Parallel Algorithms for Counting Triangles and Computing Clustering Coefficients

S M Arifuzzaman*†, Maleq Khan † and Madhav Marathe*†
*Dept. of Computer Science, Virginia Tech, Blacksburg, VA
†Virginia Bioinformatics Institute, Blacksburg, VA

I. Introduction

We present efficient MPI-based distributed memory parallel algorithms for counting triangles and computing clustering coefficients of the nodes in massive networks that do not fit in the main memory of a single computing node.

Given an undirected network $G(V, E)$ with V and E being the sets of nodes and edges, respectively, and $m = |E|$, $n = |V|$, a triangle is defined as a set of three nodes $u, v, w \in V$ such that there is an edge between each pair of these three nodes. Let the set of all neighbors of $v \in V$ is denoted by $N(v)$ and the degree of v is $d_v = |N(v)|$. The number of triangles incident on v is given by,

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

The clustering coefficient (CC) of a node $v \in V$, denoted by C_v is the ratio of the number of edges between neighbors of v to the number of all possible edges between neighbors of v. Then, we have,

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

Thus, CC of a node can be computed by simply counting the number of triangles incident on the node.

Triangles and clustering coefficients play significant roles in the analysis of complex networks. Existence of triangles and the resulting high clustering coefficient reveals important characteristics of social, biological, web and different other networks [1], [2], [3].

The problem of computing the clustering coefficients of the nodes, and almost equivalently counting triangles, in a graph has a rich history [4]–[6]. Much of the earlier work focused on improving running time rather than paying attention to memory issues, and these algorithms are mainly based on matrix multiplication and adjacency matrix representation of the network [4].

Matrix based algorithms [4] are not useful in the analysis of social networks as adjacency matrix representation of network requires $O(n^2)$ memory space. In the last decade the focus has been shifted to algorithms that use adjacency list representation [5], [6], which takes $O(m)$ memory. Although a fairly large volume of work has been done on this problem, much less attention was given, until recently, to the problems associated with massive networks that do not fit in the main memory. Several techniques can be employed to deal with such massive graphs: streaming algorithms ([2], [3]), sparsification based algorithms [7], external-memory algorithms [8], and distributed memory parallel algorithms [9]. The streaming and sparsification based techniques provide approximation algorithms whereas external-memory and parallel algorithms can be used to find exact solutions

To the best of our knowledge, very few works ([8]) have addressed the problems associated with massive networks that do not fit in the main memory and provide an exact solution. A very recent paper [9] presents a parallel algorithm for exact triangle counts using MapReduce framework. Our parallel algorithm improves the performance, both in time and space, over [9] significantly.

Our algorithms scales well to networks with billions of nodes. It can compute the exact number of triangles and clustering coefficient for a network with two billion nodes and 50 billion edges in 10 minutes. We also adopt an edge sparsification technique to approximate the number of triangles with very high accuracy. Moreover, we propose a simple modification of a state-of-the-art sequential algorithm for counting triangles. This modification improves both the running time and space requirement of the algorithm. We use this modified sequential algorithm as a basis for our parallel algorithm.

II. Sequential Algorithms to Count Triangles

A naive approach to count triangles in a graph $G(V, E)$ is to check, for all possible triples (u, v, w), $u, v, w \in V$, whether (u, v, w) forms a triangle, i.e., check if $(u, v), (v, w), (u, w) \in E$. There are $\binom{n}{3}$ such triples, and thus this algorithm takes $\Omega(n^3)$ time. There exist many algorithms [5], [6], [8]–[10] which provide significant improvement over the above method. A very comprehensive survey of the sequential algorithms can be found in [6], [10]. One of the state of the art algorithms is known as *NodeIterator++*, as identified in two very recent papers [8], [9].

NodeIterator++ uses a total ordering $\prec$ of the nodes to avoid duplicate counts of the same triangle. It is easy to see that use of any arbitrary ordering of the nodes, e.g., ordering the nodes based on their IDs, makes sure that each triangle is counted exactly once – counts only one permutation among the six possible permutations. However, the algorithm *NodeIterator++* incorporates an interesting node ordering based on the degrees of the nodes, with ties broken by node IDs, as defined below:

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

This degree based ordering can improve running time significantly, especially for a graph with skewed degree distribution.

We modify *NodeIterator++* by performing the comparison $u \prec v$ for each edge $(u, v) \in E$ in a preprocessing step rather than doing it while counting the triangles. This preprocessing step reduces the total number of $\prec$ comparisons to $O(m)$ and allows us to use efficient set intersection operation. All triangles containing node v and any $u \in N(v)$ can be found by set intersection $N(u) \cap N(v)$.

III. The Parallel Algorithm

Let P be the number of processors used in the computation. The network is partitioned into P partitions, and each processor is assigned one such partition $G_i(V_i, E_i)$. Processor i is responsible for counting triangles incident on the nodes in V_i^c, called *core* nodes for processor i, where $V_i^c \subset V_i \subset V$, such that for any i and j, $V_i^c \cap V_j^c = \emptyset$ and $\bigcup_i V_i^c = V$. Set V_i contains all nodes in V_i^c and any node w that is a neighbor of some node $v \in V_i^c$. Each processor, in parallel, reads its own part of the network (the data that is necessary to construct its own partition G_i) in its local memory and does computation on G_i. Once all processors complete their local computation, the results are combined. Our algorithm uses degree-based ordering that reduces $|N(v_i)| \leq d_v$; nodes with larger degree have larger reduction. Thus, degree-based ordering smoothers skewness of degree and provides very good load balancing.

We perform our experiments using a computing cluster (Dell C6100) with 30 computing nodes and 12 processors (Intel Xeon X5670, 2.93GHz) per node. The memory per processor is 4GB, and the operating system is SLES 11.1. The runtime performance of our algorithm is significantly better than the only available distributed memory parallel algorithm provided in [9]. The reason behind this improvement is, [9] has an hadoop implementation that generates huge volume of intermediate data, which are all possible 2-paths centered at each node. The algorithm shuffles and regroups these 2-paths, which take significantly larger time and also memory.

A. Computation of Clustering Coefficients

To compute clustering coefficients of the nodes, we need to count the triangles incident on each node v. Each processor i keeps track of the counts for the nodes in V_i. Processor i needs to collect and aggregate the counts for $v \in V_i^c$ from other processors. One way to do it is to maintain an array of counts of size n for storing the count information of any of the n vertices of the graph. Then the arrays in all processors can be aggregated simply by MPI function MPI_Reduce. However, we cannot afford $O(n)$ space for each processor. Instead, we adopt an external-memory approach for aggregating the counts for the individual nodes. Each processor creates P intermediate files: one for each processor, and writes the counts for the core nodes of the other processors in the corresponding files. Once all processors are done with creating the intermediate files, each processor merges (aggregates) the counts for its own core nodes $v \in V_i^c$ from the files created for it by other processors.

IV. A Sparsification-based Parallel Approximation Algorithm

We integrate a sparsification technique, called DOULION, proposed in [7] with our parallel algorithm so as to be able to run very large networks. Our adopted version of DOULION provides more accuracy than DOULION.DOULION works as follows. Let $G(V, E)$ and $G'(V, E' \subset E)$ be the networks before and after sparsification, respectively. Network $G'(V, E')$ is obtained from $G(V, E)$ by retaining each edge, independently, with probability p and removing with probability $1-p$. Let $T(G')$ be the number of triangles in G'. The estimated number of triangles in G is given by $\frac{1}{p^3}T(G')$, which is an unbiased estimation. In our parallel algorithm, sparsification is done slight differently: each processor i independently performs sparsification on its partition $G_i(V_i, E_i)$ instead of sparsifying the whole graph at the beginning. As processor i and j perform sparsification independently, survivals of two non-edge-disjoint triangles (v, u, w) and (v', u, w) are independent events. This improves accuracy of our algorithm.

For more details on any of our algorithms and information about experiments and results, our technical report ([11]) may be consulted.

V. Conclusion

We presented parallel algorithms for counting triangles and computing clustering coefficient of the nodes in a massive network that has billions of nodes and edges. The algorithm shows very good scalability with both the number of processors and the problem size and performs well on both real-world and artificial networks. Further, we have adopted the sparsification approach of DOULION in our parallel algorithm with improved accuracy. This adoption will allow us to deal with even larger networks.

References

[1] M. McPherson *et al.*, "Birds of a feather: Homophily in social networks," *Annual Review of Sociology*, vol. 27, no. 1, pp. 415–444, 2001.

[2] L. Becchetti *et al.*, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proc. of KDD*, 2008.

[3] Z. Bar-Yosseff *et al.*, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proc. of the 13th SODA*, 2002, p. 623632.

[4] N. Alon *et al.*, "Finding and counting given length cycles," *Algorithmica*, vol. 17, pp. 209–223, 1997.

[5] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *Proc. of Experimental and Efficient Algorithms*, 2005, pp. 606–609.

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

[7] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "Doulion: counting triangles in massive graphs with a coin," in *Proc. of KDD*, 2009, pp. 837–846.

[8] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *Proc. of KDD*, 2011, pp. 672–680.

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

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

[11] S. Arifuzzaman, M. Khan, and M. Marathe, "NDSSL 12-042," Virginia Bioinformatics Institute, Tech. Rep., 2012.