A Fast Parallel Algorithm for Counting Triangles in Graphs using Dynamic Load Balancing

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

***Abstract*—Finding the number of triangles in a graph (network) is an important problem in graph analysis. The number of triangles also has important applications in graph mining. Big graphs emerging from numerous application areas pose a significant challenge for the analysis and mining since these graphs consist of millions, or even billions, of nodes and edges. Graphs of such scale necessitate the development of efficient parallel algorithms. Existing distributed memory parallel algorithms for counting exact triangles are either Map-Reduce or message passing interface (MPI) based. Map-Reduce based algorithms generate prohibitively large intermediate data and do not demonstrate reasonably good runtime efficiency. The MPI based algorithms offer fast computation of the number of triangles. However, the partitioning and load balancing schemes these algorithms employ are static in nature– the partitions are precomputed based on some estimations.**

In this paper, we present an efficient MPI-based parallel algorithm for counting triangles in large graph. We consider the case where the main memory of each compute node is large enough to contain the entire graph. We observe that for such a case, computation load can be balanced dynamically and present a dynamic load balancing scheme which improves the performance of the algorithm significantly. Our algorithm demonstrates very good speedups and scales to a large number of processors. The algorithm computes the exact number of triangles in a network with 1 billion edges in 2 minutes with only 100 processors. Our results demonstrate that the algorithm is significantly faster than the related algorithms with static partitioning. In fact, for the real-world networks we experimented on, our algorithm achieves at least 2 times runtime efficiency over the fastest algorithm with static load balancing.

***Keywords*-triangle-counting; parallel algorithms; large graphs; graph mining; social networks.**

I. 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, transitivity, and triangular connectivity [1], [2]. Existence of triangles and the resulting high clustering coefficient in a social network reflect some common theories of social science, e.g., *homophily* where people become friends with those similar to themselves and *triadic closure* where people who have common friends tend to be friends themselves [3]. Further, triangle counting has important applications in graph mining such as detecting spamming activity and assessing content quality [4], uncovering the thematic structure of the web [5], query planning optimization in databases [6], and detecting communities or clusters in social and information networks [7].

Graph is a powerful abstraction for representing underlying relations in large unstructured datasets. Examples include the web graph [8], various social networks [9], biological networks [10], 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 [2], [11]. Such big graphs motivate the need for efficient parallel algorithms.

Counting triangles and related problems such as computing clustering coefficients has a rich history [12]–[20]. 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 [21], [22], sparsification based algorithms [15], [23], external-memory algorithms [2], and parallel algorithms [16], [22], [24], [25]. The streaming and sparsification based algorithms are approximation algorithms. External memory algorithms can be very I/O intensive leading to a large runtime. Efficient parallel algorithms can solve such a problem of a large running time by distributing computing tasks to multiple processors. Over the last couple of years, several parallel algorithms, either shared memory or distributed memory (MapReduce or MPI) based, have been proposed.

In [22], a shared memory parallel algorithm is proposed for counting triangles in a streaming setting. The algorithm provides approximate counts. Over the last couple of years, few more shared memory algorithms have been proposed [18], [19].

Although the algorithms for shared memory paradigm 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 its application to big graphs. A GPU-based parallel algorithm is proposed recently in [17] which achieves an speedup of 32 with 2880 streaming processors.

There exist several algorithms based on MapReduce framework: in [16], two parallel algorithms for exact triangle counting using the MapReduce framework are presented. The first algorithm generates huge volumes of intermediate data, which are all possible 2-paths centered at each node. Shuffling and regrouping these 2-paths require a significantly large amount of time and memory. The second algorithm suffers from redundant counting of triangles. An improvement of the second algorithm is given in a very recent paper [26]. Although this algorithm reduces the redundant counting to some extent, the redundancy is not entirely eliminated. In fact, for P partitions, the algorithm overcounts (P-1 times) triangles whose vertices lie in the same partition. In another recent work [20], Park et al. propose a randomized MapReduce algorithm for triangle enumeration which gives an approximate count.

A MapReduce based parallelization of a wedge-based sampling technique [23] is proposed in [25], which is an approximation algorithm.

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

An MPI based parallel algorithm for counting the *exact* number of triangles in massive networks is proposed in [24]. The algorithm employs an overlapping partitioning scheme and a novel load balancing scheme. This algorithm does not require any inter-processor communication and is demonstrated to be very fast. Another MPI based parallel algorithm is proposed in [27], which employs a non-overlapping partitioning and provide a space efficient algorithm. Both of these algorithms partition the network such that each processor works on a single partition. This allows these algorithm to work on very large graphs. Further, both algorithms offer very fast computation. However, both algorithms are based on static load balancing. Besides, the second algorithm [27] involves exchanging data messages among processors, which reduces its runtime efficiency to some extent.

Now, with the overlapping partitioning scheme in [24], if average degree of input network is large (or the network has few high degree nodes), the largest partition contains almost the entire network. Thus the algorithm requires to store the whole network in the memory of a single machine (which is assigned the largest partition). In such a case, we observe that if the system being used can accommodate the entire network in the main memory of a single machine, we can apply a dynamic load balancing scheme to further improve the runtime efficiency.

In fact, due to the advancement of hardware technology, big-memory machines are becoming increasingly available and affordable. In a recent paper [28], Leskovec et al. reported, a machine with 1TB of main memory and 80 cores costs around $35K. The paper also mentioned, most of the graphs we analyze today comfortably fit in the memory of one such *big-memory* machine. Among the 71 graphs publicly available in the Stanford Large Network Collection [29], 90% of graphs have less than 100M edges. Thus, it is often more useful to provide fast algorithms for commonly used big graphs, rather than to provide very scalable algorithms for extremely large and rare graphs.

Contributions. In this paper, we present an efficient MPI-based parallel algorithm for finding the exact number of triangles in a graph where the memory of each machine is large enough to contain the entire network. We present a dynamic load balancing scheme which improves the performance of the algorithm significantly. Further, we not only assign computational task dynamically among processors, but also vary the task granularity on-the-fly. This dynamic re-adjustment of task granularity offers additional runtime efficiency. Our algorithm achieves very good speedups and scales well to a large number of processors. The algorithm computes the exact number of triangles in a network with 1B edges in only 2 minutes using 100 processors. Our results demonstrate that the algorithm is the fastest among the algorithms for counting exact number of triangles. In fact, the algorithm is more than twice as fast as the previous fastest algorithm.

Note that this paper presents an exact algorithm which can be used to count triangles incident on individual nodes (local triangles). Such *local* counts facilitate computing clustering coefficient of nodes and finding vertex neighborhood and community seeds [30]. On the other hand, approximation algorithms only provide an overall (global) estimate of number of triangles in the graph and might fail to provide local statistics of triangles with reasonable accuracy. To the best of our knowledge, among all exact algorithms, our algorithm offers the best runtime efficiency.

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

II. Preliminaries

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

Basic definitions. The given graph (network) is denoted by $G(V, E)$, where V and E are the sets of vertices and edges, respectively, with $m = |E|$ edges and $n = |V|$

Table I
DATASET USED IN OUR EXPERIMENTS. K, M AND B DENOTE THOUSANDS, MILLIONS AND BILLIONS, RESP.

Network	Nodes	Edges	Source
web-Google	0.88M	5.1M	SNAP [29]
web-BerkStan	0.69M	6.5M	SNAP [29]
Miami	2.1M	50M	[31]
LiveJournal	4.8M	43M	SNAP [29]
Twitter	42M	2.4B	[33]
PA(n, d)	n	$\frac{1}{2}nd$	Pref. Attachment

vertices labeled as $0, 1, 2, \ldots, n-1$. We use the words *node* and *vertex* interchangeably. We assume that the input graph is undirected. If $(u, v) \in E$, we say u and v are neighbors of each other. The set of all neighbors of $v \in V$ is denoted by $\mathcal{N}_v$, i.e., $\mathcal{N}_v = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |\mathcal{N}_v|$.

A triangle is a set of three nodes $u, v, w \in V$ such that there is an edge between each pair of these three nodes, i.e., $(u, v), (v, w), (w, u) \in E$. The number of triangles containing node v (in other words, triangles incident on v) is denoted by T_v. Notice that the number of triangles containing node v is same as the number of edges among the neighbors of v, i.e.,

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

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

Datasets. We use both real world and artificially generated networks for our experiments. A summary of all the networks is provided in Table I. Miami [31] is a synthetic, but realistic, social contact network for Miami city. Twitter, LiveJournal, web-BerkStan, and web-Google are real-world networks. Artificial network PA(n, d) is generated using preferential attachment (PA) model [32] with n nodes and average degree d. Both real-world and PA(n, d) networks have very skewed degree distributions. Networks having such distributions create difficulty in partitioning and balancing loads and thus give us a chance to measure the performance of our algorithms in some of the worst case scenarios. Note that in our experiments we consider edges of the input graph to be undirected– we ignore the original directionality of edges for web-Google, web-BerkStan, and LiveJournal networks.

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

III. A BACKGROUND ON COUNTING TRIANGLES

First, we describe the state-of-the-art sequential algorithm for counting triangles, which our parallel algorithm is based on. A brief discussion of some related parallel algorithms follows.

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

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

A. Efficient Sequential Algorithm

A naïve approach to count triangles in a graph $G(V, E)$ is as follows: check, for all possible triples (u, v, w), $u, v, w \in V$, whether (u, v, w) forms a triangle; i.e., check if $(u, v), (v, w), (u, w) \in E$. There are $\binom{n}{3}$ such triples, and thus this algorithm takes $\Omega(n^3)$ time, which is very expensive. A simple but efficient algorithm is: for each node $v \in V$, find the number of pairs of neighbors that complete a triangle with vertex v. In this method, each triangle (u, v, w) is counted six times – all six permutations of u, v, and w. A total ordering $\prec$ of the nodes (e.g., ordering based on node IDs or any arbitrary ordering) makes sure each triangle is counted exactly once. However, algorithms in [13], [14] incorporate an interesting node ordering based on the degrees of the nodes, with ties broken by node IDs, as defined as follows:

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

These algorithms are further improved in a recent paper [24] by a simple modification. The algorithm [24] defines $N_v \subseteq \mathcal{N}_v$ as the set of neighbors of v having a higher order $\prec$ than v,

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

That is, for an edge (u, v), the algorithm stores u in N_v if $v \prec u$, and consequentially, $u \in N_v \iff v \notin N_u$. Then the triangles containing node v and any $u \in N_v$ can be found by set intersection $N_u \cap N_v$. The above state-of-the-art sequential algorithm is presented in Fig. 1. Our parallel algorithm is based on this sequential algorithm.

B. Parallel Algorithms Counting Exact Triangles

In Section I, we mentioned several MapReduce and MPI based algorithms [16], [24], [26], [27] that count *exact* number of triangles. Here, we briefly discuss these algorithms along with their shortcomings to further clarify the context of our work.

The MapReduce based algorithm proposed in [16] works in two rounds of Map and Reduce phases. In Map phases, the algorithm generates a huge amount of intermediate data which are all possible 2-paths w-v-u centered around each node $v \in V$, such that $u, w \in \mathcal{N}_v$. The algorithm

then check whether such 2-paths are closed by an edge, i.e. if $(w, u) \in E$. Since the number of these 2-paths is very large, even larger than the network size, shuffling and regrouping these data requires a large runtime and enormous memory. As instance, for Twitter network, $300B$ 2-paths are generated whereas the network has only $2.4B$ edges. Even for smaller networks, if there are few nodes with high degrees, say $O(n)$, this algorithm generates $O(n^2)$ 2-paths centered at those nodes, which is quite unmanageable. Many real networks demonstrate power-law degree distributions where some nodes have very large degrees (See d_{max} in Table II).

The MPI based algorithm in [24] divides the input graph into a set of P overlapping partitions as follows. First, V is partitioned into P disjoint subsets V_i^c, such that $\bigcup_{0 \leq k < P} V_k^c = V$. Then, a set V_i is constructed as $V_i = V_i^c \cup \left(\bigcup_{v \in V_i^c} N_v\right)$. Now, set of edges E_i, defined as $E_i = \{(u, v) | u \in V_i, v \in N_u\}$, constitutes the i-th overlapping partition which p_i works on. Note that edges in $E_i^c = \{(u, v) | u \in V_i^c, v \in N_u\}$ constitute the *disjoint* (non-overlapping) portion of the partition i. Rest of the edges $(u, v) \in E_i - E_i^c$ overlaps across multiple partitions.

Now, the overlapping partitions allow the algorithm to count triangles without any communication among processors leading to faster computation. However, if average degree of input network is large (or the network has few high degree nodes), the largest partition contains almost the entire network. Table II shows that real world networks have high degree nodes. In many cases, average degrees of networks are also high.

Another algorithm presented in [27] divides the input networks into non-overlapping partitions. This partitioning provides the best space efficiency among the related algorithms. Space required to store individual partitions add up to the space required to store the whole network. However, such partitioning requires inter-processor communications for counting triangles. Although the paper [27] presents an efficient method to reduce the communication cost drastically making it reasonably a fast algorithm, exchanging messages still reduces its runtime efficiency to some extent. Note that algorithms in both [24], [27] employ static load balancing schemes based on some estimates for the cost of counting triangles. Different estimations (as referred to as *cost functions* in those papers) offer varying degree of performance in load balancing, and none of them are entirely precise. Thus, some processors might experience idle time.

Now consider that each computing machine has enough memory for storing the whole network. For such a case, we observe, unlike the algorithms [24], [27], we can apply a dynamic load balancing scheme to reduce idle time of processors drastically and make the computation even faster. Further, since all processors store the whole network, we do not require to exchange data messages as required in [27].

In the following section, we present an efficient parallel algorithm with dynamic load balancing, which is faster than the algorithm with static partitioning. Our algorithm exchanges only small control messages (request, response, or termination messages). This has a very little communication overhead comparing with [27]. To the best of our knowledge, this algorithm is the fastest among algorithms producing exact count of triangles in big graphs.

Table II
MEMORY REQUIRED FOR STORING NETWORKS ALONG WITH THEIR AVERAGE AND MAXIMUM DEGREE STATISTICS.

Network	Memory (GB)	Avg. d	d_{max}
web-Google	0.127	11.6	6332
Miami	2.7	47.6	425
LiveJournal	2.4	18	20333
Twitter	23.7	57.1	1001159
PA(10M, 100)	18.3	100	25068

IV. A FAST PARALLEL ALGORITHM WITH DYNAMIC LOAD BALANCING

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

A. Overview of the Algorithm

Let P be the number of processors used in our computation. Our algorithm distributes the computation of counting triangles on all nodes $v \in V$ in the network among these processors. We refer the computation assigned to and performed by a processor as a task. For the convenience of future discussion, we present the following definitions related to computing tasks.

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

Definition 2. ***An atomic task:*** *A task $\langle v, 1 \rangle$ referring to counting triangles incident on a single node v is an atomic task. An atomic task cannot be further divided.*

Definition 3. ***Task size:*** *Let, $f : V \to \mathcal{R}$ be a cost function such that $f(v)$ denotes some measure of the cost for counting triangles on node v. We define the size $S(v, t)$ of a task $\langle v, t \rangle$ as follows.*

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

We consider the cost functions $f(v) = 1$ and $f(v) = d_v$ since those are known for all $v \in V$ and have no computational overhead. The function $f(v) = 1$ corresponds to same cost for each node, whereas $f(v) = d_v$ implies that

the cost is proportional to the degree of node v. We recall that for the purpose of static load balancing, the paper [24] estimated several cost functions with varying computational overhead. However, since our algorithm balances load dynamically, using a computationally expensive cost function for computing task granularity is not required at all– this might even lead to poor performance of the algorithm.

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

B. An Efficient Dynamic Load Balancing Scheme

We design a dynamic load balancing scheme with a dedicated processor for coordinating balancing decisions. We distinguish this processor as the *coordinator* and the rest as *workers*. The *coordinator* assigns tasks, receives notifications and re-assigns tasks to idle workers, and *workers* are responsible for actually performing *tasks*. At the beginning, each worker is assigned an initial task. Once any worker i completes its current task, it sends a request to *coordinator* for an additional task. From the available un-assigned tasks, *coordinator* assigns a new task to worker i.

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

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

Initial Assignment. The set of $(P-1)$ initial tasks corresponds to counting triangles on nodes $v \in \{0, 1, \ldots, t' - 1\}$ such that $S(0, t') \approx S(t', n-t')$. Thus we need to find node t' which divides the set of nodes V into two disjoint subsets in such a way that $\sum_{v=0}^{t'-1} f(v) \approx \sum_{v=t'}^{n-1} f(v)$, given $f(v)$ for each $v \in V$. Now if we compute sequentially, it takes $O(n)$ time to perform the above computations. However, we observe that a parallel algorithm for computing balanced partitions of V proposed in [24] can be used to perform the above computation which takes $O(n/P+\log P)$ time. Once t' is determined, the task $\langle 0, t'\rangle$ is divided into $(P-1)$ tasks $\langle v, t\rangle$, one for each worker, in almost equal sizes.

$$S(v, t) = \frac{1}{P-1} \sum_{v \in 0}^{t'-1} f(v). \tag{3}$$

That is, set of nodes $\{0, 1, \ldots, t' - 1\}$ is divided into $(P - 1)$ subsets such that for each subset $\{v, v + 1, \ldots, t - 1\}$, $\sum_{i=0}^{t-1} f(v + i) \approx \frac{1}{P-1} \sum_{v \in 0}^{t'-1} f(v)$. This computation can also be done using the parallel algorithm [24] mentioned above.

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

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

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

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

$$S(v, t) = \frac{1}{P-1} \sum_{v \in V'} f(v). \tag{4}$$

Note that the size $S(v, t)$ of a dynamically assigned task $\langle v, t\rangle$ decreases at every new assignment. By the definition

of atomic task (in definition 2) we have a finite number of tasks. When the coordinator has no more unassigned tasks, i.e., $W = \emptyset$, it sends a special termination message $\langle terminate \rangle$ to the requesting worker. Once the coordinator completes sending termination messages to all workers, it aggregates counts of triangles from all workers, and the algorithm terminates.

Note that while workers are performing the initial assignment, the coordinator proceeds to compute task granularity for subsequent assignments and fills a task queue. Thus when any worker requests further tasks, the coordinator can readily respond. Further, responding and receiving task requests have low communication overhead. Thus, the coordinator does not become a bottleneck in this algorithm

C. Counting Triangles

Once a processor i has an assigned task $\langle v, t \rangle$, it uses the algorithm presented in Fig. 2 to count the triangles incident on nodes $v \in \{v, v+1, \ldots, v+t-1\}$.

```
1: Procedure CountTriangles(v, t) :
2: T ← 0   //T is the count of triangles
3: for v ∈ {v, v+1, ..., v+t−1} do
4:     for u ∈ N_v do
5:         S ← N_v ∩ N_u
6:         T ← T + |S|
7: return T
```

Figure 2. A procedure executed by processor i to count triangles corresponding to the task $\langle v, t \rangle$.

The complete pseudocode of our algorithm for counting triangles with an efficient dynamic load balancing scheme is presented in Fig. 3.

D. Correctness of the Algorithm

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

E. Performance

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

Strong Scaling. Strong scaling of a parallel algorithm shows how much speedup a parallel algorithm gains as the number of processors increases. We present the strong scaling of our algorithm on Miami, LiveJournal, and web-BerkStan networks with both cost functions $f(v) = 1$ and

```
1: All processors initially do the following:
2: Determine initial tasks (see discussion of Eqn. 3)
3:
4: The coordinator does the following:
5: W ← ∅
6: for all remaining tasks ⟨v, t⟩ do
7:     EnQueue (W, ⟨v, t⟩ )
8: while W is not ∅ do
9:     Receive task requests ⟨i⟩
10:    ⟨v, t⟩ ← DeQueue (W)
11:    Send message ⟨v, t⟩ to worker i
12: Send ⟨terminate⟩ to proc. i for requests ⟨i⟩
13:
14: Each worker i does the following:
15: T_i ← 0
16: T_i ← T_i + CountTriangles(v, t)   //for initial task
17: while worker i is idle do
18:     Send message ⟨i⟩ to coordinator
19:     Receive message M from coordinator
20:     if M is ⟨terminate⟩ then
21:         Stop execution
22:     else if M is a task ⟨v, t⟩ then
23:         T_i ← T_i + CountTriangles(v, t)
24:
25: MpiBarrier
26: Find Sum T ← Σ_i T_i using MpiReduce
27: return T
```

Figure 3. An algorithm for counting triangles with dynamic load balancing.

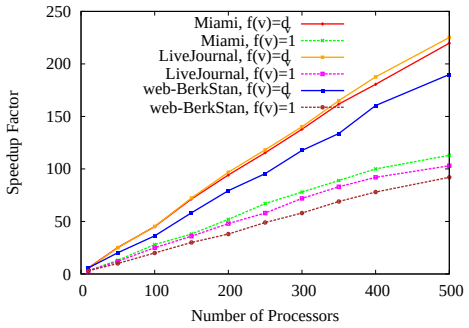

Figure 4. Speedup factors of our algorithm on Miami, LiveJournal and web-BerkStan networks with both $f(v) = 1$ and $f(v) = d_v$ cost functions.

$f(v) = d_v$ in Fig. 4. Our algorithm demonstrates very good speedups and scales almost linearly to a large number of processors. Further, speedup factors are significantly higher with the function $f(v) = d_v$ than with $f(v) = 1$. The function $f(v) = 1$ refers to equal cost of counting triangles for all nodes whereas the function $f(v) = d_v$ relates the cost to the degree of v. Distributing tasks based on the sum of degrees of nodes (Eqn. 3 and 4) reduces the effect of skewness of degrees and makes tasks more balanced leading to higher speedups. Our subsequent experiments will be based on cost function $f(v) = d_v$.

We also observe that the larger networks Miami and LiveJournal achieve higher speedups than web-BerkStan. This is, in fact, a desirable advantage when we want to process big graphs. For small networks, the communication overhead in load balancing becomes relatively significant

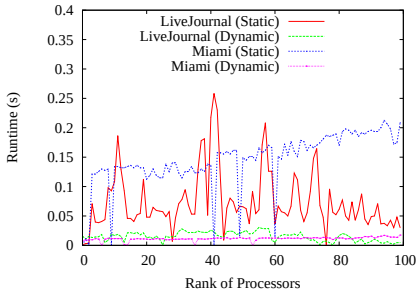

Figure 5. Runtime required by processors (rank-wise) with both static tasks and dynamic adjustment of task granularity.

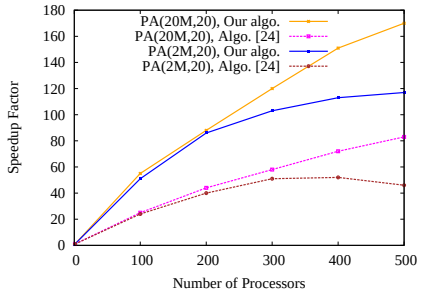

Figure 6. Improved scalability of our algorithm with increasing network size. Further, our algorithm achieves higher speedups than [24].

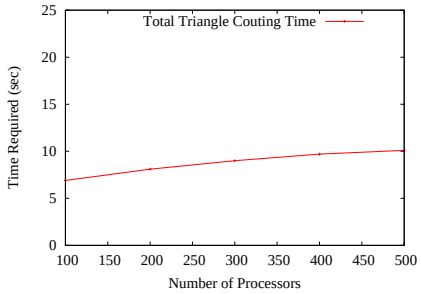

Figure 7. Weak scaling of our algorithm. We perform this experiment on PA($t/10 * 1M, 50$) networks, t = number of processors used.

affecting the speedups to some extent.

Comparison with Previous Algorithms. We compare the runtime of our parallel algorithm with the algorithms in [24] and [27] on a number of real and artificial networks. Note that both algorithms in [24] and [27] are demonstrated to be faster than the MapReduce based algorithms discussed in section I and III (Fig. 4 in [27]). We compare the runtime of our algorithm with dynamic load balancing with these two state-of-the-art fast parallel algorithms. As shown in Table III, our algorithm is more than $2\times$ faster than [24] and about $3\times$ than [27] for all these networks. The algorithm in [24] and [27] are based on static partitioning whereas our algorithm employs a dynamic load balancing scheme to reduce idle time of processors leading to improved performance. We also present a comparison of speedup factors for our algorithm and the algorithms in [24] and [27] on Miami and LiveJournal networks. Our algorithm achieves significantly higher speedups than the others.

Table III
RUNTIME PERFORMANCE OF OUR ALGORITHM AND ALGORITHM [24].

Networks	Runtime			Triangles
	[27]	[24]	Our algo.	
web-BerkStan	0.14	0.10s	0.041s	65M
LiveJournal	1.24	0.8s	0.384s	286M
Miami	0.79	0.6s	0.301s	332M

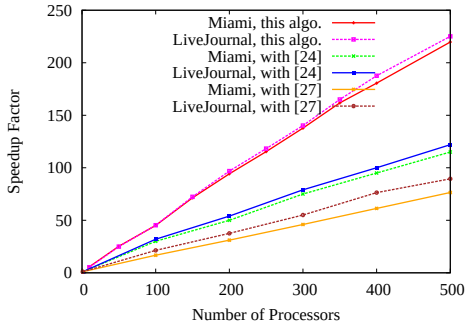

Figure 8. Comparison of speedup factors of our algorithm with [24] and [27] on Miami and LiveJournal networks.

We also notice the reported performance of several shared memory algorithms. The parallel approximation algorithm in [22] demonstrates a speedup of $\approx$ 11 with 12 cores. However, it is not shown how the algorithm will scale for a larger number of cores (or processors). As we demonstrated, our algorithm scales almost linearly to a large number of processors. Another shared memory based parallel approximation algorithm is proposed in [18]. The paper reports speedups using only 32 cores. Further, these speedups are due to both approximation and parallel threads. For example, with a sample factor $p = 0.01$, the paper reports a speedup of 837.74 for Wiki-1 graph with 32 threads, where the approximation contributes a factor of 33.54 in the speedup. The results for other networks demonstrate a parallelization speedup between 1.44 and 24 with 32 threads, which is not impressive for many networks. Further, results for a larger number of cores are not shown in the paper. Similarly, the shared memory algorithm in [19] is reported to scale to 64 cores and achieves speedups ranging from 17 to 50 .

Effect of Dynamic Adjustment of Task Granularity. We show how the granularity of tasks affects idle time of worker processors for Miami and LiveJournal networks. As Fig. 5 shows, with tasks of static size, the distribution of runtime among processors are very uneven leading to large idle times of some processors. However, dynamic adjustment of task granularity provides an almost even distribution of runtime leading to very short idle times. This allows balanced computing loads among processors and consequently improves the runtime performance of the algorithm. Note that we used 100 processors for this experiment. Although we could use a higher number of processors, using fewer processors helped demonstrate the differences in idle times for static and dynamic adjustment of task granularity more clearly. In our next experiment, we show that our algorithm scales to higher number of processors when networks grow larger.

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

is very comparable to that of [24]. To our advantage, our algorithm achieves significantly higher speedup factors than [24].

Weak Scaling. Weak scaling of a parallel algorithm shows the ability of the algorithm to maintain constant computation time when with the increase of the number of processors, the problem size also grows proportionally. The weak scaling of our algorithm is shown in Fig. 7. With the addition of processors, communication overhead increases since idle workers exchange messages with the coordinator for new tasks. However, since the overhead for requesting and assigning tasks is very small, the increase of runtime with additional processors is rather slow (not drastic). Thus, our algorithm demonstrates a reasonably good weak scaling.

V. Conclusion

We present a fast parallel algorithms for counting triangles in large graphs. When the main memory of each computing machine is large enough to store the whole graph, our parallel algorithm with dynamic load balancing can be used for faster analysis of the graph. We believe that for emerging big graphs, this algorithm will be proven very useful.

Acknowledgment

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

References

[1] R. Milo, S. Shen-Orr *et al.*, "Network motifs: simple building blocks of complex networks." *Science*, vol. 298, no. 5594, pp. 824–827, October 2002.

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

[3] M. McPherson, L. Smith-Lovin, and J. Cook, "Birds of a feather: Homophily in social networks," *Annual Rev. of Soc.*, vol. 27, no. 1, pp. 415–444, 2001.

[4] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proc. of KDD*, 2008.

[5] J. Eckmann and E. Moses, "Curvature of co-links uncovers hidden thematic layers in the world wide web," *Proc. Natl. Acad. of Sci. USA*, vol. 99, no. 9, pp. 5825–5829, 2002.

[6] Z. Bar-Yosseff, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proc. of SODA*, 2002.

[7] A. Prat-Pérez *et al.*, "Shaping communities out of triangles," in *CIKM*, 2012.

[8] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Computer Networks*, vol. 33, no. 16, pp. 309 – 320, 2000.

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

[10] M. Girvan and M. Newman, "Community structure in social and biological networks," *Proc. Natl. Acad. of Sci. USA*, vol. 99, no. 12, pp. 7821–7826, Jun. 2002.

[11] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The anatomy of the facebook social graph," *CoRR*, vol. abs/1111.4503, 2011.

[12] N. Alon, R. Yuster, and U. Zwick, "Finding and counting given length cycles," *Algorithmica*, vol. 17, pp. 209–223, 1997.

[13] T. Schank, "Algorithmic aspects of triangle-based network analysis," Ph.D. dissertation, University of Karlsruhe, 2007.

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

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

[16] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proc. of WWW*, 2011.

[17] O. Green *et al.*, "Fast triangle counting on the gpu," in *Proc. of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3, 2014.

[18] M. Rahman and M. A. Hasan, "Approximate triangle counting algorithms on multi-cores," in *Proceedings of the 2013 IEEE International Conference on Big Data*, 2013.

[19] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *ICDE*, 2015.

[20] H.-M. Park *et al.*, "Mapreduce triangle enumeration with guarantees," in *CIKM*, 2014.

[21] M. Jha *et al.*, "A space efficient streaming algorithm for triangle counting using the birthday paradox," in *Proc. of KDD*, 2013.

[22] K. Tangwongsan, A. Pavan, and S. Tirthapura, "Parallel triangle counting in massive streaming graphs," in *Proc. of CIKM*, 2013.

[23] C. Seshadhri *et al.*, "Triadic measures on graphs: the power of wedge sampling," in *Proc. of SDM*, 2013.

[24] S. Arifuzzaman, M. Khan, and M. Marathe, "PATRIC: A parallel algorithm for counting triangles in massive networks," in *Proc. of CIKM*, 2013.

[25] T. G. Kolda *et al.*, "Counting triangles in massive graphs with mapreduce," *CoRR*, vol. abs/1301.5887, 2013.

[26] H.-M. Park and C.-W. Chung, "An efficient mapreduce algorithm for counting triangles in a very large graph," in *Proc. of CIKM*, 2013.

[27] S. Arifuzzaman *et al.*, "A space-efficient parallel algorithm for counting exact triangles in massive networks," in *IEEE HPCC*, 2015.

[28] Y. Perez *et al.*, "Ringo: Interactive graph analytics on big-memory machines," in *SIGMOD*, 2015.

[29] SNAP. (2012) Stanford network analysis project. http://snap.stanford.edu/. [Online]. Available: http://snap.stanford.edu/

[30] D. Gleich and C. Seshadri, "Vertex neighborhoods, low conductance cuts, and good seeds for local community methods," in *KDD*, 2012.

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

[32] A. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, pp. 509–512, 1999.

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