Parallel Algorithms for Finding, Counting and Listing Triangles and Quadrilaterals in Large Graphs

Xiaoyuan Guo, Hanyi Yu, Yifei Ren

May 5, 2019

1 Introduction

Graphs and matrices are used by social network analysts to represent and analyze information about patterns of ties among social actors. In network analysis, data is usually modeled as a graph or set of graphs. A graph is a data structure that has a finite set of nodes, called vertices, together with a finite set of lines, that join some or all of these nodes [4]. A graph might have hundreds of millions of nodes and edges and counting the number of triangles is very time-consuming. Besides, triangle listing is one of the fundamental algorithmic problems whose solution has numerous applications especially in the analysis of complex networks, such as the computation of clustering coefficient, transitivity, triangular connectivity, etc [3] [5].

In this project, we implemented MPI-based and Spark-based triangle-finding algorithms. As an extension, we realized corresponding quadrilateral-finding algorithms. Also, since many graph datasets are directed, we also tried to implement parallelized triangle-finding algorithm for directed graphs.

2 Methods

2.1 Preliminaries

Here are the preliminary notations, definitions in this project. The graph is denoted by $G(V, E)$, where V and E are the set of vertices(nodes) and edges, respectively [1]. $m = |E|$ and $n = |V|$. The set of all neighbors of $v \in V$ is denoted by N_v. The degree of v is $d_v = |N_v|$. A triangle is composed with three nodes $u, v, w \in V$ with edges $< u, v >, < u, w >, < v, w > \in E$.

2.2 MPI-based Triangle Finding in Undirected Graph

MPI(Message Passing Interface) is a parallel computing tool to help users improve computation efficiency with multiple computing nodes. Therefore, to solve the memory-constriant issue, we can develop MPI-based triangle-finding algorithm by splitting the input files into several chunks and process them simultaneously. In [1, 2], the authors proposed an efficient MPI-based parallel algorithm for finding the exact number of triangles in a graph when main memory of each compute node is large enough to contain the entire graph. The code implementation can be found in `https://github.com/cbaziotis/patric-triangles`.

2.3 MPI-based Quadrilateral Finding in Undirected Graph

We propose to find quadrilaterals based on previous triangle-finding process since quadrilaterals can be formed by two distinct triangles sharing one edge. Therefore, we can use the results of triangles found by last step and match two triangles to find quadrilaterals. However, since some quadrilaterals are not formed by two triangles, this method cannot correctly detect all quadrilaterals. To solve this limitation, **MPI-based Quadrilateral Finding** algorithm is developed to detect quadrilaterals in undirected graphs. The core algorithm that how to detect quadrilateral without previous triangle finding information is implemented. The algorithm is shown as follows:

```
Algorithm 1: Quadrilateral-Counting Algorithm
  Result: Quadrilateral lists $Q$
  Input: $AsBs$(edge pairs);
  Read input edge lists and build adjacency list $L$(each pair is stored twice, e.g. input
    $< a, b >$ is stored as both $< a, b >$ and $< b, a >$);
  $S = \phi, Q = 0$;
  for $v \in V$ do
      for $u \in V$ and $v! = u$ do
          $S = N_u \cap N_v$;
          $Q = Q \cap \{u, v, x, y\}$, for $x! = y, x, y \in S$;
      end
  end
```

MPI communication: When the input edge files become large, it is very difficult to compute all the intersection of each node pair because of memory limitation. To solve the problem, we first split the large input files into smaller ones, the number of small files is decided by how many processors will be used, in this way, each processor will read its own input edge files. The split process is performed by counting the total number of nodes and then assigning equal number of nodes to each processor. Suppose we use P processors, then each processor will have V/P nodes. To ensure each node(with its neighbors) is assigned to only one unique processor, a sorting process(for the original input files) is performed before splitting. Since each node pair should do intersection computation to check whether a quadrilateral exists, when the node pair occurs in different processors, it is inevitable for one processor to request data from the other processor. Thus, each processor has its own data and will find quadrilaterals using Algorithm 1 in itself. After that, processor1 will package its data and send it to other processors, the other processors will unpackage the data and run Algorithm 1 using its own data and received data; then processor2 will send its data to other processors whose rank value is greater than itself, and so on. When all nodes are paired with each other, each processor will send back found quadrilaterals to processor1. The first processor will collect all the quadrilaterals and find unique ones to filter the duplicated findings. In this case, the algorithm can finally give us the total number of quadrilaterals. The code implementation has been uploaded to `https://github.com/XiaoyuanGuo/Parallel_Computing.github.io`.

Complexity analysis: The core algorithm to finding quadrilateral traverses all the vertices V of the input graph G , for each vertex v, it traverses all the other vertices $u \in V$ to compute the intersection of their neighbors N. Only when those intersection has more than one same vertex can the intersection detect possible quadrilaterals composed by the two main vertices and their two intersection vertices. Therefore, the running time requires at most $O(n^2 {d_{max}}^2)$. Since there may have same quadrilaterals with same vertices formed but detected in different steps, eliminating the replicates is also an important step to perform. This process requires traversing all the detected quadrilaterals Q, of which the total number is less than the total vertices. Thus, the cost of this step can be ignored with respect to $O(n^2 {d_{max}}^2)$.

2.4 Spark-based Triangle Finding in Undirected Graph

Apache Spark is an open-source unified analytics engine for large-scale data processing. Spark stores data on multiple nodes and each node can independently process data on it. Since Spark has features of in-memory processing and lazy evaluation, it avoids many IO operations on disk and runs much faster than Hadoop MapReduce. Thus we build a small Spark cluster to solve triangle finding problem. Here we implement the algorithm in [4], which mainly contains following steps:

(1) Read all edges and store them as key-value pairs in set P_1, e.g. "$a\ b$" $\Rightarrow \{(a : b), (b : a)\}$;

(2) Group P_1 by key, generate key-value pair set P_2, e.g. $\{(a : b_1), (a : b_2), (a : b_3)\} \Rightarrow (a : [b_1, b_2, b_3])$;

(3) For each element in P_2, map it to a set of key-value pairs and add them to P_3, the mapping could be described as:

$$\begin{aligned}(a : [b_1, b_2, \ldots, b_n]) \Rightarrow & \{((a, b_1) : -), \ldots, ((a, b_n) : -), \\ & ((b_1, b_2), a), ((b_1, b_3), a), \ldots, ((b_{n-1}, b_n), a)\}\end{aligned}$$

where "-" means the pseudo node and represents that the corresponding key is an existing edge;

(4) Group P_3 by key, generate key-value pair set P_4, e.g. $\{(a, b) : c_1), (a, b) : c_2), (a, b) : c_3)\} \Rightarrow ((a, b) : [c_1, c_2, c_3])$;

(5) For each element in P_4, if its value contains the pseudo node, map it to a set of tuples and add them to P_5, e.g. $((a, b) : [-, c_1, \ldots, c_n]) \Rightarrow \{(a, b, c_1), \ldots, (a, b, c_n)\}$;

(6) Eliminate duplicate tuples in P_5 to obtain the final result.

2.5 Spark-based Quadrilateral Finding in Undirected Graph

A quadrilateral is actually a point set where two points are connected via another two points. To realize quadrilateral finding algorithm, we applied some modifications to previous Spark-based triangle finding algorithm.:

(1) Read all edges and store them as key-value pairs in set P_1, e.g. "$a\ b$" $\Rightarrow \{(a : b), (b : a)\}$;

(2) Group P_1 by key, generate key-value pair set P_2, e.g. $\{(a : b_1), (a : b_2), (a : b_3)\} \Rightarrow (a : [b_1, b_2, b_3])$;

(3) For each element in P_2, map it to a set of key-value pairs and add them to P_3, the mapping could be described as: $(a : [b_1, b_2, \dots, b_n]) \Rightarrow ((b_1, b_2), a), ((b_1, b_3), a), \dots, ((b_{n-1}, b_n), a)\}$;

(4) Group P_3 by key, generate key-value pair set P_4, e.g. $\{(a, b) : c_1), (a, b) : c_2), (a, b) : c_3)\} \Rightarrow ((a, b) : [c_1, c_2, c_3])$;

(5) For each element in P_4, if its value contains more than one node, map it to a set of tuples and add them to P_5, e.g. $((a, b) : [c_1, \dots, c_n]) \Rightarrow \{(a, b, c_1), \dots, (a, b, c_n)\}$;

(6) Eliminate duplicate tuples in P_5 to obtain the final result.

The code implementations of Spark-based algorithms have been uploaded to `https://github.com/hyu88/HPC-CS581.git`.

2.6 Triangle Finding in Directed Graph

The serial version of directed triangle finding is straightforward. For a graph with n vertices, we use a $n \times n$ binary matrix E to store the edges. $E(i, j) =$ True means that there is an edge from vertex i to vertex j. Then we used a nested loop to traverse each possible triplet (i, j, k). Once $E(i, j) = E(j, k) = E(k, i) =$ True and $i \neq j, j \neq k, k \neq i$, the count adds one. In case of directed graph, the number of permutation would be 3 (as order of nodes becomes relevant). Hence in this case the total number of triangles will be obtained by dividing total count by 3.

The parallelized algorithm for directed triangle finding is based on Spark. Below is the main steps:

(1) Read all edges and store them as key-value pairs in set P_1, e.g. "$a\ b$" $\Rightarrow (a : b)$;

(2) Exchange the key and the value in P_1 to generate set P_2, e.g. $(a : b) \Rightarrow (b : a)$;

(3) Join P_1 and P_2 to generate P_3, e.g. $(a : b) \bowtie (a : c) \Rightarrow (a : (b, c))$, it means that there is a triad "$c \rightarrow a \rightarrow b$";

(4) Exchange the key and the value in P_3 to generate set P_4, e.g. $(a : (b, c)) \Rightarrow ((b, c) : a)$;

(5) Map P_1 to P_5, e.g. $(b : c) \Rightarrow ((b : c) : -)$, "-" means the pseudo node and means that there is an edge $b \rightarrow c$;

(6) Join P_4 and P_5 to generate set P_6, e.g. $((b, c) : a) \bowtie ((b, c) : -) \Rightarrow ((b, c) : (a, -))$, it means that a triad "$c \rightarrow a \rightarrow b$" and an edge $b \rightarrow c$ construct a directed triangle;

(7) Map key-value pairs in P_6 to ordered tuple in P_7, e.g. $((b, c) : (a, -)) \Rightarrow (a, b, c)$;

(8) Eliminate duplicate tuples in P_7 to obtain the final result.

3 Experiment Results

We evaluated the performance of our algorithm for undirected graphs with datasets from `http://snap.stanford.edu/data/ego-Facebook.html` and `https://snap.stanford.edu/data/ca-AstroPh.html`. ego-Facebook dataset contains 4,039 nodes, 88,234 edges and 1,612,010 triangles, while Astro Physics collaboration network dataset contains 18,772 nodes, 198,110 edges and 1,351,441 triangles. To test the scalability of algorithms, wo use above datasets to generate some subsets with certain number of vertices.

For MPI-based triangle finding algorithm, we test our programs on the BMI cluster. The strong scaling is evaluated with a dataset of 100000 vertices, and the results is shown in Fig.1(a). The weak scaling is evaluated in the way that N processors process the dataset of $1000 \times N$ vertices. The result is shown in Fig.1(b). Similarly, Fig.1(c) and Fig.1(d) show the strong and weak scalability of MPI-based quadrilateral finding algorithm.

To test Spark-based algorithms, we created an cluster on Google Cloud Platform. Since the limitation of worker nod number, we have to run our programs with at least two worker nodes. Besides, we read time cost from the web application of GCP, so it should contains the time to start and stop Hadoop, Yarn, and Spark services. Fig.1(e) shows the time cost of Spark-based triangle finding algorithm while Fig.1(f) is for the quadrilateral finding algorithm.

For direct triangle finding, we test with dataset from `http://snap.stanford.edu/data/wiki-Vote.html`, which contains 7,115 nodes, 103,689 edges, and 508,389 triangles. The algorithm time cost is 18.45s.

4 Conclusions

It is very hard to directly extend triangle finding algorithm to quadrilateral finding algorithm, unless the aim is to find those quadrilaterals that consists of two triangles sharing one edge. For MPI-based algorithms, triangle finding and quadrilateral finding algorithms require the input file to be ordered, therefore a preprocess step is required before runing the algorithms. As for Spark-based algorithms, there is no requirement for input order. The time complexity of quadrilateral finding algorithms are obviously higher than triangle finding algorithms.

5 Contribution

Xiaoyuan Guo mainly focuses on MPI-based triangle and quadrilateral finding algorithms for undirected graphs. She is responsible for completing the design of simple-version triangle finding, MPI-based triangle finding, simple-version quadrilateral finding and MPI-based quadrilateral finding. The corresponding part of coding, slides and reports is done by herself.

Hanyi Yu mainly focuses on Spark-based triangle and quadrilateral finding algorithms for undirected graphs. He completed the implementation corresponding codes by himself. He also contributed to the design of Spark-based triangle finding algorithm for directed graphs.

Yifei Ren main focuses on triangle finding algorithms for directed graphs. She implemented simple-version and Spark-based directed triangle finding algorithms.

References

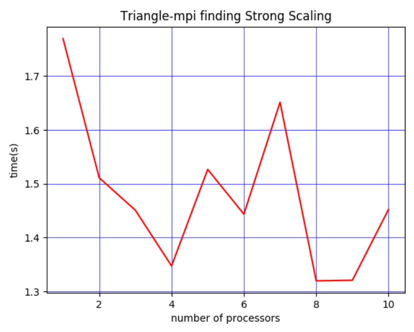
[1] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 529–538. ACM, 2013.

[2] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. A fast parallel algorithm for counting triangles in graphs using dynamic load balancing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1839–1847. IEEE, 2015.

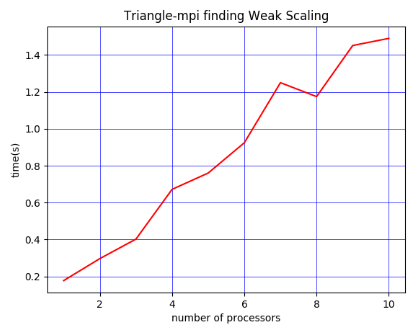
[3] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680. ACM, 2011.

[4] Mahmoud Parsian. *Data Algorithms: Recipes for Scaling Up with Hadoop and Spark.* " O'Reilly Media, Inc.", 2015.

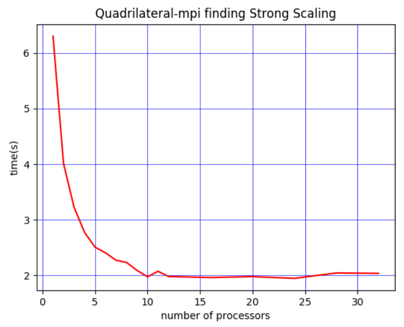
[5] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*, pages 606–609. Springer, 2005.



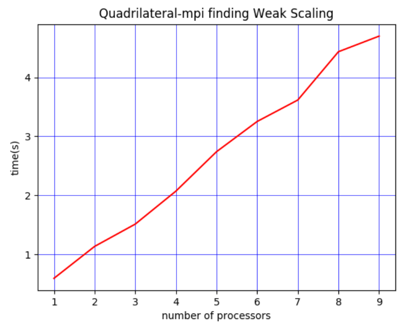
(a)



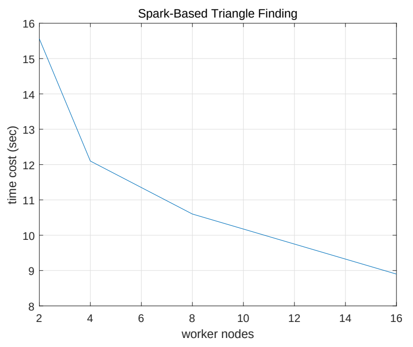
(b)



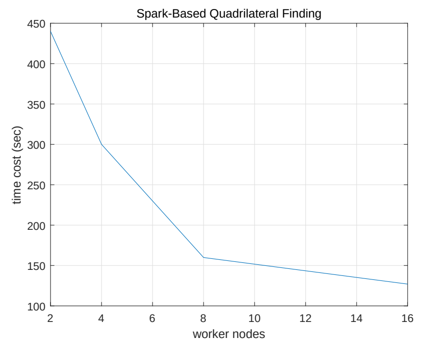
(c)



(d)



(e)



(f)

Figure 1: Algorithm scalability