department, University of New Orleans, 2000 Lakeshore Drive, Math 349, New Orleans, LA 70122; email: smarifuz@uno.edu; M. Khan, Department of Electrical Engineering & Computer Science, Texas A&M University–Kingsville, 700 University Blvd. Kingsville, TX 78363; email: maleq.khan@tamuk.edu; M. Marathe, Department of Computer Science, University of Virginia, 85 Engineer's Way, Charlottesville, VA 22904; email: mvm7hz@virginia.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1556-4681/2019/12-ART5 $15.00
https://doi.org/10.1145/3365676

Scansionato con CamScanner

1 INTRODUCTION

Counting triangles in a graph is a fundamental and important algorithmic problem in graph analysis, and its solution can be used in solving many other problems such as the computation of clustering coefficient (CC), transitivity, and triangular connectivity [18, 34]. Existence of triangles and the resulting high CC in a social network reflect some common theories of social science, e.g., *homophily* where people become friends with those similar to themselves and *triadic closure* where people who have common friends tend to be friends themselves [33]. Further, triangle counting has important applications in graph mining such as detecting spamming activity and assessing content quality in social networks [14], uncovering the thematic structure of the web [20], query planning optimization in databases [11], and detecting communities or clusters in social and information networks [41].

Graph is a powerful abstraction for representing underlying relations in large unstructured datasets. Examples include the web graph [17], various social networks [29], biological networks [23], and many other information networks. In the era of big data, the emerging graph data is also very large. Social networks such as Facebook and Twitter have millions to billions of users [18, 53]. Such big graphs motivate the need for efficient parallel algorithms. Furthermore, these massive graphs pose another challenge of a large memory requirement. These graphs may not fit in the main memory of a single processing unit, and only a small part of the graph is available to a processor.

Counting triangles and related problems such as computing CCs have a rich history [5, 24, 30, 38, 44, 46, 49, 52]. Despite the fairly large volume of work addressing this problem, only recently has attention been given to the problems associated with big graphs. Several techniques can be employed to deal with such graphs: streaming algorithms [1, 2, 5, 14, 26, 50], sparsification based algorithms [52, 56], external-memory algorithms [18], and parallel algorithms [27, 49, 50]. The streaming and sparsification based algorithms are approximation algorithms [21]. Note that approximation algorithms provide an overall (global) estimate of the number of triangles in the graph, which might not be used to count triangles incident on individual nodes (local triangles) with reasonable accuracy. Thus, certain local patterns such as local CC distribution can not be computed with approximation algorithms. Exact algorithms are necessary to discover such local patterns. External memory algorithms can provide exact solution, however they can be very I/O intensive leading to a large runtime. Efficient parallel algorithms can solve the problem of a large runtime by distributing computing tasks to multiple processors. Over the last couple of years, several parallel algorithms, both shared memory and distributed memory (MapReduce or MPI) based, have been proposed.

A shared memory parallel algorithm is proposed in [50] for counting triangles in a streaming setting. The algorithm provides approximate counts. The article reports scalability using only 12 cores. Two other shared memory algorithms have been presented recently in [42, 46]: the reported speedups with the first algorithm vary between 17 and 50 with 64 cores. The second article reports speedups using only 32 cores, and the obtained speedups are due to both approximation and parallelization. Tom et al. [51] provide shared-memory based optimization of triangle counting algorithms for Intel Haswell and KNL processors. Although these algorithms are useful, shared memory systems with a large number of processors and at the same time sufficiently large memory per processor are not widely available. Further, the overhead for locking and synchronization mechanism required for concurrent read and write access to shared data might restrict their scalability. A GPU-based parallel algorithm is proposed recently in [24] which achieves a speedup of only 32 with 2,880 streaming processors.

There exist several algorithms based on the MapReduce framework. Suri et al. presented two algorithms for counting the exact number of triangles [49]. The first algorithm generates huge

Scansionato con CamScanner

volumes of intermediate data and requires a significantly large amount of time and memory. The second algorithm suffers from redundant counting of triangles. Two papers by Park et al. [36, 38] achieved some improvement over the second algorithm of Suri et al., although the redundancy is not entirely eliminated. Another MapReduce based parallelization of a wedge-based sampling technique is proposed in [27], which is also an approximation algorithm.

MapReduce framework provides several advantages such as fault tolerance, abstraction of parallel computing mechanisms, and ease of developing a quick prototype or program. However, the overhead for doing so results in a larger runtime. On the other hand, MPI-based systems provide the advantages of defining and controlling parallelism from a granular level, implementing application specific optimizations, such as load balancing, memory, and message optimization.

In this article, we present fast algorithms for counting the *exact* number of triangles. Our algorithms store a small portion of input graph in the main memory of each processor and can work on big graphs. Below are the summaries of our contributions.

(i) *A fast parallel algorithm:* We propose an MPI based parallel algorithm that employs an overlapping partitioning scheme and a novel load balancing scheme. The overlapping partitions eliminate the need for message exchanges leading to a fast algorithm. The algorithm scales almost linearly with the number of processors, and is able to process a graph with 1 billion nodes and 10 billion edges in 16 minutes.

(ii) *A space-efficient parallel algorithm:* We present a space-efficient MPI based parallel algorithm that divides the graph into non-overlapping partitions and achieves a significant space efficiency over the first algorithm. This algorithm requires inter-processor communications to count a certain type of triangles. However, we present a novel approach that reduces communication cost drastically 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 cost function offers additional runtime efficiency to the algorithm.

(iii) *Sequential algorithm and node ordering:* We show, both theoretically and experimentally, a simple modification of a state-of-the-art sequential algorithm for counting triangles improves its performance and use this modified algorithm in the development of our parallel algorithm. We also present a proof of the optimal node ordering that minimizes the computational cost of this sequential algorithm.

(iv) *Parallel computation of clustering coefficients:* In a sequential setting, an algorithm for counting triangles can be directly used for computing CCs of the nodes by simply keeping the counts of triangles for each node individually. However, in a distributed-memory parallel system, combining the counts from all processors for all nodes poses another level of difficulty. We show how our algorithm for triangle counting can be used to compute CCs in parallel.

(v) *Parallel approximation using sparsification technique:* Although we present algorithms for counting the exact number of triangles in massive graphs, our algorithm can be used for approximate counting in conjunction with an edge sparsification technique [52]. We show how this technique can be adapted to our parallel algorithms and that our parallel sparsification improves the accuracy of the approximation over the sequential sparsification [52].

Organization. The rest of the article is organized as follows. The preliminary concepts, notations, and datasets are briefly described in Section 2. We discuss sequential algorithms for counting triangles and present a proof for the optimal node ordering in Sections 3 and 4, respectively. Our parallel algorithms for counting triangles are presented in Sections 5 and 6. The parallelization of the sparsification technique is given in Section 7. We show in Section 8 how we can list all

Scansionato con CamScanner

Table 1. Dataset Used in Our Experiments

Network	Nodes	Edges	Source
Email-Enron	37K	0.36M	SNAP [31]
web-Google	0.88M	5.1M	SNAP [31]
web-BerkStan	0.69M	6.5M	SNAP [31]
Miami	2.1M	50M	[13]
LiveJournal	4.8M	43M	SNAP [31]
Orkut	3.07M	117.2M	SNAP [31]
Twitter	42M	2.4B	[28]
Friendster	65.6M	1.8B	SNAP [31]
Gnp(n, d)	n	$\frac{1}{2}nd$	Erdős–Réyni [16]
PA(n, d)	n	$\frac{1}{2}nd$	Pref. Attachment [12]

K, M, and B denote thousands, millions, and billions, respectively.

triangles in graphs in parallel. Section 9 presents a parallel algorithm for computing CCs of nodes. We discuss some applications of counting triangles in Section 10, present a comparative discussion of some other recent work in Section 11 and conclude in Section 12.

2 PRELIMINARIES

The given graph is denoted by $G(V, E)$, where V and E are the sets of nodes (vertices) and edges, respectively, with $m = |E|$ edges and $n = |V|$ nodes labeled as $0, 1, 2, \ldots, n-1$. We assume the graph $G(V, E)$ is undirected. If $(u, v) \in E$, we say u and v are neighbors of each other. The set of all neighbors of $v \in V$ is denoted by $\mathcal{N}_v$, i.e., $\mathcal{N}_v = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |\mathcal{N}_v|$.

A triangle in G 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 the same as the number of edges among the neighbors of v, i.e., $T_v = |\{(u, w) \in E : u, w \in \mathcal{N}_v\}|$.

The 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

$$C_v = \frac{T_v}{\binom{d_v}{2}} = \frac{2T_v}{d_v(d_v - 1)}.$$

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 1. Miami [13] is a synthetic, but realistic, social contact network for Miami city. Twitter, LiveJournal, Orkut, Friendster, Email-Enron, web-BerkStan, and web-Google are real-world networks. Artificial network PA(n, d) is generated using the preferential attachment (PA) model [12] with n nodes and average degree d. Network Gnp(n, d) is generated using the Erdős–Réyni random graph model [16], also known as $G(n, q)$ model, with n nodes and edge probability $q = \frac{d}{n-1}$ so that the expected degree of each node is 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.

Scansionato con CamScanner

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

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

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 directly access the local memory of another processor, and the processors communicate via exchanging messages using MPI.

Experimental setup. We perform our experiments using a high performance computing cluster with 64 computing nodes (QDR InfiniBand interconnect), 16 processors (Sandy Bridge E5-2670, 2.6 GHz) per node, memory 64 GB per computing node, and operating system CentOS Linux 6.

3 SEQUENTIAL ALGORITHMS

In this section, we discuss sequential algorithms for counting triangles and show that a simple modification to a state-of-the-art algorithm improves both runtime and space requirement. Although the modification seems quite simple, and others might have used it previously, to the best of our knowledge, our analysis is the first to show that such modification improves the performance significantly. Our parallel algorithms are based on this improved algorithm.

A simple but efficient algorithm [44, 49] for counting triangles is as follows: 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. Many existing algorithms [18, 30, 44, 45, 49] provide significant improvement over the above method. A very comprehensive survey of the sequential algorithms can be found in [30, 44]. One of the state of the art algorithms, known as NodeIterator++, as identified in two recent papers [18, 49], is shown in Figure 1. Both [18] and [49] use this algorithm as a basis of their external-memory and parallel algorithm, respectively.

The algorithm NodeIterator++ 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 among the six possible permutations. However, NodeIterator++ incorporates an interesting node ordering based on the degrees of the nodes, with ties broken by node IDs, defined as follows:

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

Definition 3.1 (effective degree). While N_v is the set of all neighbors of $v \in V$, let $N_v = \{u \in V | (u, v) \in E \wedge v \prec u\}$, i.e., N_v is the set of neighbors u of v such that $v \prec u$. We define $\hat{d}_v = |N_v|$ as the effective degree of v.

The degree based ordering can improve the running time. Assuming N_v, for all v, are sorted and a binary search is used to check $(u, w) \in E$, a runtime of $O\left(\sum_v (\hat{d}_v d_v + \hat{d}_v^2 \log d_{\max})\right)$ can be shown, where $d_{\max} = \max_v d_v$. This runtime 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, variance of the $\hat{d}_v$ values are reduced significantly. We also

Scansionato con CamScanner

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

Fig. 2. Algorithm NodeIteratorN, a modification of NodeIterator++.

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 as discussed in detail in Section 5.

A simple modification of NodeIterator++ is as follows: 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 $O(m)$ from $\sum_v \hat{d}_v d_v$ and allows us to use an efficient set intersection operation. 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 3.2.

THEOREM 3.2. *Algorithm NodeIteratorN counts each triangle in G once and only once.*

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 construction 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 because $x_1 \notin N_{x_2}$ and $x_1, x_2 \notin N_{x_3}$. □

In NodeIteratorN, 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 3.3, in contrast to $O\left(\sum_v (\hat{d}_v d_v + \hat{d}_v^2 \log d_{\max})\right)$ for NodeIterator++.

THEOREM 3.3. *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 $O\left(\sum_v d_v\right) = O(m)$, and sorting these N_v requires $O\left(\sum_v \hat{d}_v \log \hat{d}_v\right)$ 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

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

Scansionato con CamScanner

Table 2. Running Time for Sequential Algorithms

Networks	Runtime (sec.)		Triangles
	NodeIterator++	NodeIteratorN	
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

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. □

Notice that the 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 [18] (Page 674):

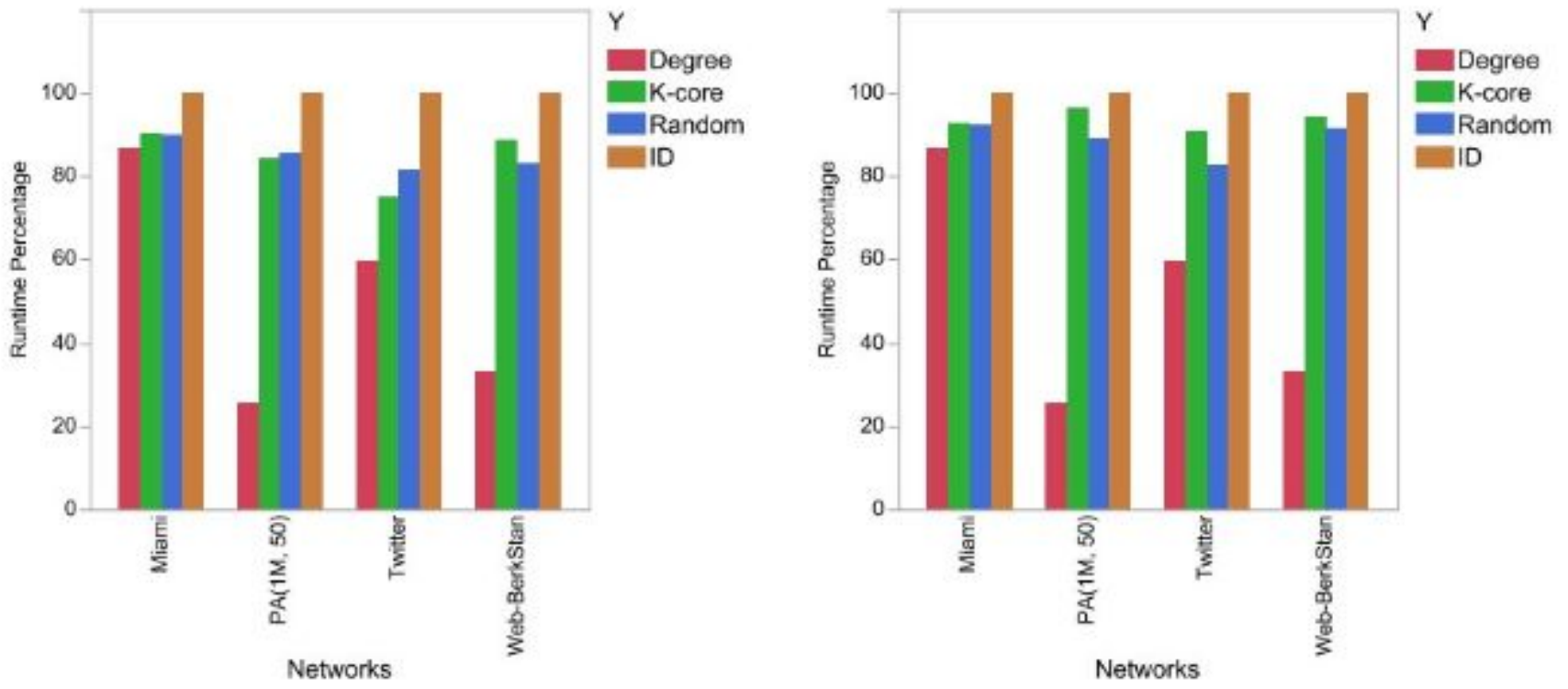
```
1: S ← N_v ∩ N_u
2: for w ∈ S and u ≺ w do
3:     T ← T + 1
```

However, with this set intersection operation, the runtime of NodeIterator++ is $O\left(\sum_v d_v^2\right)$ since $|N_v| = d_v$, 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 the two algorithms presented in [30, 45] take the same asymptotic time complexity as NodeIteratorN. However, the algorithm in [45] requires three times more memory than NodeIteratorN. The algorithm in [30] 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. Our experimental results show that NodeIteratorN is significantly faster than NodeIterator++ for both real-world and artificial networks as presented in Table 2.

4 AN OPTIMAL NODE ORDERING

A total ordering $\prec$ of the nodes helps avoid duplicate counts of the same triangle. Any ordering of the nodes, e.g., ordering based on node IDs, random ordering, k-coreness based ordering, make sure each triangle is counted exactly once. By avoiding duplicate counts, these orderings also improve running time of the algorithm. However, different orderings lead to different runtimes. Figure 3 shows the runtime of our sequential algorithm for triangle counting with four orderings of nodes: ordering based on node IDs, degree, k-coreness, and random ordering. Node IDs and degrees are readily available with network data and do not require any additional computation. On the other hand, k-coreness based ordering requires computing coreness of nodes, and for random ordering, we generate n random numbers. Figure 3(a) shows the comparison of runtime of counting triangles without considering the cost for computing orderings. Figure 3(b) shows the comparison with total runtime of counting triangles and computing orderings. In both cases, degree based ordering provides the best runtime efficiency among all orderings. For networks with relatively even degree distribution such as Miami, all the orderings provide similar runtimes. However, for networks with skewed degree distribution, degree based ordering provides the least runtime. In our datasets, nodes with large degrees somehow appear at the beginning (having smaller IDs)

Scansionato con CamScanner



(a) Runtime for triangle counting without considering the runtime for computing ordering of nodes

(b) Total runtime for counting triangles and computing ordering of nodes

Fig. 3. Comparison of runtime of sequential triangle counting (NodeIteratorN) with four distinct orderings of nodes. For each network, we compute the percentage of runtime with respect to the maximum runtime given by any of these orderings. In all cases, the degree based ordering gives the least runtime. Note that we compute the average runtime from 25 independent runs for the random ordering.

giving ID based ordering almost the opposite effect of degree based ordering. As a result, ID based ordering provides the largest runtime for our datasets.

Now that our experimental results show degree based ordering provides the best runtime efficiency, next we show in Theorem 4.3 that the degree based ordering is, in fact, the optimal ordering that minimizes the runtime of algorithm *NodeIteratorN*.

We denote the degree based ordering as $\prec_{\mathcal{D}}$, which is defined as follows:

$$u \prec_{\mathcal{D}} v \iff d_u < d_v \text{ or } (d_u = d_v \text{ and } u < v). \tag{2}$$

Assume there is another total ordering $\prec_{\mathcal{K}}$ based on some quantity k_v of nodes v:

$$u \prec_{\mathcal{K}} v \iff k_u < k_v \text{ or } (k_u = k_v \text{ and } u < v). \tag{3}$$

We now define a function that quantifies how ordering $\prec_{\mathcal{K}}$ agrees with $\prec_{\mathcal{D}}$ on the relative order of $x, y \in V$.

Definition 4.1 (Agreement function Y). The *agreement* function $Y : V \times V \to \mathbb{Z}$ is defined as follows:

$$Y(x,y) = \begin{cases} -1, & \text{if } (x,y) \in E \text{ and } x \prec_{\mathcal{D}} y \text{ and } y \prec_{\mathcal{K}} x \\ 1, & \text{if } (x,y) \in E \text{ and } y \prec_{\mathcal{D}} x \text{ and } x \prec_{\mathcal{K}} y \\ 0, & \text{Otherwise} \end{cases}$$

It is, then, easy to see that $Y(x,y) = -Y(y,x)$.

We now prove an important result in the following lemma, which we subsequently use in Theorem 4.3.

Lemma 4.2. *For any* $(x,y) \in E$, $Y(x,y)(d_x - d_y) \geq 0$.

Proof. Let $c_{xy} = Y(x,y)(d_x - d_y)$. If orderings $\prec_{\mathcal{K}}$ and $\prec_{\mathcal{D}}$ agree on the relative order of x and y, then $Y(x,y) = 0$ by definition, and hence, $c_{xy} = 0$. Otherwise, consider the following three cases.

Scansionato con CamScanner

– $d_x = d_y$: This gives $d_x - d_y = 0$, and thus, $c_{xy} = 0$.
– $d_x < d_y$: We have $x \prec_{\mathcal{D}} y$ and $y \prec_{\mathcal{K}} x$, and thus, $Y(x, y) = -1$. Since $d_x - d_y < 0$, $c_{xy} > 0$.
– $d_x > d_y$: We have $y \prec_{\mathcal{D}} x$ and $x \prec_{\mathcal{K}} y$, and thus, $Y(x, y) = 1$. Since $d_x - d_y > 0$, $c_{xy} > 0$.

Therefore, for any $(x, y) \in E$, $c_{xy} = Y(x, y)(d_x - d_y) \geq 0$. □

THEOREM 4.3. *Degree based ordering $\prec_{\mathcal{D}}$ minimizes the runtime for counting triangles using algorithm* NodeIteratorN.

PROOF. Let $\hat{d}_v$ be the effective degree of vertex v with ordering $\prec_{\mathcal{D}}$. Then, the corresponding runtime for counting triangles is $\Theta\left(\sum_{i \in V} d_i \hat{d}_i\right)$. We provide a proof by contradiction. Assume that $\prec_{\mathcal{D}}$ is not an optimal ordering. Then, there exists another ordering $\prec_{\mathcal{K}}$ that leads to a lower runtime for counting triangles than that of $\prec_{\mathcal{D}}$. Let $\prec_{\mathcal{K}}$ yields an effective degree $\tilde{d}$, the corresponding runtime for counting triangles is $\Theta\left(\sum_{i \in V} d_i \tilde{d}_i\right)$. Let $C_{\mathcal{D}} = \sum_{i \in V} d_i \hat{d}_i$ and $C_{\mathcal{K}} = \sum_{i \in V} d_i \tilde{d}_i$. Then, we have $C_{\mathcal{K}} < C_{\mathcal{D}}$.
Now, using Definition 4.1, the effective degree $\tilde{d}_x$ of node x obtained by $\prec_{\mathcal{K}}$ can be expressed as

$$\tilde{d}_x = \hat{d}_x + \sum_{y \in N_x} Y(x, y).$$

Now, we have,

$$\begin{aligned}
C_{\mathcal{K}} &= \sum_{x \in V} d_x \tilde{d}_x \\
&= \sum_{x \in V} d_x \left(\hat{d}_x + \sum_{y \in N_x} Y(x, y) \right) \\
&= \sum_{x \in V} d_x \hat{d}_x + \sum_{x \in V} \left(d_x \sum_{y \in N_x} Y(x, y) \right) \\
&= \sum_{x \in V} d_x \hat{d}_x + \sum_{(x, y) \in E} \left(d_x Y(x, y) + d_y Y(y, x) \right) \\
&= \sum_{x \in V} d_x \hat{d}_x + \sum_{(x, y) \in E} Y(x, y) \left(d_x - d_y \right).
\end{aligned}$$

The second last step follows from rearranging terms of the second summation and distributing them over edges. The last step follows from the fact that $Y(y, x) = -Y(x, y)$. Now, from Lemma 4.2 we have, $Y(x, y)(d_x - d_y) \geq 0$ for any $(x, y) \in E$. Thus, $\sum_{(x, y) \in E} Y(x, y) \left(d_x - d_y \right) \geq 0$, and therefore,

$$C_{\mathcal{K}} \geq \sum_{x \in V} d_x \hat{d}_x = C_{\mathcal{D}}.$$

This contradicts our assumption of $C_{\mathcal{K}} < C_{\mathcal{D}}$. Therefore, degree based ordering $\prec_{\mathcal{D}}$ is an optimal ordering that minimizes the runtime for counting triangles of our algorithm. □

We use algorithm *NodeIteratorN* with degree based ordering in our parallel algorithms.

5 A FAST PARALLEL ALGORITHM WITH OVERLAPPING PARTITIONING

In this section, we present our fast parallel algorithm for counting triangles in massive graphs with overlapping partitioning and novel load balancing schemes.

Scansionato con CamScanner

```
1: Each processor P_i, in parallel, executes the following:(lines 2–4)
2: G_i(V_i, E_i) ← ComputePartition(G, i)
3: T_i ← CountTriangles(G_i, i)
4: Barrier
5: Find T = Σ_i T_i
6: return T
```

Fig. 4. The main steps of our fast parallel algorithm.

5.1 Overview of the Algorithm

We assume that the graph is massive and does not fit in the local memory of a single computing node. Only a part of the entire graph is available to a processor. Let p be the number of processors used in the computation. The graph is partitioned into p partitions, and each processor P_i is assigned one such partition $G_i(V_i, E_i)$ (formally defined below). P_i performs computation on its partition G_i. The main steps of our fast parallel algorithm are given in Figure 4. In the following subsections, we describe the details of these steps and several load balancing schemes.

5.2 Partitioning the Graph

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 the processors. For the input graph $G(V, E)$, processor P_i works on $G_i(V_i, E_i)$, which is a subgraph of G induced by V_i. The subgraph G_i is constructed as follows: First, set of nodes V is partitioned into p disjoint subsets $V_0^c, V_1^c, \ldots, V_{p-1}^c$, such that, for any j and k, $V_j^c \cap V_k^c = \emptyset$ and $\bigcup_k V_k^c = V$. Second, set V_i is constructed containing 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 $\{(u, v) : u \in V_i \text{ and } v \in N_u\}$.

Each processor P_i is responsible for counting triangles incident on the nodes in V_i^c. We call any node $v \in V_i^c$ a *core* node of partition i. Each $v \in V$ is a core node in exactly one partition. How the nodes in V are distributed among the core sets V_i^c for all P_i affect the load balancing and hence performance of the algorithm crucially. Later in Section 5.4, we present several load balancing schemes and the details of how sets V_i^c are constructed.

Now, P_i stores the set of neighbors 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 $x \notin V_i$. Such nodes x can be safely removed from N_w and the number of triangles incident on all $v \in V_i^c$ can still be computed correctly. But, the presence of these nodes in N_w does not affect the correctness of the algorithm either. However, as our experimental results in Figure 5 show, we can save about 50% of memory space by not storing such nodes $x \notin V_i$ in N_w. Figure 5 also demonstrates that as more processors are used, memory requirement per processor decreases.

5.3 Counting Triangles

Once each processor P_i has its partition $G_i(V_i, E_i)$, it uses the improved sequential algorithm *NodeIteratorN* presented in Section 3 to count 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 P_i is given in Figure 6. 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

Scansionato con CamScanner

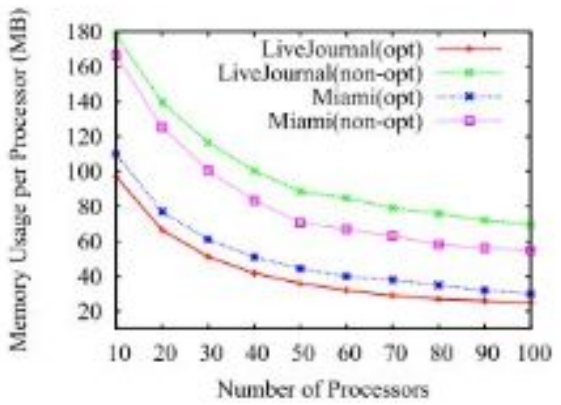


Fig. 5. Memory usage with optimized and non-optimized data storing.

```
for v ∈ V_i do
    sort N_v in ascending order
T ← 0
for v ∈ V_i^c do
    for u ∈ N_v do
        S ← N_v ∩ N_u
        T ← T + |S|
return T
```

Fig. 6. Algorithm executed by processor P_i to count triangles in $G_i(V_i, E_i)$.

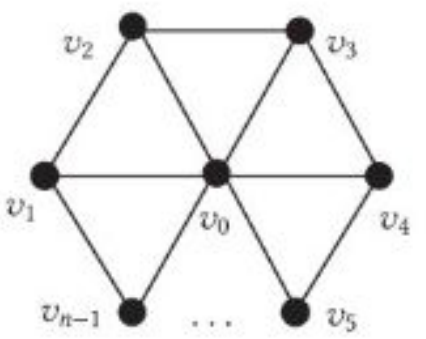


Fig. 7. A network with a skewed degree distribution: $d_{v_0} = n - 1$, $d_{v_{i \neq 0}} = 3$.

correctness of the sequential algorithm *NodeIteratorN* shown in Section 3 ensures that each triangle is counted exactly once.

5.4 Load Balancing

To reduce the runtime of a parallel algorithm, it is desirable that no processor remains idle and all processors complete their executions almost at the same time. In Section 3, we discussed how degree based ordering of the nodes can reduce the runtime of the sequential algorithm, and hence it reduces the runtime of the local computation in each processor P_i. We observe that, interestingly, this ordering also provides load balancing to some extent, both in terms of runtime and space, at no additional cost. Consider the example network shown in Figure 7. With an arbitrary ordering of the nodes, $|N_{v_0}|$ can be as much as $n - 1$, and a single processor that contains v_0 as a core node is responsible for counting all triangles incident on v_0. Then, the runtime of the parallel algorithm can essentially be 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. 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 the processors does not make the load well-balanced in many cases. Figure 8 shows speedup of the parallel algorithm with an equal number of core nodes assigned to each processor. LiveJournal network shows 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 9. Unlike Miami network, LiveJournal network has a very skewed degree distribution. (Note that we used 100 processors for our experiments on load distribution. Although we could use a higher number of processors, using fewer processors helped demonstrate the pattern of imbalance of loads more clearly. In our subsequent experiments on scalability, we use a higher number of processors. In fact, we show that our algorithm scales to a larger number of processors when networks grow larger.)

In the next section, we present several load balancing schemes that improve the performance of our algorithm significantly.

Proposed load balancing schemes. The balanced loads are determined before counting triangles. Thus, our parallel algorithm works in two phases:

(1) *Computing balanced load:* This phase computes partitions V_i^c so that the computational loads are well-balanced.
(2) *Counting triangles:* This phase counts the triangles following the algorithms in Figures 4 and 6.

Scansionato con CamScanner

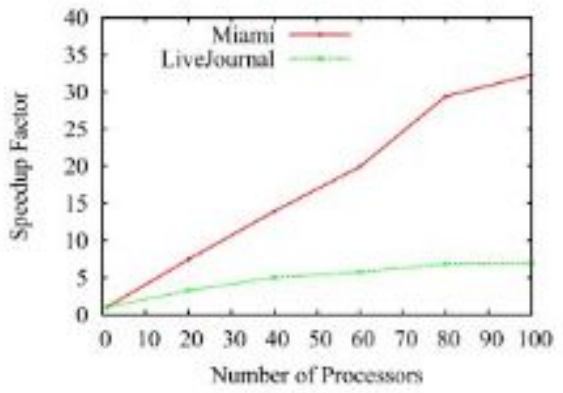


Fig. 8. Speedup with equal number of core nodes in all processors on two networks–Miami and LiveJournal.

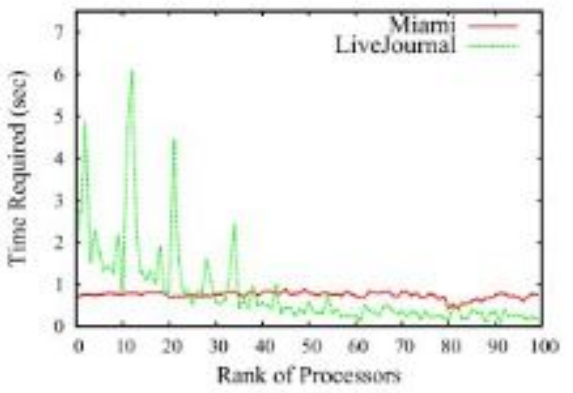


Fig. 9. Runtime of individual processors for equal number of core nodes on Miami and LiveJournal networks.

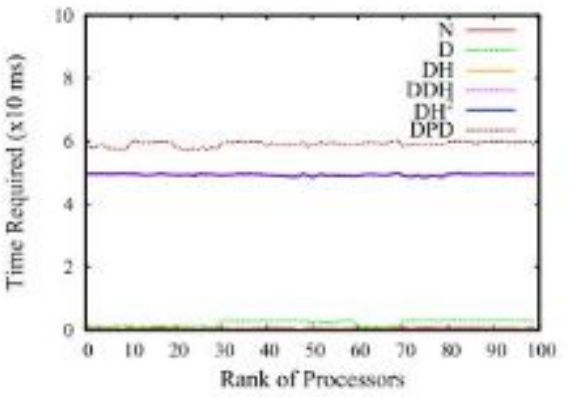


Fig. 10. Load balancing cost for LiveJournal network with different schemes.

Table 3. Cost Functions $f(.)$ for Load Balancing Schemes

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

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 this 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 6). Then, the total cost incurred to P_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 two steps:

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

$$\sum_{v \in V_i^c} f(v) \approx \frac{1}{p} \sum_{v \in V} f(v). \tag{4}$$

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 or even have a negative effect on the performance. Parallelizing the above computation, especially Step 2 (computing partitions), 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 counting triangles takes place. Fortunately, Theorem 3.3 provides a mathematical formulation of 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 shows identifying notations of the individual schemes.

Scansionato con CamScanner

The input graph is given as a sequence of adjacency lists: adjacency list of the first node followed by that of the second node, and so on. The input sequence is considered divided by size (number of bytes) into p chunks. However, it is made sure that adjacency list of a particular node reside in only one processor. Initially, processor P_i stores the ith chunk in its memory. Let C_i be the set of all nodes in the ith chunk. Next, P_i computes $f(v)$ for all nodes $v \in C_i$ as follows.

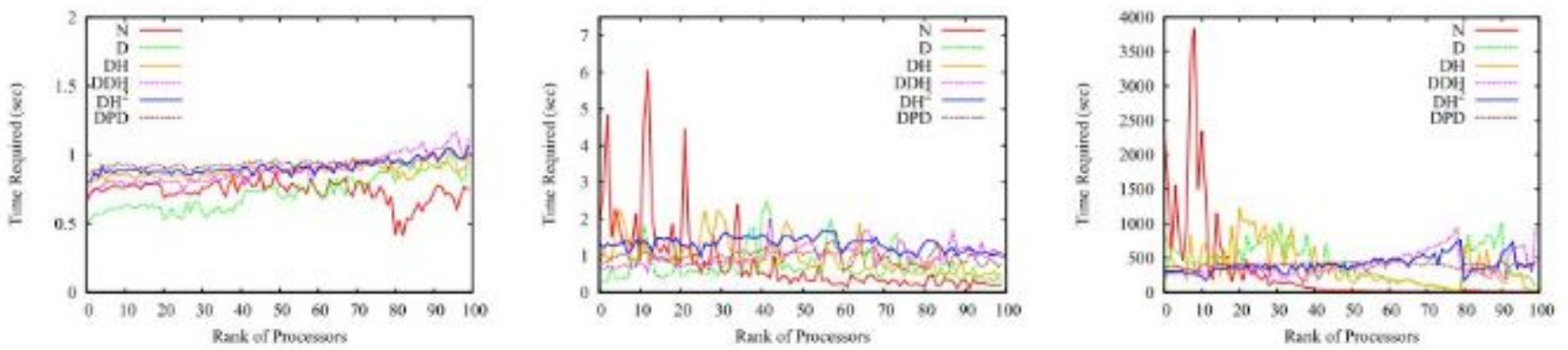
- *Scheme* $\mathbb{N}$: Function $f(v) = 1$ requires no computation. This scheme, essentially, assigns an equal number of core nodes to each processor.
- *Scheme* $\mathbb{D}$: Function $f(v) = d_v$ requires no computation. This scheme, essentially, assigns an equal number of edges to each processor.
- *Scheme* $\mathbb{DH}$: Computing function $f(v) = \hat{d}_v$ requires degrees of all $u \in \mathcal{N}_v$. Let $u \in C_j$. Then, P_i sends a request message to P_j, and P_j replies with a message containing d_u.
- *Scheme* $\mathbb{DDH}$: For $f(v) = d_v \hat{d}_v$, $\hat{d}_v$ is computed as above.
- *Scheme* $\mathbb{DH}^2$: For $f(v) = \hat{d}_v^{\,2}$, $\hat{d}_v$ is computed as above.
- *Scheme* $\mathbb{DPD}$: Function $f(v) = \sum_{u \in \mathcal{N}_v} (\hat{d}_v + \hat{d}_u)$ is computed as follows.
  (i) Each P_i computes $\hat{d}_v$, $v \in C_i$, as discussed above.
  (ii) Then P_i finds $\hat{d}_u$ for all $u \in \mathcal{N}_v$: Let $u \in C_j$. P_i sends a request message to P_j, and P_j replies with a message containing $\hat{d}_u$.
  (iii) Now, $f(v) = \sum_{u \in \mathcal{N}_v} (\hat{d}_v + \hat{d}_u)$ is computed using $\hat{d}_v$ and $\hat{d}_u$ obtained in (i) and (ii).

Computing partitions: Given that each processor P_i knows $f(v)$ for all $v \in C_i$, our goal is to partition V into p disjoint subsets V_i^c such that $\sum_{v \in V_i^c} f(v) \approx \frac{1}{p} \sum_{v \in V} f(v)$.

We first compute cumulative sum $F(t) = \sum_{v=0}^{t} f(v)$ in parallel by using a parallel prefix sum algorithm [6]. Processor P_i computes and stores $F(t)$ for nodes $t \in C_i$. This computation takes $O\left(\frac{n}{p} + \log p\right)$ time. Notice that P_{p-1} computes $F(n-1) = \sum_{v=0}^{n-1} f(v)$, cost for counting all triangles in the graph. P_{p-1} then computes $\alpha = \frac{1}{p} \sum_{v \in V} f(v) = \frac{1}{p} F(n-1)$ and broadcast α to all other processors. Now, let $V_i^c = \{x_i, x_i + 1 \ldots, x_{(i+1)} - 1\}$ for some node $x_i \in V$. We call x_i the *start* or *boundary* node of partition i. Node x_j is the jth boundary node if and only if $F(x_j - 1) < j\alpha \leq F(x_j)$ or equivalently, $x_j = argmin_{v \in V} (F(v) \geq j\alpha)$. A chunk C_i may contain 0, 1, or multiple boundary nodes in it. Each P_i finds the boundary nodes x_j in its chunk: we use the algorithm presented in [4] to compute boundary nodes of partitions, which takes $O(n/p + p)$ time in the worst case. At the end of this execution, each processor P_i knows boundary nodes x_i and $x_{(i+1)}$. Now P_i can construct V_i^c and compute its partition $G_i(V_i, E_i)$ as described in Section 5.2.

Since scheme $\mathbb{DPD}$ requires two levels of communication for computing $f(.)$, it has the largest load balancing cost among all 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 10 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. However, since scheme $\mathbb{DPD}$ gives the best estimation of the counting cost, it provides better load balancing. Figure 11 demonstrates *total computation cost* (load) incurred in individual processors with different schemes on Miami, LiveJournal, and Twitter networks. 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 almost equally among processors. However, 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 capture the computational

Scansionato con CamScanner



(a) Miami network (b) LiveJournal network (c) Twitter network

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

load more precisely and produce a very even load distribution among processors. In fact, for such networks, scheme $\mathbb{DPD}$ provides the best load balancing.

Although there exist several standard graph partitioning algorithms in literature [39, 58] such as Metis, Parmetis, Zoltan, and Patoh, 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. Sequential partitioning algorithms (Metis and Patoh) are not useful for partitioning massive networks. Additionally, we require to estimate weights of nodes (based on triangle counting cost) in parallel in the partitioning procedure, which is not readily available in parallel schemes Zoltan and Parmetis. Hence, we design the above parallel partitioning scheme that considers the actual triangle counting cost incurred at nodes and thus helps in balancing computation loads.

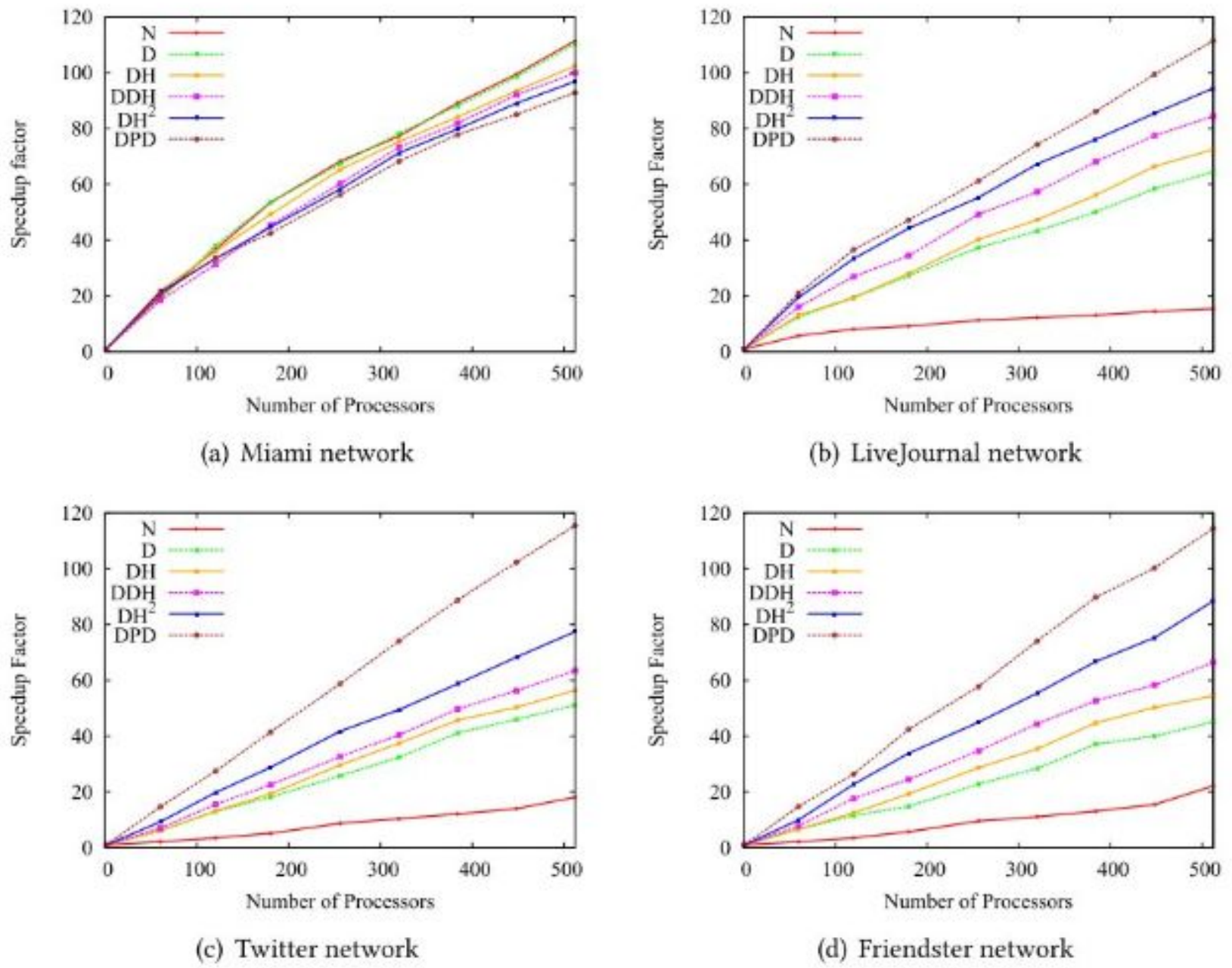
5.5 Performance Analysis

In this section, we present the experimental results evaluating the performance of our algorithm and the load balancing schemes.

5.5.1 Strong Scaling. Strong scaling of a parallel algorithm shows how much speedup a parallel algorithm gains as the number of processors increases. Figure 12 shows strong scaling of our algorithm on LiveJournal, Miami, Twitter, and Friendster networks with different load balancing schemes. The speedup factors of these 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, Twitter, and Friendster networks, speedup factors for different load balancing schemes vary quite significantly. Scheme $\mathbb{DPD}$ achieves better speedup than other schemes. As discussed before, for Miami network, all load balancing schemes distribute loads equally among processors. This produces an almost same speedup on Miami network with all schemes. A lower load balancing cost of schemes $\mathbb{N}$ and $\mathbb{D}$ (Figure 10) yields a little higher speedup. However, real-world graphs such as LiveJournal and Twitter networks, scheme $\mathbb{DPD}$ gives the best load distribution (Figure 11) and thus provides the best speedups. Although $\mathbb{DPD}$ has a higher load balancing cost than others, the benefit gained from $\mathbb{DPD}$ as an even load distribution outweighs this cost. Thus, we recommend for using $\mathbb{DPD}$ on real-world big graphs. Our subsequent results will be based on scheme $\mathbb{DPD}$.

5.5.2 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 x processors, we use network PA$(x/10 \times 1M, 50)$. The weak scaling of our algorithm is shown in Figure 13. Triangle counting cost remains almost constant (blue line). Since the load-balancing step has a communication overhead of $O(p \log p)$, load-balancing cost increases gradually with the increase

Scansionato con CamScanner



(a) Miami network (b) LiveJournal network

(c) Twitter network (d) Friendster network

Fig. 12. Speedup gained from different load balancing schemes for LiveJournal, Miami, Twitter, and Friendster networks.

Table 4. Runtime Performance of Our Fast Parallel Algorithm Using 200 Processors and the Algorithm in [49]

Networks	Runtime		Triangles
	Our algorithm	[49]	
Twitter	9.4 m	423 m	34.8B
web-BerkStan	0.10s	1.70 m	65M
LiveJournal	0.8 s	5.33 m	286M
Miami	0.6 s	–	332M
PA(1B, 20)	15.5 m	–	0.403M

of processors. It causes the total computation time to grow slowly with the addition of processors (red line). Since the growth is very slow and the runtime remains almost constant, the weak scaling of our algorithm is very good.

5.5.3 Comparison with Previous Algorithms. The runtime of our algorithm on several real and artificial networks are shown in Table 4. We also compare our algorithm with another distributed-memory parallel algorithm for counting triangles given in [49]. We select three of the five networks used in [49]. Twitter and LiveJournal are the two largest among the networks used in [49]. We also use web-BerkStan, which has a very skewed degree distribution. No artificial network is used in [49]. For all of these three networks, our algorithm has significantly smaller runtime than that of [49], even though the later uses a larger number of processors (1, 636 processors). The improvement

Scansionato con CamScanner

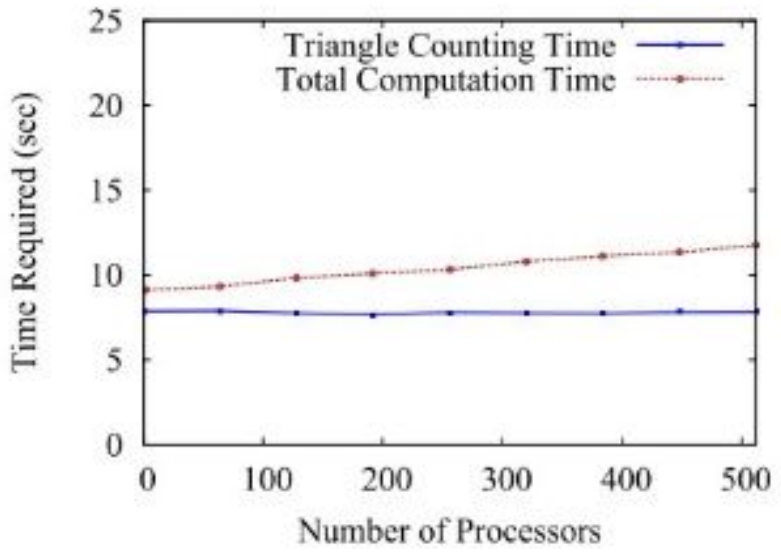


Fig. 13. Weak scaling on PA($p/10 \times 1M, 50$) networks.

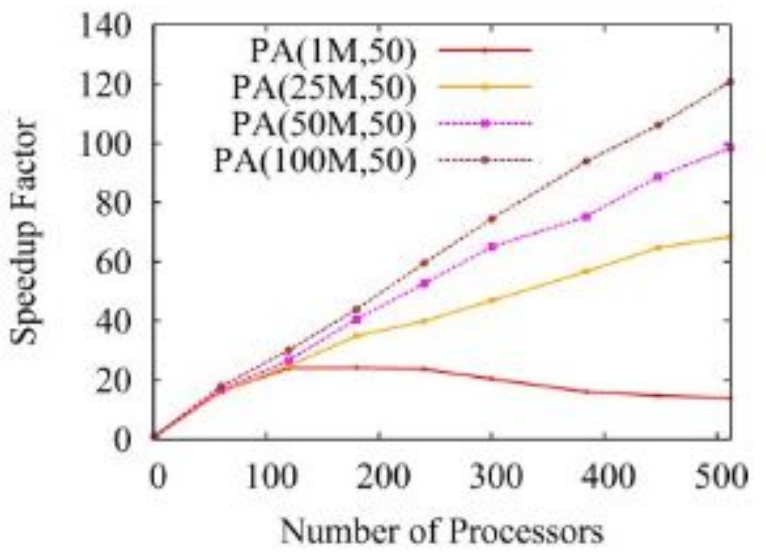


Fig. 14. Improved scalability with increased network size.

over [49] is due to the fact that their algorithm generates a huge volume of intermediate data, which are all possible two-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 two-paths are generated while there are only 2.4B edges in the network. The algorithm in [49] shuffles and regroups these two-paths, which take significantly larger time and also memory.

5.5.4 Scaling with Network Size. The load-balancing cost of our algorithm, as shown in Section 5.4, is $O(m/p + p \log p)$, where p is the number of processors used in the computation. For the algorithm given in Figure 6, 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

$$\begin{aligned} F(p) &= O\left(\frac{m}{p} + p \log p + \max_i \sum_{v \in V_i^c} \sum_{u \in N_v} (\hat{d}_u + \hat{d}_v)\right) \\ &\approx c_1 \frac{m}{p} + c_2 p \log p + c_3 \max_i \sum_{v \in V_i^c} \sum_{u \in N_v} (\hat{d}_u + \hat{d}_v), \end{aligned}$$

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. 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 14. 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 its maximum. To have an estimation of p_{opt}, we replace d and $\hat{d}$ with average degree $\bar{d}$ and $\bar{d}/2$, respectively, 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, $\frac{\mathrm{d}}{\mathrm{d}p}\big(F(p)\big) = 0$, 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 $\bar{d}$.

Assume that a network with the 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 $\bar{d}$ has an

Scansionato con CamScanner

Table 5. Memory Usage of Our Algorithms (Size of the Largest Partition) with Both Overlapping and Non-overlapping Partitioning

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

Number of partitions used is 100.

approximate optimum number of processors:

$$p_{opt} \approx p'_{opt} \frac{\bar{d}}{\bar{d}'} \sqrt{\frac{n}{n'}}. \tag{5}$$

Thus, if we compute p'_{opt} experimentally by trial and error for an available network (let's call it the *base network*), we can estimate p_{opt} for all other networks. The base network might be a small network for which this trial-error should be fairly fast. From the result presented in Figure 14, the network $PA(1M, 50)$ can serve as a base network, and p_{opt} for the network $PA(25M, 50)$ can be estimated as $p_{opt} \approx 600$, which is approximately five times of that of $PA(1M, 50)$ ($p'_{opt} \approx 120$). The relationship is also justified when we vary average degree of the networks.

6 A SPACE-EFFICIENT PARALLEL ALGORITHM WITH NON-OVERLAPPING PARTITIONING

The algorithm presented in Section 5 divides the input graph into a set of p overlapping partitions, where some edges (u, v) might be repeated (overlapped) in multiple partitions. Such overlapping allows the algorithm to count triangles without any communication among processors leading to faster computation. Further, since each processor works on a part of the entire graph, the algorithm can work on large graphs. However, for instances where the graph has a high average degree or a few nodes with high degrees, overlapping partitions can be large. Now, if overlapping of edges among partitions are avoided, we can further improve the space efficiency of the algorithm. In this section, we present a parallel algorithm that divides the input graph into non-overlapping partitions. Each edge resides in a single partition, and the sizes of all partitions sum up to the size of the graph. Non-overlapping partitioning leads to a more space efficient algorithm and thus allows to work on larger graphs. In fact, non-overlapping partitioning offers as much as $\bar{d}$ (average degree of the graph) times space saving over the overlapping partitions. Table 5 shows the space requirement of non-overlapping partitions, which is up to 25 times smaller than that overlapping partitions for the networks we experimented on.

Notice the space requirement of the other distributed-memory parallel algorithms for counting the exact number of triangles in literature: the first MapReduce based algorithm proposed in [49] generates a huge amount of intermediate data, which is significantly larger than the original network (e.g., 125 times larger for Twitter network). The second MapReduce based algorithm proposed in [49], 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 [36] also requires $O(mp)$ memory space. Our space-efficient algorithm requires only a total of $O(m)$ space for storing all p partitions.

Scansionato con CamScanner

6.1 Overview of Our Space-Efficient Parallel Algorithm

This algorithm partitions the input graph $G(V, E)$ into a set of p partitions constructed as follows: set of nodes V is partitioned into p disjoint subsets V_i^c, such that, for $0 \leq j, k \leq p-1$ and $j \neq k$, $V_j^c \cap V_k^c = \emptyset$ and $\bigcup_k V_k^c = V$. Edge set E_i^c, constructed as $E_i^c = \{(u, v) : u \in V_i^c, v \in N_u\}$, constitutes the ith 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 $j \neq k$, $E_j^c \cap E_k^c = \emptyset$ and $\bigcup_k E_k^c = E$. The sum of space required to store all partitions equals to the space required to store the whole graph.

Now, to count triangles incident on $v \in V_i^c$, processor P_i needs N_u for all $u \in N_v$ (Lines 7–10, Figure 2). If $u \in V_i^c$, information of both N_v and N_u is available in the ith partition, and P_i counts triangles incident on (v, u) by computing $N_u \cap N_v$. However, if $u \in V_j^c$, $j \neq i$, 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.

6.2 An Efficient Communication Approach

Processor P_i and P_j require to exchange messages for counting triangles incident on (v, u) where $v \in V_i^c$ and $u \in N_v \cap V_j^c$. A simple way to count such triangles is as follows: P_i requests P_j for N_u. P_j sends N_u to P_i, and P_i counts triangles incident on the edge (v, u) by computing $N_v \cap N_u$. For further reference, we call this approach as *direct approach*. This approach requires exchanging as much as $O(m\bar{d})$ messages ($\bar{d}$ is the average degree of the network), which is substantially larger than the size of the graph.

The above 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 $v_1, v_2, \ldots, v_k \in V_i^c$. Then, P_i sends k separate requests for N_u to P_j while computing triangles incident on $v_1, v_1, \ldots, v_k$. In response to those requests, P_j sends N_u to P_i 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 the same as that of storing an overlapping partition. This diminishes our original goal of a space-efficient algorithm.

Another way of eliminating message redundancy is as follows. When N_u is fetched, P_i completes all computation that requires N_u: P_i finds all k nodes $v \in V_i^c$ such that $u \in N_v$. It then performs all k computations $N_v \cap N_u$ involving N_u and discards N_u. Now, since $u \in N_v \Rightarrow v \notin N_u$, P_i cannot extract all such 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—requiring $O(\sum_{v \in V_i^c} d_v)$ time 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. Below we propose an efficient approach to reduce message exchanges drastically without adding further overhead.

Reduction of messages. To compute $N_v \cap N_u$ for $v \in V_i^c$ and $u \in N_v \cap V_j^c$, P_i requires fetching N_u from partition j. Instead, P_j can perform the same computation if P_i sends N_v to P_j. Specifically, we consider the following approach: P_i sends N_v to P_j instead of fetching N_u. P_j counts triangles incident on edge (u, v) by performing the operation $N_v \cap N_u$. We call this approach as *Surrogate* approach.

Scansionato con CamScanner

```
1: Procedure SurrogateCount(X, i) :
2: T ← 0   //T is the count of triangles
3: for all u ∈ X such that u ∈ V_i^c do
4:     S ← N_u ∩ X
5:     T ← T + |S|
6: return T
```

Fig. 15. The procedure executed by P_i after receiving message $\langle data, X\rangle$ from some P_j.

On a surface, this approach 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^c, by scanning N_v only. For all such nodes u, P_j counts triangles incident on edge (u, v) by performing the operation $N_v \cap N_u$. P_j then discards N_v since it is no longer needed. Note that extracting all u such that $u \in N_v$ and $u \in V_j$ requires $O(d_v)$ time (compare this to $O(\sum_{v \in V_i^c} d_v)$ time of direct approach for the same purpose). In fact, this extraction can be done while computing triangles $N_v \cap N_u$ for first such u. This saves from any additional overhead.

As we noticed, if delegated, P_j can count triangles on multiple edges (u, v) from a single message N_v, where $v \in V_i^c$ and $u \in N_v \cap V_j^c$. Thus, P_i does not require to send N_v to P_j multiple times for each such u. However, to avoid multiple sending, P_i needs to keep track of which processors it has already sent N_v to. This *message tracking* needs to be done carefully, otherwise 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 jth flag to see if it is already sent. This implementation is conceptually simple but cost for resetting flags for each $v \in V_i^c$ sums to a significant cost of $O(|V_i^c|.p)$. Now notice that an overhead of $O(|V_i^c|.p)$ will lead to a runtime of at least $\Omega(n)$ because $\max_i |V_i^c| \geq \frac{n}{p}$. An algorithm with $\Omega(n)$ will not be scalable to a large number of processors since with the increase of p, the runtime $\Omega(n)$ does not decrease.

Now, observe the following simple yet useful property of N_v: *Since V_j^c is a set of consecutive nodes, and all neighbor lists N_v are sorted, all nodes $u \in N_v \cap V_j^c$ reside in N_v in consecutive positions.* This property enables each P_i to track messages by only recording the last processor (say, *LastProc*) it has sent N_v to. When P_i encounters $u \in N_v$ such that $u \in V_j^c$, it checks *LastProc*. If $LastProc \neq P_j$, then P_i sends N_v to P_j and set $LastProc = P_j$. Otherwise, the node u is ignored, meaning it would be redundant to send N_v. Resetting a single variable *LastProc* has an overhead of $O(|V_i^c|)$ as opposed to $O(|V_i^c|.p)$.

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 experimentally in Section 6.7.

6.3 Pseudocode for Counting Triangles

We denote a message by $\langle t, X\rangle$, where $t \in \{data, control\}$ is the type and X is the actual data associated with the message. For a data message ($t = data$), X refers to a neighbor list N_x, whereas for a control ($t = control$), $X = \emptyset$. The pesudocode for counting triangles for an incoming data message $\langle data, X\rangle$ is given in Figure 15.

Once a processor P_i completes the computation on all $v \in V_i^c$, it broadcasts a completion message $\langle control, \emptyset\rangle$. However, it cannot terminate execution until it receives $\langle control, \emptyset\rangle$ from all other processors since other processors might send data messages for surrogate computation. Finally, P_0

Scansionato con CamScanner

```
1: T_i ← 0   //T_i is P_i's count of triangles
2: for each v ∈ V_i^c do
3:    for u ∈ N_v do
4:       if u ∈ V_i^c then
5:          S ← N_v ∩ N_u
6:          T_i ← T_i + |S|
7:       else
8:          Send ⟨data, N_v⟩ to P_j, where u ∈ V_j, if not sent already
9:
10:   for each incoming message ⟨t, X⟩ do
11:      if t = data then
12:         T_i ← T_i + SurrogateCount(X, i) // See Figure 16
13:      else
14:         Increment completion counter
15:
16: Broadcast ⟨control, ∅⟩
17: while completion counter < p-1 do
18:   for each incoming message ⟨t, X⟩ do
19:      if t = data then
20:         T_i ← T_i + SurrogateCount(X, i) // See Figure 16
21:      else
22:         Increment completion counter
23:
24: MpiBarrier
25: Find Sum T ← Σ_i T_i using MpiReduce
```

Fig. 16. An algorithm for counting triangles using surrogate approach. Each processor P_i executes Line 1–22. After that, they are synchronized, and the aggregation is performed (Line 24–25).

sums up counts from all processors using MPI aggregation function. The complete pseudocode of our algorithm using surrogate approach is presented in Figure 16.

6.4 Partitioning and Load Balancing

While constructing partitions i, set of nodes V is partitioned into p disjoint subsets V_i^c of consecutive nodes. Ideally, the set V should be partitioned in such a way that the cost for counting triangles is almost equal for all processors. Similar to our fast parallel algorithm presented in Section 5, we need to compute p disjoint partitions of V such that for each partition V_i^c:

$$\sum_{v \in V_i^c} f(v) \approx \frac{1}{p} \sum_{v \in V} f(v). \tag{6}$$

Several estimations for $f(v)$ were proposed in Section 5 among which $f(v) = \sum_{u \in N_v} (\hat{d}_v + \hat{d}_u)$ was shown experimentally as the best. Since our algorithm employs a different communication scheme for counting triangles, none of those estimations corresponds to the cost of this algorithm. Thus, we derive a new cost function $f(v)$ to estimate the computational cost of our algorithm more precisely.

Deriving An Estimation for Cost Function $f(v)$. We want to find $f(v)$ such that $\sum_{v \in V_i^c} f(v)$ gives a good estimation of the computation cost incurred on processor P_i. We derive $f(v)$ as follows.

Scansionato con CamScanner

Recall that $\mathcal{N}_v = \{u : (u, v) \in E\}$ and $N_v = \{u : (u, v) \in E, v \prec u\}$. Then, it is easy to see that

$$u \in \mathcal{N}_v - N_v \Leftrightarrow v \in N_u. \tag{7}$$

Now, P_i performs two types of computations due to all $v \in V_i^c$ as follows.

(1) *Surrogate or delegated computation:* P_i compute $N_v \cap N_u$ for all $v \in N_u$ and $u \in V_j^c$, $i \neq j$, i.e., $u \in (\mathcal{N}_v - N_v) \cap (V - V_i^c)$. The cost incurred on P_i for such u and v is given by

$$\Theta\left(\sum_{v \in V_i^c} \sum_{u \in (\mathcal{N}_v - N_v) \cap (V - V_i^c)} (\hat{d}_v + \hat{d}_u)\right).$$

(2) *Local computation:* P_i compute $N_v \cap N_u$ for all $u \in N_v \cap V_i^c$. Let E_i^c be the set of edges (u, v) where both u and v are in V_i^c, i.e., $E_i^c = \{(u, v) \in E | u, v \in V_i^c\}$. Now, the cost incurred on P_i for local computations is given by

$$\begin{aligned}\Theta\left(\sum_{v \in V_i^c} \sum_{u \in N_v \cap V_i^c} (\hat{d}_v + \hat{d}_u)\right) &= \Theta\left(\sum_{(u,v) \in E_i^c} (\hat{d}_v + \hat{d}_u)\right) \\ &= \Theta\left(\sum_{v \in V_i^c} \sum_{u \in (\mathcal{N}_v - N_v) \cap V_i^c} (\hat{d}_v + \hat{d}_u)\right).\end{aligned}$$

By adding costs from (1) and (2) above, we get the computation cost:

$$\Theta\left(\sum_{v \in V_i^c} \sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)\right).$$

Now, if we assign $f(v) = \left(\sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)\right)$, the computation cost incurred on P_i becomes $\sum_{v \in V_i^c} f(v)$. Thus, we use the following cost function:

$$f(v) = \left(\sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)\right).$$

Parallel computation of the cost function $f(v)$. In parallel, each processor P_i computes $f(v)$ for all $v \in C_i$. Recall that C_i is the set of all nodes in the ith chunk, as discussed in Section 5.4. Function $f(v) = \left(\sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)\right)$ is computed as follows.

(i) First P_i computes $\hat{d}_v$, $v \in C_i$: computing $\hat{d}_v$ requires d_u for all $u \in \mathcal{N}_v$. Let $u \in C_j$. Then, P_i sends a request message to P_j, and P_j replies with a message containing d_u.
(ii) Then, P_i finds $\hat{d}_u$ for all $u \in \mathcal{N}_v - N_v$: let $u \in C_j$. P_i sends a request message to P_j, and P_j replies with a message containing $\hat{d}_u$.
(iii) Now, $f(v) = \sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)$ is computed using $\hat{d}_v$ and $\hat{d}_u$ obtained in step (i) and (ii).

Computing balanced partitions. Once $f(v)$ is computed for all $v \in V$, we compute V_i^c using the same algorithm we used for overlapping partitioning as described in Section 5.

6.5 Correctness of the Algorithm

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

Scansionato con CamScanner

Table 6. Number of Messages Exchanged in Direct and Surrogate Approaches

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

THEOREM 6.1. *Given a graph $G = (V, E)$, our space efficient parallel algorithm counts every triangle in G exactly once.*

PROOF. Consider a triangle (x_1, x_2, x_3) in G, and without the loss of generality, assume that $x_1 \prec x_2 \prec x_3$. By the constructions of N_x (Line 2–4 in Figure 2), we have $x_2, x_3 \in N_{x_1}$ and $x_3 \in N_{x_2}$. Now, there are two cases:

- *case 1.* $x_1, x_2 \in V_i^c$: Nodes x_1 and x_2 are in the same partition i. Processor P_i executes the loop in Line 2–6 (Figure 16) 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 $x_1, x_2 \notin N_{x_3}$.
- *case 2.* $x_1 \in V_i^c, x_2 \in V_j^c, i \neq j$: Nodes x_1 and x_2 are in two different partitions i and j, respectively. P_i attempts to count the triangle executing the loop in Line 2–6 with $v = x_1$ and $u = x_2$. However, since $x_2 \notin V_i^c$, P_i sends N_{x_1} to P_j (Line 8). P_j counts this triangle while executing the loop in Line 10–12 with $X = N_{x_1}$, and node x_3 appears in $S = N_{x_2} \cap N_{x_1}$ (Line 4 in Figure 15). 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, each triangle in G is counted once and only once. □

6.6 Analysis of the Number of Messages

For $v \in V_i^c$, we call $(v, u) \in E$ a *cut edge* if $u \in V_j^c, j \neq i$. Let ℓ_{vj} is the number of cut edges emanating from node v to all nodes u in partition j with $v \prec u$. Now, in *Surrogate* approach, for all such cut edges (v, u), processor P_i sends N_v to P_j at most once instead of ℓ_{vj} times. This leads to a saving of the number of messages by a factor of ℓ_{vj} for each $v \in V_i^c$. To get a crude estimate of how the number of messages for *direct* and *surrogate* approaches compare, let ℓ be the number of cut edges ℓ_{vj} averaged over all $v \in V_i^c$ and partitions j. Then, the number of messages exchanged in *direct* approach is roughly ℓ larger than *surrogate* approach.

As shown experimentally in Table 6, *direct* approach exchanges messages that is 4 to 12 times larger than that of *surrogate* approach. Thus, *surrogate* approach reduces approximately 70% to 90% of messages leading to faster computations as shown in Table 7 of the following section.

6.7 Experimental Evaluation

We presented the experimental evaluation of our algorithm with overlapping partitioning in Section 5.5. In this section, we present the performance of our parallel algorithm with non-overlapping partitioning and compare it with other related algorithms. We will denote our algorithm with overlapping partitioning as *AOP* and the algorithm with non-overlapping partitioning as *ANOP* for the convenience of discussion.

Scansionato con CamScanner

Table 7. Runtime Performance of Our Algorithms *AOP* and *ANOP*

Networks	Runtime			Part. time	Comm. time	Triangles
	AOP	Direct	Surrogate			
web-BerkStan	0.10 s	0.8 s	0.14 s	0.005 s	0.0345 s	65M
Miami	0.6 s	3.85 s	0.79 s	0.027 s	0.183 s	332M
LiveJournal	0.8 s	5.12 s	1.24 s	0.03 s	0.411 s	286M
Orkut	1.45 s	9.33 s	2.25 s	0.054 s	0.757 s	627.6M
Twitter	9.4 m	35.49 m	12.33 m	23.02 s	2.803 m	34.8B
PA(1B, 20)	15.5 m	78.96 m	20.77 m	45.01 s	4.98 m	0.403M
Friendster	10.81 m	41.03 m	14.18 m	26.45 s	3.16 m	4.17B

We used 200 processors for this experiment. We showed both direct and surrogate approaches for *ANOP*. The reported runtime (in column 2, 3, and 4) is the total triangle counting time including partitioning and communication time. To show how much time is spent on partitioning, the column "Part. time" denotes the partitioning time with $\mathbb{DPD}$ scheme. The column "comm. time" shows the communication time for Surrogate approach of ANOP algorithm.

Table 8. Runtime of a Matrix Multiplication Based Distributed Parallel Algorithm for Triangle Counting Given in [7]

Networks	web-BerkStan	Miami	LiveJournal	Orkut	Twitter	Friendster	PA(1B, 20)
Runtime	3.28 s	7.21 s	10.26 s	18.85 s	–	–	–

We found the algorithm does not scale to a large number of processors and the runtime performance deteriorates with increasing the number of processors. We took the best runtime below while running with a couple of processors to a hundred. Most of the times, they do not scale beyond 16 processors. The algorithm also fail to work on Twitter, Friendster, and PA(1B, 20) networks.

Comparison with previous algorithms. Algorithm *AOP* does not require message passing for counting triangles leading to a very fast algorithm (Table 7). In the contrary, *ANOP* achieves huge space saving over *AOP* (Table 5), although *ANOP* requires message passing for counting triangles. Our proposed communication approach (surrogate) reduces number of messages quite significantly leading to an almost similar runtime efficiency to that of *AOP*. In fact, *ANOP* loses only ~20% runtime efficiency for the gain of a significant space efficiency of up to 25 times, thus allowing to work on larger networks.

Figure 17 demonstrates the runtimes for counting triangles in Twitter network with other related algorithms [36, 38, 49]. Our algorithms *AOP* and *ANOP* take 9.4 minutes and 12.33 minutes, respectively, with 200 processors. The other algorithms, [49], [36], and [38], require more than 90 minutes with a significantly larger number of processors.

We also experimented with the distributed matrix multiplication based algorithm GraphPad presented in [7] on the same set of networks as given in Table 7. The runtime performance of the algorithm is given in Table 8. As evident from both Tables 7 and 8, our algorithm ANOP is significantly faster than GraphPad. Further, our fastest algorithm AOP is more than 10 times faster than GraphPad for all networks. Besides, GraphPad fails to count triangles in the larger three networks. The authors of the GraphPad paper explain and acknowledge that triangle counting is computationally expensive with matrix-based frameworks including GraphPad (last paragraph of Section VI in [7]).

Strong scaling. Figure 18 shows strong scaling (speedup) of our algorithm *ANOP* 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

Scansionato con CamScanner

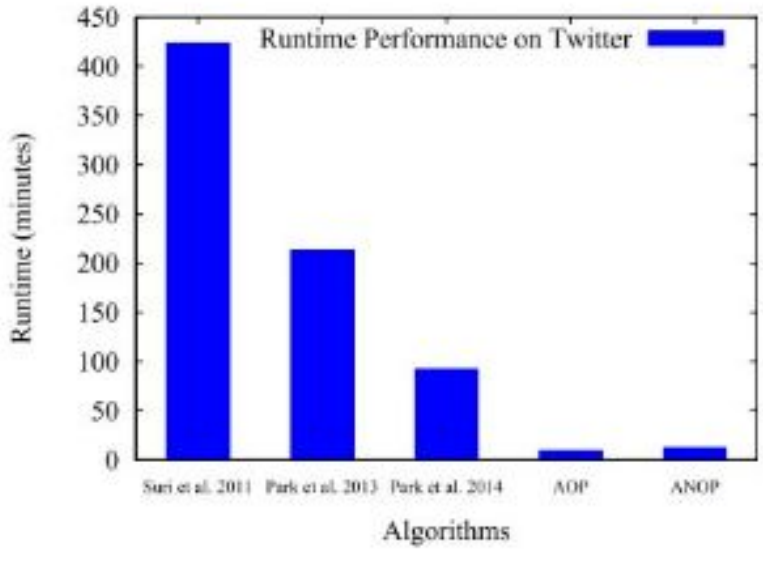


Fig. 17. Runtime reported by various algorithms for counting triangles in Twitter network.

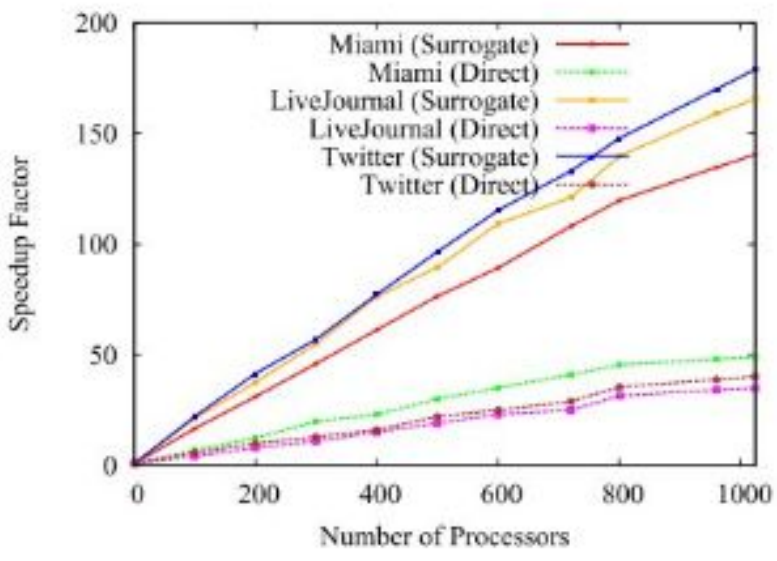


Fig. 18. Speedup factors of our algorithm with both direct and surrogate approaches.

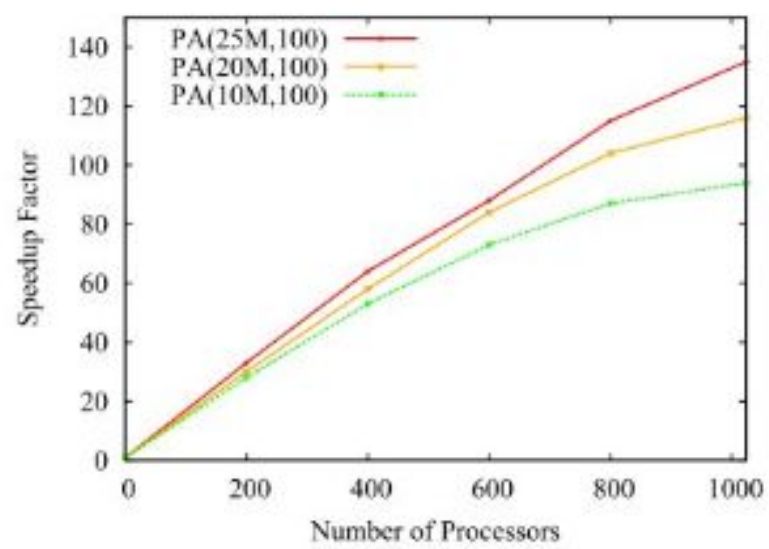


Fig. 19. Improved scalability of our algorithm with increasing network size.

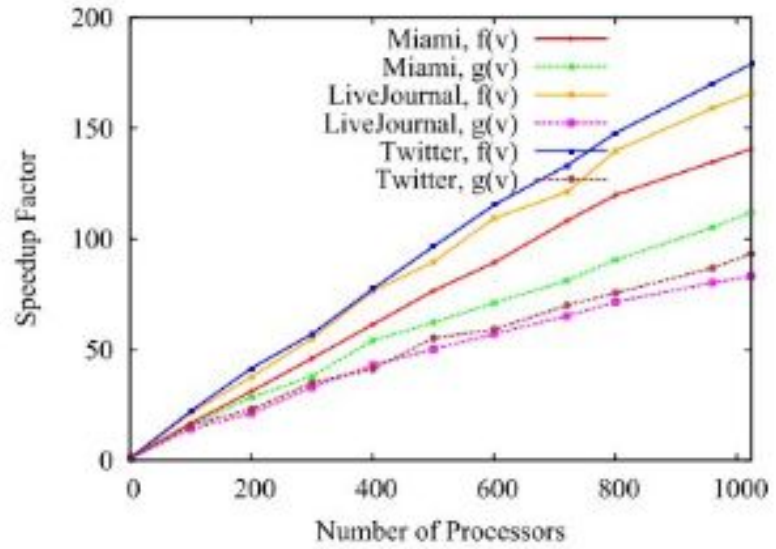


Fig. 20. Comparison of the cost function $f(v)$ estimated for our algorithm with non-overlapping partitioning and the best function $g(v)$ in Section 5.

capability to reduce communication cost drastically. We have experimented our algorithms using up to 1,024 processors (we did not have access to more than 1,024 processors) and these results show that the algorithm scales very well up to this many processors. Further, *ANOP* scales to a higher number of processors when networks grow larger, as shown in Figure 19. 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 estimations for f(v). We show the performance of our algorithm *ANOP* with the new cost function $f(v) = \sum_{u \in N_v - N_v} (\hat{d}_v + \hat{d}_u)$ and the best function $g(v) = \sum_{u \in N_v} (\hat{d}_v + \hat{d}_u)$ computed for *AOP*. As Figure 20 shows, *ANOP* with $f(v)$ provides better speedup than that with $g(v)$. Function $f(v)$ estimates the computational cost more precisely for *ANOP* with surrogate approach, which leads to improved load balancing and better speedup.

Weak scaling. The weak scaling of our algorithm with non-overlapping partitioning is shown in Figure 21. 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.

Scansionato con CamScanner

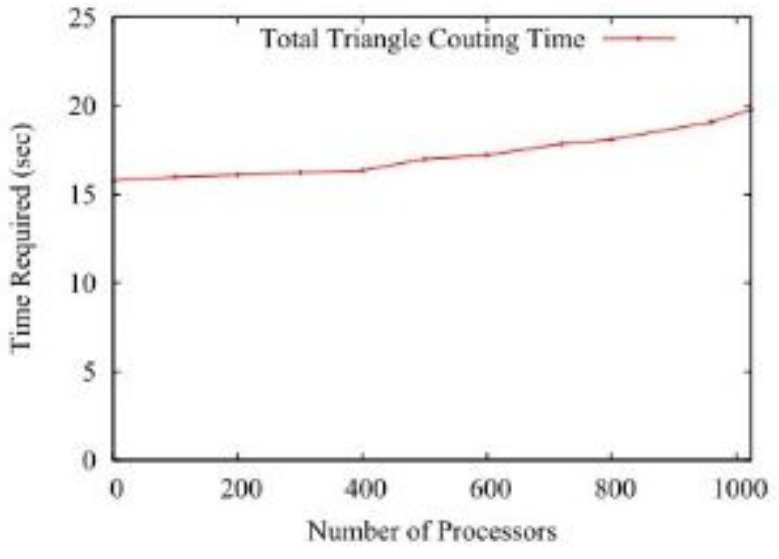


Fig. 21. Weak scaling of our algorithm, experiment performed on PA($t/10 * 1M$, 50) networks, t = number of processors used.

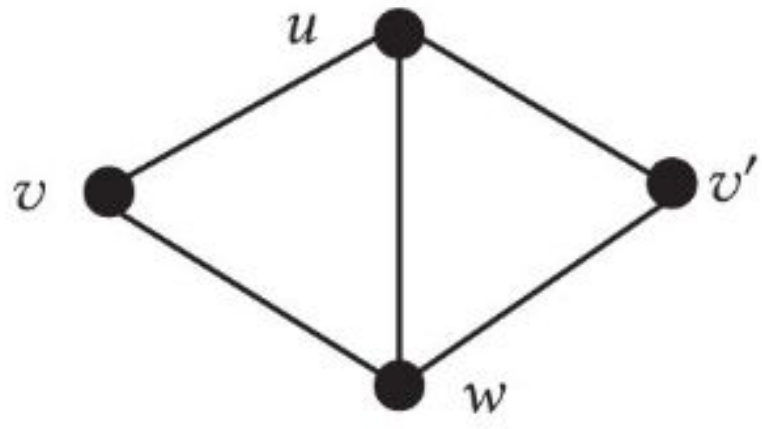


Fig. 22. Two triangles (v, u, w) and (v', u, w) with an overlapping edge (u, v).

7 SPARSIFICATION-BASED PARALLEL APPROXIMATION ALGORITHMS

We discussed our parallel algorithms for counting the exact number of triangles in Sections 5 and 6. In this section, we show how those algorithms can be combined with an edge sparsification technique to design a parallel approximation algorithm.

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 on the sparsified network. Such technique saves both computation time and memory space and provides an approximate result. We integrate a sparsification technique, called DOULION, proposed in [52] with our parallel algorithms. Our adapted version of DOULION provides more accuracy than DOULION when used with overlapping partitioning. The adaptation with non-overlapping partitioning provides the same accuracy as original DOULION.

7.1 Overview of the Sparsification

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 q and removing with probability $1 - q$. 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'. It is easy to see that the expected value of $T(G')$ is $q^3T(G)$, where $T(G)$ is the number of triangles in the original network G. Let the triangles in G be arbitrarily numbered as $1, 2, \ldots, T(G)$, and x_i be an indicator random variable that takes value 1 if triangle i of G survives in G'. A triangle survives if all of its three edges are retained in G'. Then, we have $\Pr\{x_i = 1\} = q^3$ and, by the linearity of expectation,

$$E\left[T(G')\right] = \sum_{i=1}^{T(G)} E[x_i] = \sum_{i=1}^{T(G)} \Pr\{x_i = 1\} = q^3T(G).$$

As shown in [52], the variance of the estimated number of triangles is

$$\text{Var} = \left(\frac{1}{q^3} - 1\right) T(G) + 2k\left(\frac{1}{q} - 1\right), \tag{8}$$

where k is the number of pairs of triangles in G with an overlapping edge (see Figure 22).

7.2 Parallel Sparsification Algorithm

In our parallel algorithm, sparsification is done as follows: each processor P_i independently performs sparsification on its partition $G_i(V_i', E_i')$, where $V_i' = V_i$, $E_i' = E_i$ for AOP and $V_i' = V_i^c$,

Scansionato con CamScanner

```
for v ∈ V'_i do
    for (v, u) ∈ E'_i do
        if v < u then
            store u in N_v with probability q
T_i ← count of triangles on G_i //using alg. in Sec. 5 or 6
Find Sum T' = Σ_i T_i using MpiReduce
T ← (1/q^3) × T'
```

Fig. 23. Counting approximate number of triangles with parallel sparsification algorithm.

Table 9. Accuracy of Our Parallel Sparsification Algorithm and DOULION [52] with $q = 0.1$

Networks	Variance			Avg. error (%)			Max. error (%)		
	AOP	ANOP	DLN	AOP	ANOP	DLN	AOP	ANOP	DLN
web-BerkStan	**1.287**	1.991	2.027	**0.389**	0.391	0.392	**1.024**	1.082	1.082
LiveJournal	**1.770**	1.952	1.958	**1.463**	1.857	1.862	**3.881**	4.774	4.752
web-Google	**1.411**	2.003	1.998	**1.327**	1.564	1.580	**2.455**	3.923	3.942
Miami	**1.675**	2.105	2.112	**1.55**	1.921	1.905	**3.45**	4.88	4.75

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. The best values for each attribute are marked as bold. DLN refers to DOULION.

$E'_i = E^c_i$ for ANOP. While loading the partition G_i into its local memory, P_i retains each edge $(u, v) \in E'_i$ with probability q and discards it with probability $1 - q$ as shown Figure 23.

Now, our parallel sparsification algorithm with overlapping partitioning 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 Figure 22. 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.

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 q^3, we have $E[T'] = q^3 T$. 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 Figure 22) and v and v' are core nodes in two different partitions. Then, clearly, $k' + k'' = k$ and $k' \leq k$. Now following the same steps as in [52], one can show that the variance of our estimation is

$$\mathrm{Var}' = \left(\frac{1}{q^3} - 1\right) T(G) + 2k' \left(\frac{1}{q} - 1\right). \tag{9}$$

Comparing Equations (8) and (9), if $k'' > 0$, we have $k' < k$ and reduced variance leading to improved accuracy. We verify this observation by the experimental results on one realistic synthetic and three real-world networks in Table 9. For all networks, our parallel sparsification algorithm with overlapping partitioning results in smaller variance and errors than that of DOULION. Here, the error percentage (ep) is given by Equation (10).

$$ep = \frac{|T_{ext} - T_{est}| * 100}{T_{ext}}, \tag{10}$$

Scansionato con CamScanner

Table 10. Comparison of Accuracy between Our Parallel Sparsification Algorithms and DOULION on One Realistic Synthetic and Three Real-World Networks with 100 Processors

Networks	Algorithms	$q = 0.1$	$q = 0.2$	$q = 0.3$	$q = 0.4$	$q = 0.5$
web-BerkStan	AOP	**99.9921**	**99.9927**	**99.9932**	**99.9947**	**99.9979**
	ANOP	99.6308	99.7490	99.8392	99.9168	99.9565
	DOULION	99.6309	99.7484	99.8401	99.9171	99.9566
LiveJournal	AOP	**99.9914**	**99.9917**	**99.9924**	**99.9936**	**99.9971**
	ANOP	99.6325	99.7488	99.8412	99.9178	99.9575
	DOULION	99.6310	99.7544	99.8392	99.9121	99.9584
web-Google	AOP	**99.9917**	**99.9923**	**99.9929**	**99.9939**	**99.9975**
	ANOP	99.6299	99.7391	99.8435	99.9168	99.9577
	DOULION	99.6305	99.7398	99.8428	99.9170	99.9574
Miami	AOP	**99.9916**	**99.9919**	**99.9926**	**99.9938**	**99.9974**
	ANOP	99.6285	99.7495	99.8384	99.9168	99.9562
	DOULION	99.6288	99.7494	99.8381	99.9169	99.9563

The best values for each q are marked as bold.

where T_{ext} is the exact (actual) number of triangles and T_{est} is the estimated number of triangles given by the sparsification algorithm. *Average* (or *Max*) error percentage is the avarage (or maximum) value of these error percentages for 25 different runs of the same algorithm.

However, the accuracy does not improve for parallel sparsification with non-overlapping partitioning. Since the partitioning is non-overlapping, the effect of parallel sparsification is the same as that of the sequential sparsification. As a result, our parallel sparsification algorithm with non-overlapping partition has effectively the same accuracy as that of DOULION, as evident in Tables 9 and 10.

Sparsification reduces memory requirement since only a subset of the edges are stored in the main memory. As a result, adaptation of sparsification allows our parallel algorithms to work with even larger networks. With sampling probability q (the probability of retaining an edge), the expected number of edges to be stored in the main memory is $q|E|$. Thus, we can expect that the use of sparsification with our parallel algorithms will allow us to work with a network $1/q$ times larger. Sparsification technique also offers additional speedup due to working on a reduced graph. In [52], it was shown that due to sparsification with parameter q, 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 larger than $1/p$. As an example, with our parallel sparsification with AOP on LiveJournal network, we obtain sparsification speedups of 57.88, 24.36, 11.04, 6.19, and 4.0 for $q = 0.1$ to 0.5, respectively. When an application requires only an approximate count of the total triangles in graph with a reasonable accuracy, such parallel sparsification algorithm will be proven useful. Table 11 shows the comparison of runtimes with parallel sparsification and serial algorithms. Note that the speedups gained in parallel sparsification with AOP and ANOP are due to both sparsification and parallelization.

8 LISTING TRIANGLES IN GRAPHS

Our parallel algorithms for counting triangles in Sections 5 and 6 can easily be extended to list all triangles in graphs. Triangle listing has various applications in the analysis of graphs such as the computation of CCs, transitivity, triangular connectivity, and trusses [18]. Our parallel algorithms counts the exact number of triangles in the graph. To count the number of triangles incident on an

Scansionato con CamScanner

Table 11. Execution Time of Our Parallel Sparsification Algorithms (with Both AOP and ANOP) and Serial Sparsification Algorithm DOULION [52] with $q = 0.1$

Networks	Serial Alg.	DOULION	AOP	ANOP
Web-Google	1,250	36.7647	1.316	1.561
Web-BerkStan	1,400	41.1765	1.282	1.652
LiveJournal	42,000	1,200	30.684	41.379
Miami	32,300	950	26.694	34.75

Our parallel algorithm was run with 100 processors. Times are given in milliseconds. This time includes the initial sparsification step. The serial algorithm in column 2 is NodeIteratorN.

```
1: S ← N_v ∩ N_u
2: for w ∈ S do
3:     Output triangle (u, v, w)
```

Fig. 24. Listing triangles after performing the set intersection operation for counting triangles.

edge (u, v), the algorithms perform a set intersection operation $N_v \cap N_u$. After each intersection operation, all associated triangles can be listed simply by the code shown in Figure 24.

9 COMPUTING CLUSTERING COEFFICIENT OF NODES

Our parallel algorithms can be extended to compute local CC without increasing the cost significantly. In a sequential setting, an algorithm for counting triangles can be directly used for computing CCs of the nodes by simply keeping the counts of triangles for each node individually. However, in a distributed-memory parallel system, combining the counts from all processors for a node poses another level of difficulty. We present an efficient aggregation scheme for combining the counts for a node from different processors.

Parallel computation of clustering coefficients. Recall that CCs of nodes v is computed as follows:

$$C_v = \frac{T_v}{\binom{d_v}{2}} = \frac{2T_v}{d_v(d_v - 1)},$$

where T_v is the number of triangles containing node v.

Our parallel algorithms for counting triangles count each triangle only once. However, all triangles containing a node v might not be computed by a single processor. Consider a triangle (u, v, w) with $u \prec_{\mathcal{D}} v \prec_{\mathcal{D}} w$. Further, assume that $u \in V_i^c$, $v \in V_j^c$, and $w \in V_k^c$, where $i \neq j \neq k$. Now, for our parallel algorithm *AOP*, the triangle (u, v, w) is counted by P_i. Let T_v^i be the number of triangles incident on node v computed by P_i. We also call such counts *local counts* of v in processor P_i. For the triangle (u, v, w), P_i tracks local counts of all of u, v, and w. Thus, the total count of triangles incident on a node v might be distributed among multiple processors. Each processor P_i needs to aggregate local counts of $u \in V_i^c$ from other processors. (For algorithm *ANOP*, the above triangle (u, v, w) is counted by P_j, and a similar argument as above holds.)

To aggregate local counts from other processors, the following approach can be adopted: for each processor, we can store local counts T_v^i in an array of size $\Theta(n)$ and then use MPI *All-Reduce* function for the aggregation. However, for a large network, the required system buffer to perform MPI aggregation on arrays of size $\Theta(n)$ might be prohibitive. Another approach for aggregation might be as follows. Instead of using main memory, local counts can be written to disk files based on some hash functions of nodes. Each processor P_i then aggregates counts for nodes $v \in V_i^c$ from

Scansionato con CamScanner

```
for for each triangle (v, u, w) counted in G_i do
    T_v^i ← T_v^i + 1
    T_u^i ← T_u^i + 1
    T_w^i ← T_w^i + 1
```

Fig. 25. Tracking local counts by processor P_i. Each triangle (v, u, w) is detected by the triangle listing algorithm shown in Figure 24.

```
for v ∈ V_i^c do
    T_v ← T_v^i
for each processor P_j do
    Construct message ⟨Y_i^j, 𝒯_i^j⟩ s.t.:
    Y_i^j ← {v|v ∈ N_u, u ∈ V_i^c} ∩ V_j^c, 𝒯_i^j ← {T_v^i|v ∈ Y_i^j}.
    Send message ⟨Y_i^j, 𝒯_i^j⟩ to P_j
for each processor P_j do
    Receive message ⟨Y_j^i, 𝒯_j^i⟩ from P_j
    T_v ← T_v + T_v^j
```

Fig. 26. Aggregating local counts for $v \in V_i^c$ by P_i.

P disk files. Even though this scheme saves the usage of main memory, performing a large number of disk I/O leads to a large runtime.

Both of the above approach compromises either the runtime or space efficiency. We use the following approach, which is both time and space efficient.

Our approach involves two steps. First, for each triangle counted by P_i, it tracks local counts $T_{\cdot}^{i}$ as shown in Figure 25.

Second, processor P_i aggregates local counts of nodes $v \in V_i^c$ from other processors. Total number of triangles T_v incident on v is given by $T_v = \sum_{j \neq i} T_v^j$. Each processor P_j sends local counts T_v^j of nodes $v \in V_i^c$ encountered in any triangles counted in partition j. P_i receives those counts and aggregates to T_v. We present the pseudocode of this aggregation in Figure 26. Finally, P_i computes $C_v = \frac{2T_v}{d_v(d_v-1)}$ for each $v \in V_i^c$.

Our approach tracks local counts for nodes $v \in V_i^c$ and neighbors of such v, which requires, in practice, significantly smaller than $\Theta(n)$ space. Next, we show the performance of our algorithm.

Performance. We show the strong and weak scaling of our algorithm for computing CCs of nodes in Figures 27 and 28, respectively. The algorithm shows good speedups and scales almost linearly to a large number of processors. Since aggregating local counts introduces additional inter-processor communication, the speedups are a little smaller than that of the triangle counting algorithms. For the same reason, the weak scalability of the algorithm is a little smaller than that of the triangle counting algorithms. However, the increase of runtime with additional processors is still not drastic, and the algorithm shows a good weak scaling.

10 APPLICATIONS FOR COUNTING TRIANGLES

The number of triangles in graphs have many important applications in data mining. Becchetti et al. [14] showed how the number of triangles can be used to detect spamming activity in web graphs. They used a public web spam dataset and compared it with a non-spam dataset: first, they computed the number of triangles for each host and plotted the distribution of triangles and CCs for

Scansionato con CamScanner

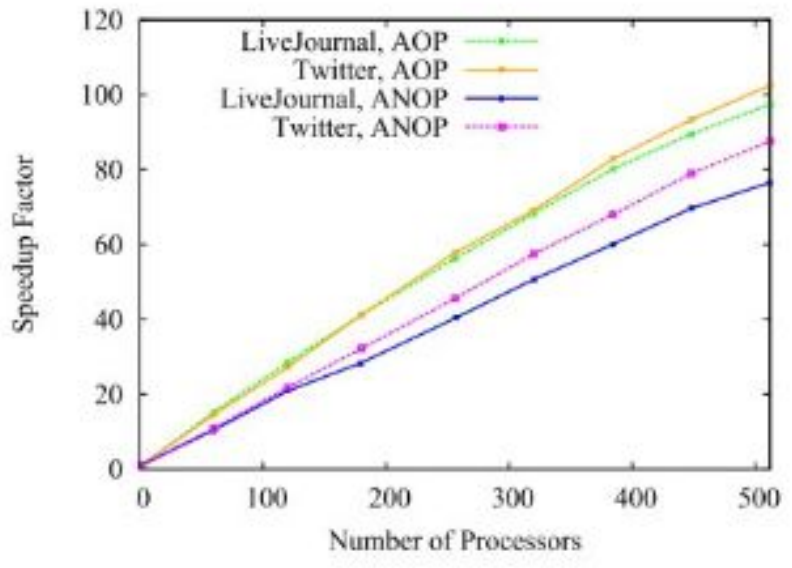


Fig. 27. Strong scaling of clustering coefficient algorithm with both *AOP* and *ANOP* on LiveJournal and Twitter networks.

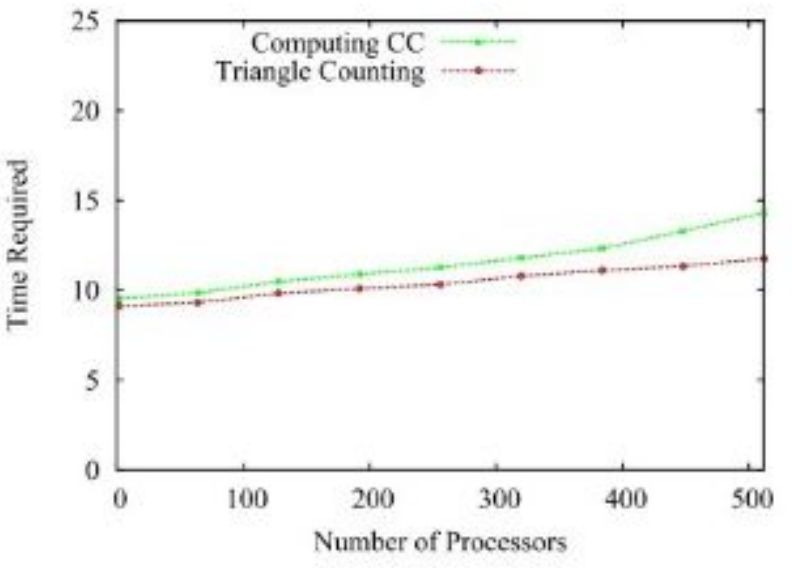


Fig. 28. Weak scaling of the algorithms for computing clustering coefficient (CC) and counting triangles (TC).

Table 12. Comparison of the Number of Triangles ($\triangle$) and Normalized Triangle Count (NTC) in Various Networks

Network	n	$\triangle$	NTC($\triangle/n$)
Gnp($500K$, 20)	500K	1308	0.0026
PA($25M$, 50)	25M	1.3M	0.052
Email-Enron	37K	727044	19.815
web-Google	0.88M	13.39M	15.293
LiveJournal	4.85M	285.7M	58.943
web-BerkStan	0.69M	64.69M	94.408
Miami	2.1M	332M	158.095
Orkut	3.07M	627.6M	204.262
Twitter	42M	34.8B	828.571

We used both artificially generated and real-world networks.

both dataset. Using Kolmogorov–Smirnov test, they concluded the distributions are significantly different for spam and non-spam datasets. Further, the authors also showed how to comment on the role of individual nodes in a social network based on the number of triangles they participate. Eckmann et al. [20] used triangle counting in uncovering the thematic structure of the web. The abundance of triangles also implies community structures in graphs. Nodes forming a subgraph of high triangular density usually belong to the same community. In fact, the number of triangles incident on nodes has been used by several methods in the literature of community detection [41, 47, 57]. The computation of CCs also requires the number of triangles incident on nodes. Social networks usually demonstrate high average CCs. We show how CCs can be computed using our parallel algorithms in Section 9.

In this section, we discuss how the number of triangles can be used to characterize various types of networks. There is a multitude of real-world networks including social contact networks, online social networks, web graphs, and collaboration networks. These networks vary in terms of triangular density and community or social structure in them. As a result, it is possible to characterize real-world networks based on their triangle based statistics. We define the *normalized triangle count* (*NTC*) as the mean number of triangles per node in the network. We compute NTC for a variety of networks and show the comparison in Table 12. Many random graph models such

Scansionato con CamScanner

as Erdős–Réyni and PA models do not generate many triangles, and the resulting NTCs are also very low. Some communication and web graphs (e.g., Email-Enron) generate a descent number of triangles because of the nature of the communication and links among web pages in the host domain. When social or cluster structure exists in the network, we get a larger number of triangles per node, as shown in Table 12 for LiveJournal and web-BerkStan networks. Further, for networks with a more developed social structure and realistic person-to-person interactions, NTCs are very large, as evident for Miami, Orkut, and Twitter networks. Thus, the number of triangles offers good insights about the underlying social and community structures in networks.

11 OTHER PROMINENT RELATED WORK

Azad et al. [10] provide parallel implementation of triangle counting and enumeration based on matrix algebra. Their method scales to networks with around $100M$ edges. Scalability on large graphs and processing cores is adversely affected by communication cost. Their implementation of SpRef (computation of submatrices requested by other processors) is also not efficient and dominates the overall cost.

Satish et al. [43] compared several graph analytics frameworks using a number of graph algorithms (including triangle counting). They reported the performance gaps between native optimized implementation and framework implementations, and proposed suggestion for improvement. (They did not provide the detailed description of their native implementation.)

The work in [19] provides MapReduce based algorithms for several graph problems including the one for enumerating triangles. The work does not provide any experimental evaluation though.

Anderson et al. [7] implement distributed sparse matrix–vector and matrix–matrix multiplication primitives to use in several graph applications including triangle counting. The combination of MPI and OpenMP based implementation of triangle counting is reported to scale to only 64 nodes for graphs with a couple of hundred million edges. We provided an experimental comparison of their work and ours in Section 6.7. Ortmann et al. [35] presented a generic framework of sequential triangle counting algorithms identifying different variants of such algorithms and their running time.

The work in [37] provides MapReduce based algorithms for triangle enumeration by improving a previous algorithm given in [36]. Their algorithms scale well for several large-scale web graphs up to 41 machines.

Elenberg et al. [22] provide a sampling based approximation algorithms for counting triangles on a distributed platform (based on GraphLab PowerGraph framework). Ahmed et al. [3] also provide estimation methods for local graphlets including triangles. They also present a shared-memory based parallel implementation. The article demonstrates scalability on 16 cores for several medium-scale graphs. Lim et al. [32] presented sampling based algorithms (MASCOT) for counting triangles in a streaming setting– a graph is represented as stream of edges. A subsequent work by Stefani et al. [48] provides another sampling based algorithm for graph streams and demonstrates improved accuracies over MASCOT and several other approaches.

Recent MIT GraphChallenge competitions have produced several efficient and scalable algorithms for triangle counting [15, 40, 54, 55]. The work in [55] uses linear algebra-based approach. The article reports performance improvement of several order of magnitude over a Python based reference implementation. However, the article focuses mainly on performance on a single multi-core node. Voegele et al. [54] provide a CPU and GPU based implementation in the Galois and IrGL systems. The implementations are based on a graph-centric abstraction called the operator formulation of algorithms. The article presents optimization specifically for GPU based systems. The work in [15] also provide GPU-based implementation of triangle counting algorithm. For Twitter graph, the reported execution time is 196 seconds. Another work given in [40] provides an

Scansionato con CamScanner

implementation of triangle counting using HavoqGT framework (an asynchronous vertex-centric graph analytics framework). The work achieves 4.7 times improvement over the reference implementation of GraphChallenge. However, the article did not provide an extensive comparison with other related work. The work in [25] presents another GPU-based algorithm using 2-D graph partitioning and binary search based intersection. The algorithm can process a graph with 64 billion edges in 35 minutes.

All of the above work make important contributions to solving the problem of triangle counting and enumeration. However, they differ in focuses (exact or approximate solution), platforms or programming paradigm (shared memory, MapReduce, MPI, etc.), and solution strategies (e.g., matrix-based, set intersection based), among others. Unlike many existing methods mentioned above, our algorithms provide exact solutions for MPI-based distributed memory system. We design parallel communication and load balancing schemes to enhance scalability of the solutions. We showed how these solution can be extended for approximate solutions using any sparsification methods (e.g., DOULION). Our methods complement many existing work by providing alternate approaches and also outperform several related approaches as evident in our experiment sections. We believe our methods will be useful for distributed graph computation domain—where a single machine cannot store or work with the entire graph, or when the input graph is distributed from a previous step of the workflow, or when alternative resources (large shared memory or GPU cores) are not available to the users.

12 CONCLUSION

We presented parallel algorithms for counting triangles and computing CCs in massive networks. These algorithms can work with networks that have billions of nodes and edges. Such capability of our algorithms will enable various types of analysis of massive real-world networks, networks that otherwise do not fit in the main memory of a single computing node. These algorithms show very good scalability with both the number of processors and the problem size and performs well on both real-world and artificial networks. We have been able to count triangles of a massive network with $10B$ edges in less than 16 minutes. 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 algorithms with improved accuracy. This adoption will allow us to deal with even larger networks. We also extend our triangle counting algorithm for listing triangles and computing CCs in massive graphs.

REFERENCES

[1] N. Ahmed, N. Duffield, J. Neville, and R. Kompella. 2014. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. ACM.

[2] N. Ahmed, N. Duffield, T. Willke, and R. Rossi. 2017. On sampling from massive graph streams. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1430–1441.

[3] N. Ahmed, T. Willke, and R. Rossi. 2016. Estimation of local subgraph counts. In *Proceedings of the 2016 IEEE International Conference on Big Data (Big Data'16)*.

[4] M. Alam and M. Khan. 2015. Parallel algorithms for generating random networks with given degree sequences. In *Proceedings of the IFIP Intlernational Conference on Network and Parallel Computing*.

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

[6] S. Aluru. 2012. Teaching parallel computing through parallel prefix. In *Proceedings of ACM/IEEE International Conference on High Performance Computing, Networking Storage and Analysis (SC'12)*.

[7] M. Anderson, N. Sundaram, N. Satish, M. Patwary, T. Willke, and P. Dubey. 2016. GraphPad: Optimized graph primitives for parallel and distributed platforms. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*.

[8] S. Arifuzzaman, M. Khan, and M. Marathe. 2013. PATRIC: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM'13)*.

Scansionato con CamScanner

[9] S. Arifuzzaman, M. Khan, and M. Marathe. 2015. A space-efficient parallel algorithm for counting exact triangles in massive networks. In *Proceedings of the IEEE International Conference on High Performance Computing and Communications (HPCC'15).*

[10] A. Azad, A. Buluç, and J. Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW'15).*

[11] Z. Bar-Yosseff, R. Kumar, and D. Sivakumar. 2002. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms.*

[12] A. Barabasi and R. Albert. 1999. Emergence of scaling in random networks. *Science* 286, 5439 (1999), 509–512.

[13] Christopher L. Barrett, Richard J. Beckman, Maleq Khan, V. S. Anil Kumar, Madhav V. Marathe, Paula E. Stretz, Tridib Dutta, and Bryan Lewis. 2009. Generation and analysis of large synthetic social contact networks. In *Proceedings of the Winter Simulation Conference.* 1003–1014.

[14] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'08).*

[15] M. Bisson and M. Fatica. 2017. Static graph challenge on GPU. In *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC'17).*

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

[17] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. 2000. Graph structure in the Web. *Computer Networks* 33, 1–6 (2000), 309–320.

[18] S. Chu and J. Cheng. 2011. Triangle listing in massive networks and its applications. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD).*

[19] J. Cohen. 2009. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering* 11, 4 (2009), 29–41.

[20] J. Eckmann and E. Moses. 2002. Curvature of co-links uncovers hidden thematic layers in the World Wide Web. *Proceedings of the National Academy of Sciences of the United States of America* 99, 9 (2002), 5825–5829.

[21] T. Eden, A. Levi, D. Ron, and C. Seshadhri. 2015. Approximately counting triangles in sublinear time. In *Proceedings of the 2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS'15).* IEEE Computer Society.

[22] E. Elenberg, K. Shanmugam, M. Borokhovich, and A. Dimakis. 2015. Beyond triangles: A distributed framework for estimating 3-profiles of large graphs. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'15).*

[23] M. Girvan and M. Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America* 99, 12 (June 2002), 7821–7826.

[24] O. Green, P. Yalamanchili, and L. Munguía. 2014. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms.*

[25] Y. Hu, P. Kumar, G. Swope, and H. H. Huang. 2017. TriX: Triangle counting at extreme scale. In *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC'17).* 1–7.

[26] M. Jha, C. Seshadhri, and A. Pinar. 2015. A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox. *ACM Transactions on Knowledge Discovery* 9, 3 (2015), Article 15.

[27] T. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task. 2014. Counting triangles in massive graphs with MapReduce. *SIAM Journal on Scientific Computing* 36, 5 (October 2014), S44–S77.

[28] H. Kwak. 2010. Twitter Data. Retrieved from http://an.kaist.ac.kr/haewoon/release/twitter_social_graph.

[29] H. Kwak, C. Lee, H. Park, and S. Moon. 2010. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web.* ACM, 591–600.

[30] M. Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science* 407, 1–3 (2008), 458–473.

[31] J. Leskovec. 2012. Stanford Network Analysis Project. Retrieved from http://snap.stanford.edu/.

[32] Y. Lim and U. Kang. 2015. MASCOT: Memory-efficient and accurate sampling for counting local triangles in graph streams. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'15).*

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

[34] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[35] M. Ortmann and U. Brandes. 2014. Triangle listing algorithms: Back from the diversion. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments.*

[36] H. Park and C. Chung. 2013. An efficient MapReduce algorithm for counting triangles in a very large graph. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM'13).*

[37] H. Park, S. Myaeng, and U. Kang. 2016. PTE: Enumerating trillion triangles on distributed systems. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16).*

Scansionato con CamScanner

[38] H. Park, F. Silvestri, U. Kang, and R. Pagh. 2014. MapReduce triangle enumeration with guarantees. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM'14)*.

[39] PaToH. 2011. PaToHv3.2. Retrieved from http://bmi.osu.edu/umit/software.html.

[40] R. Pearce. 2017. Triangle counting for scale-free graphs at scale in distributed memory. In *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC'17)*.

[41] A. Prat-Pérez, D. Dominguez-Sal, J. Brunat, and J. Larriba-Pey. 2016. Put three and three together: Triangle-driven community detection. *ACM Transactions on Knowledge Discovery from Data* 10, 3 (2016), 22:1–22:42.

[42] M. Rahman and M. Hasan. 2013. Approximate triangle counting algorithms on multi-cores. In *Proceedings of the IEEE International Conference on Big Data*.

[43] N. Satish, N. Sundaram, M. Patwary, J. Seo, J. Park, M. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*.

[44] T. Schank. 2007. *Algorithmic Aspects of Triangle-Based Network Analysis*. Ph.D. Dissertation. University of Karlsruhe.

[45] T. Schank and D. Wagner. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proceedings of the 4th international conference on Experimental and Efficient Algorithms*.

[46] J. Shun and K. Tangwongsan. 2015. Multicore triangle computations without tuning. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'15)*.

[47] J. Soman and A. Narang. 2011. Fast community detection algorithm with GPUs and multicore architectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*. 568–579.

[48] L. Stefani, A. Epasto, M. Riondato, and E. Upfal. 2017. TRIÈST: Counting local and global triangles in fully dynamic streams with fixed memory size. *ACM Transactions on Knowledge Discovery from Data* 11, 4, Article 43 (2017), 1–50.

[49] S. Suri and S. Vassilvitskii. 2011. Counting triangles and the curse of the last reducer. In *Proceedings of the International World Wide Web Conference (WWW'11)*.

[50] K. Tangwongsan, A. Pavan, and S. Tirthapura. 2013. Parallel triangle counting in massive streaming graphs. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM'13)*.

[51] A. S. Tom, N. Sundaram, N. Ahmed, S. Smith, S. Eyerman, M. Kodiyath, I. Hur, F. Petrini, and G. Karypis. 2017. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC'17)*. 1–7.

[52] C. Tsourakakis, U. Kang, G. Miller, and C. Faloutsos. 2009. DOULION: Counting triangles in massive graphs with a coin. In *Proceedings of the ACM SIGKDD Conf. on Knowledge Discovery and Data Mining (KDD'09)*.

[53] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. 2011. The anatomy of the Facebook social graph. arXiv:1111.4503v1.

[54] C. Voegele, Y. Lu, Pai. S., and K. Pingali. 2017. Parallel triangle counting and k-truss identification using graph-centric methods. In *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC'17)*.

[55] M. Wolf, M. Deveci, J. Berry, S. Hammond, and S. Rajamanickam. 2017. Fast linear algebra-based triangle counting with Kokkos Kernels. In *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC'17)*.

[56] B. Wu, K. Yi, and Z. Li. 2016. Counting triangles in large graphs by random sampling. *IEEE Transactions on Knowledge and Data Engineering* 28, 8 (2016), 2013–2026.

[57] Y. Zhang, J. Wang, Y. Wang, and L. Zhou. 2009. Parallel community detection on large networks with propinquity dynamics. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'09)*. 997–1006.

[58] Zoltan. 2013. Zoltan Graph Partitioning. Retrieved from http://www.cs.sandia.gov/zoltan/.

Received May 2016; revised June 2019; accepted September 2019

Scansionato con CamScanner