15-418 Final Report

Shu-Hao Yu, YiCheng Qin Andrew ID: shuhaoy, yichengq

1 Project Summary

Social network websites are popular these years. There are many researches related to the topology of social network. Triangle counting is one basic approach to describe the information of dataset, which can measure the closeness in a social network. On the other hand, the size of these networks grow rapidly, we will need a scalable algorithm on this problem. In this project, we implemented triangle counting algorithm on CUDA. We are able to gain 10x speedup than the fastest sequential version and got comparable result with state-of-art multi-core GraphLab toolkit.[1]

2 BackGround

We use figure 1 as an example of triangle counting. In this example, there are three triangles in this graph, {Pachu, Andrew, Matt}, {Ben, Stephen, Chuck}, {Chuck, Stephen, Rajat}. As the graph get larger, you can expect the complexity becomes higher. When there are n users, there will be $\binom{n}{3}$ possible triangles. Our mission is to make this problem parallelized thus gaining the speedup.

We have explored triangle counting problem to the very detail. One of branch in this problem is to find the approximate number of triangles.[2][3] However, though it can achieve much higher speedup, we found it hard to make comparison between each other since the different degree of approximation affected the running complexity, also it's hard to choose one to implement parallelized code. We then decided to do on exact triangle counting.

In Thomas Schank's phd dissertation,[4] he clearly explained the existing algorithms on this problem. The method for triangle counting can be categorized into five types: try-all, matrix-multiplication, tree-lister, node-iterator, edge-iterator. We plan to optimization based on these methods using all tools that we can use. On the other hand, there is no good parallel

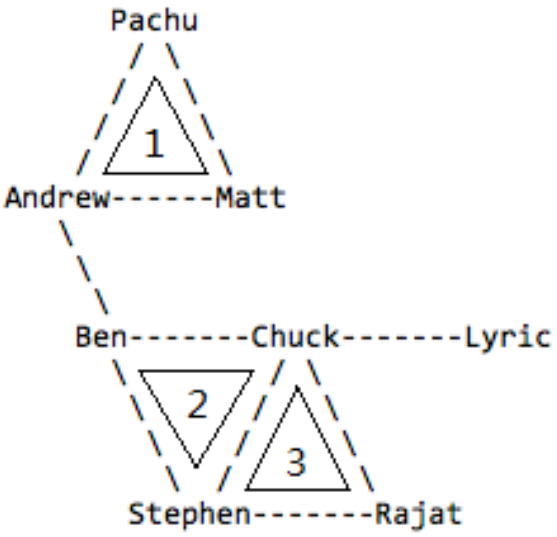


Figure 1: Triangle Counting Example

solution until 2011, when Yahoo researchers Suri and Vassilvitskii published a solution on Hadoop which avoid the curse of last reducer.[5]

3 Dataset

We used undirected graph data as our dataset. Stanford Network Analysis Project(SNAP) gives us an abundance of dataset. We basically used two dataset, one is self-made dataset with few lines for debugging, LiveJournal and AS-Skitter from SNAP. We used LiveJournal and AS-Skitter as our final experiment datasets, which includes about 4M nodes, 34M edges and 1.7M nodes, 11M edges respectively.

The dataset from SNAP contains complete information, including the number of nodes, edges, cluster coefficient, and most important part for us, the number of triangles. Thus, we can easily examine our result to the real answer. We listed information about LiveJournal and AS-Skitter in the table 1.

Basic Information	LiveJournal	AS-Skitter
Nodes	3,997,962	1,696,415
Edges	34,681,189	11,095,298
Average Clustering Coef.	0.3538	0.2963
#Triangle	177,820,130	28,769,868
Diameter	18	25

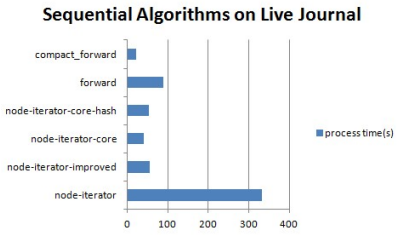
Table 1: Basic Information of Dataset

4 Existing Algorithms Implementation

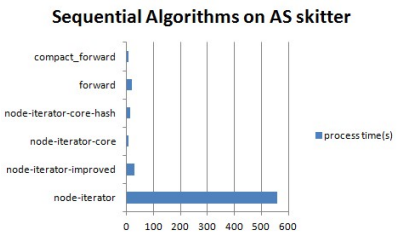
There are plenty of sequential algorithms in this problem. We implemented them based on Schank's dissertation for comparison. We implemented six algorithms of them. On the other hand, we implemented Hadoop version from Yahoo! research paper. Though in the end we found that we are not able to replicate the experiments since they used huge number of nodes, it gives us good sense to go on GPU version algorithm.

We used "node-iterator", "node-iterator-improved", "node-iterator-core", "node-iterator-core-hash", "forward", "compact_forward" from Schank's paper. And we also implemented "node-iterator++" and "Partition" from Yahoo! research paper. (The terms are consistent with both paper.)

We can see in figure 2(a) and 2(b), "compare-forward" are the best algorithm in both dataset. Moreover, we found the framework "edge-iterator", which it based on, is very helpful for us to do the parallelized algorithm.



(a) LiveJournal dataset



(b) AS-Skitter dataset

Figure 2: Sequential algorithms time omparison

5 Proposed Approach and Analysis

5.1 Cuda Compact Forward

```
Compact Forward(){
  //Assume array of vertices in non-increasing degrees,
  and neighbor arrays sorted in the same order as array of nodes.
  Sum = 0;
  for each edge E do
      Ns = start node of E;
      Ne = end node of E;
      if Ns < Ne then
          continue;
      end
      Ns_i = first neighbor of Ns;
      Ne_i = first neighbor of Ne;
      while Ns_i < Ne and Ns_i exists and Ne_i exists do
          if Ns_i < Ne then
              Ns_i= next neighbor of Ns;
          else
              if Ns_i > Ne_i then
                  Ne_i = next neighbor of Ne;
              else
                  Sum++;
                  Ns_i = next neighbor of Ns;
                  Ne_i = next neighbor of Ne;
              end
          end
      end
  end
  return Sum;
}
```

Algorithm 1: Original Compact Forward Algorithm

Algorithm above is the sequential version of Compact Forward algorithm. The Basic idea is to parallel based on edge iterator.

In detail, there are at two ways to finish compact forward in parallel. One way is to let all edges of a node be a single task for a block, while the other one is to make each edge be a single task. Because parallel works usually expect smaller and evener workload, iterating on edges is a better choice.

Based on the native first version, a lot of problems are discovered for further improvement.

5.2 Workload Balance V.S Cache Utilization

Workload balance is a troublesome problem here. Either in node iterator or in edge iterator, there exists serious load balance problem. The workload for edges depends on the number of neighbors with two verticles, which may differ from two to several thousands. In the original order of edge, it shows that active warps/active cycle only equals to 2.077 from statistics made by Compute Visual Profiler. This indicates the waste of many idle threads. The best way to handle workload balance is put threads of similar in a block.

How to utilize cache efficiently is another focus. Reuse the data in cache as much as possible, which helps to reduce the throughput and latency from global memory to core. So it is ideal that a large part of edges in a block have the same vertex to check. It is hard to say what is the best way to do it, but checking the edges that start from the same node in a block utilize cache well.

However, these two requirements are hard to fit well at the same time. For example, if we sort all edges by workload, the threads in a block will have good balance but not utilizing cache well. On the other hand, merely sorting by node number can have better use of cache but it is not work balanced.

5.3 Sorting and Remapping nodes

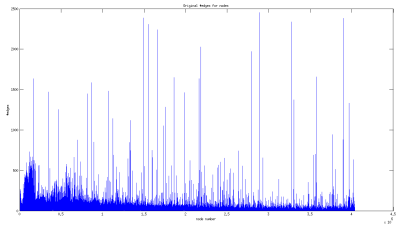
We tried to solve this problem in LiveJournal dataset. Originally we can sort edges by node number, in this way cached is used, but the adjacent nodes will have different workload.(Figure 3(a))

We would like to achieve balance and cache usage at a same time. Our first try is to sort nodes by the number of neighbors first. Then we remap the node number, give number 1 to the most neighbor node, and number 2 to the node with second neighbors... In this way, though the adjacent nodes have similar workload, the distribution is very skew, which scale from 0 to 25000 (Figure 3(b)), which causes worse balanced problem since sometimes we have to deal with these heavy nodes .

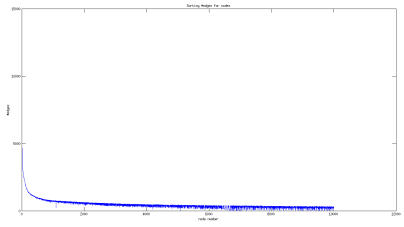
5.4 Flattening Algorithm

This reordering method, though looks simple, is our main contribution on this problem. We first do the same sorting nodes by the number of neighbors. The difference is in the second step. We then assign the most popular node to be with last node number. Since it's an

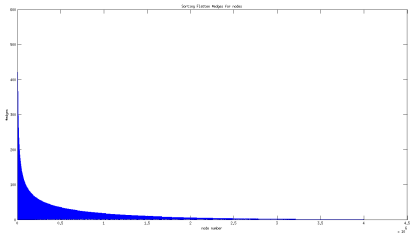
undirected graph, and we only store an edge once with $e_1 = (n_1, n_2)$, *where* $n_1 < n_2$. Now the most popular node has 0 edge. The total distribution become very balanced with scale 0 to 600 and each adjacent nodes with similar workload, which seems have good work balance and is able to utilize cache. (Figure 3(c))



(a) Original dataset



(b) Sorting and Remapping dataset



(c) Flattening dataset

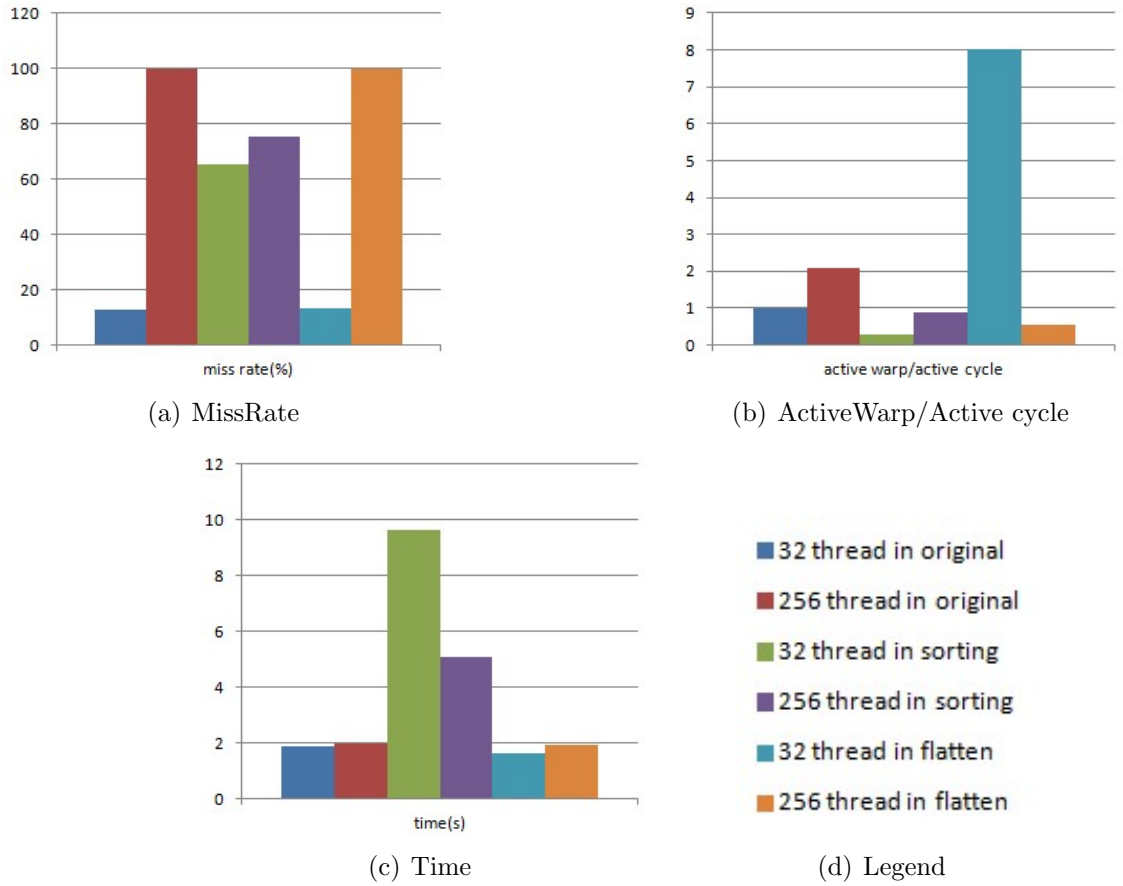
Figure 3: Distribution by different ordering methods

5.5 Result Analysis

Using CUDA profiler gives us great criteria to evaluate the strength and weakness in each method. We run our three ordering methods in 32 and 256 threads in a block.

First we look into miss rate, which is the criteria about cache usage. For original and flattened dataset in 32 threads, they used cache quite well. However, if there are 256 threads, the miss rate hit to 100% since threads are contending with the cache. On the other hand, skew distribution cannot use cache well, since a node's edges are lot, which cannot be in the same block, using more threads can enhance the ability to incorporate edges to a set but also increase contending.

For the active warp per cycle, we measure how often the cores are used to computation. There are many reason to cause warp idle, such as data dependency, unbalanced workload, memory IO...et al. In our experiment, only flattened dataset with 32 threads work well, too many thread caused memory contending thus warps are idle. For original dataset, its



(a) MissRate (b) ActiveWarp/Active cycle

(c) Time (d) Legend

Figure 4: Profiler Result from different ordering methods

unbalanced structure will cause some warps idle. And for skew distribution, it suffered both unbalanced dataset and memory reading.

Overall in time, skew distribution gave us worse result. Flattened ordering gives us about 15–20% time off to the original dataset.

5.6 Restriction and more optimization

Through the process of project, we find many restrictions of triangle counting to be fully parallelized on GPU. The first factor is that workload changes from edge to edge. Moreover, due to the big difference between node neighbor numbers, workload imbalance affects result a lot. The second factor is that the application is based on big data. Small cache size of GPU core cannot hold much of needed data, and fetching data repeatedly lowers the performance. The third factor is the heavy load for pre-work that is using or to use. In current best version,

LiveJournal	graphlab 1 core	graphlab 2 core	graphlab 4 core	graphlab 8 core	compact forward	**cuda compact forward**
handle time	25.986s	14.298s	10.081s	9.929s	20.789s	**2.057s**
Speedup	0.80x	1.45x	2.06x	2.09x	1x	**11.83x**

Table 2: Performance of algorithms on LiveJournal

60% of time is used for pre-work, especially sort function. It means that sort function will restrict more speedup later, and it doesn't leave much space for more pre-work.

The speedup compared to the best sequential algorithm now is 10x, and we expect it to be faster. We have tried to resort edges by workload and locality, divide into sets for best locality and so on. But overhead of extra pre-work brings strike on performance and prevents us from further optimization. We can see that any solution with edge sort has the limit of at most 2x more speedup on current one, because sort accounts for half of processing time now. On the other side, we have almost no expectation on solution without sort because it may lead to much redundancy on finding neighbors.

6 Performance

We run our experiments on two datasets. Using sequential "compact forward" as our baseline. Worth to mention is we compare with state-of-art multi-core GraphLab library.

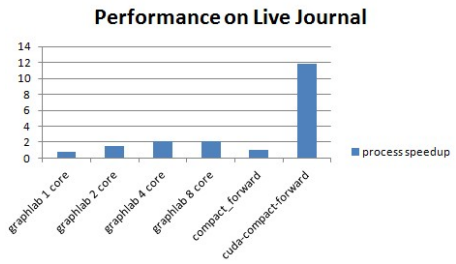
For the experiment setting, for sequential "compact forward" we used GHC 2.66GHz CPU. For GraphLab, we run on machine with 4 2.5GHz cores and with different number of cores. For our CUDA implementation, we used our flattening reordering with parallelized "compact forward" and run on GTX 480 GPU.

In table 2 and 3, we presented our results with two dataset. The results are consistent in two datasets. Also, we showed Figure 5(a) and 5(b) as a good comparison.

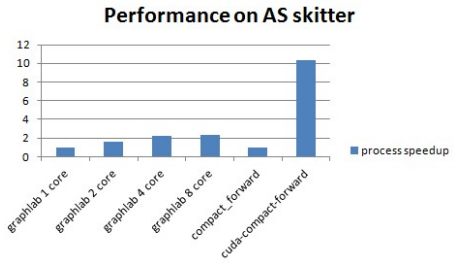
Compact forward is the best sequential algorithm and we used it as baseline. We then implemented this code in CUDA version, we can reach 10x speedup in both dataset. For comparing to GraphLab toolkit, we can still get 5x speedup.

AS-Skitter	graphlab 1 core	graphlab 2 core	graphlab 4 core	graphlab 8 core	compact forward	**cuda compact forward**
handle time	5.322s	3.210s	2.336s	2.207s	5.183s	**0.501s**
Speedup	0.97x	1.61x	2.22x	2.35x	1x	**10.35x**

Table 3: Performance of algorithms on AS-Skitter



(a) LiveJournal



(b) AS-skitter

Figure 5: Performance Comparison between algorithms

7 Attempts on scalable algorithm

```
SubGraphPartition(int n){
  Sum = 0;
  Randomly assign nodes to set[0...n-1];
  for i ∈ [0,n-1] do
    for j ∈ [i+1,n-1] do
      for k ∈ [j+1,n-1] do
        Form five sets for each (i,j,k):
        S_1 : {(e_1, e_2) ∈ (S_i, S_i) ∪ (S_j, S_j) ∪ (S_k, S_k)}
        S_2 : {(e_1, e_2) ∈ (S_i, S_i) ∪ (S_j, S_j) ∪ (S_i, S_j)}
        S_3 : {(e_1, e_2) ∈ (S_i, S_i) ∪ (S_k, S_k) ∪ (S_i, S_k)}
        S_4 : {(e_1, e_2) ∈ (S_j, S_j) ∪ (S_k, S_k) ∪ (S_j, S_k)}
        S_5 : {(e_1, e_2) ∈ (S_i, S_j) ∪ (S_i, S_k) ∪ (S_j, S_k)}
        each block calculate a set to get triangle in set;
        Δ_l = TriangleCounting(S_l);
        Sum += (Δ_2 + Δ_3 + Δ_4 + Δ_5 − Δ_1)
      end
    end
  end
  return Sum;
}
```

Algorithm 2: Cuda Subgraph Partition

Since CUDA has memory constraints and we cannot move all graph information to global memory when size grows. We wish to built up a scalable algorithm to deal with it. Partition algorithm we provided, in the end to be proved non-useful since when we reduce edges in a set to shared_memory, we will have much overhead which worsen our result much. We still listed this work here, which may be good reference. On the other hand, we developed a simpler algorithm which should be theoretically feasible, but we don't have time to implement in the end.

7.1 CUDA Subgraph Partition

We modified this algorithm from Hadoop Version. However, the challenge in Hadoop and CUDA is a little bit different. In Hadoop, their main goal is to solve last reducer problem while in CUDA, we also have to assure that in each subset, they are able to put in shared memory.

One more advantage for this algorithm is that it can be scalable. For now, we assume all edges can be moved into GPU memory. However, it will not be the case when dealing with larger dataset. In this framework, we are able to segment to smaller set and then transmit to GPU by several times.

7.2 CUDA equal-size Subgraph

This part is our proposal for the scalable algorithm which we not yet complete. The method is simple. When half dataset can be put in global memory but not whole dataset, we run subgraph algorithm from random node to get half of dataset. After calculating these two subgraph, we run through edges which two nodes are in different subgraph. This algorithm actually depends on the tightness of dataset, but since large datasets are sparse, it can be a good scalable algorithm with less overhead.

8 Summary

In this project, we learned there are tradeoff between memory structure, computation, work balanced. Also, we learned how to use CUDA profiler to evaluate the result. Thus, we provided a great reordering algorithm which gave us best result and backed up by detailed analysis. In the end, our algorithm is quite competing even to state-of-art GraphLab library. Also, we found out some restriction on this problem but also give thoughts on the scalable algorithms.

Week	What We Plan To Do	What We Actually Did
Apr 1-7	Do topic choice	Do topic choice
Apr 8-14	Research related work on Triangle Counting problem and submit the proposal	Collected related paper about Triangle Counting Problem, and explored existing solutions.
Apr 15-21	Multicore/GPU parallel solution on small data set	Implemented Serialized Triangle Counting algorithms and built up MapReduce algorithms in Hadoop.
Apr 22-28	Improve the algorithm to support large data set	Experiment our serialized and Hadoop solution on large dataset and start implementing algorithms on GPU
Apr 29-May 5	Speed up the algorithm by multiple machines	Finish CUDA version code and keep optimizing it.
May 6-11	Leave for additional work	Attempted Scalable CUDA Code and implemented flattening algorithm.

Table 4: Working Log

9 Working Log

YiCheng	Six sequential algorithms Cuda Framework Cuda compact forward algorithm Implement Flatten Algorithm Generate CUDA Profiler Result
Shu-Hao	Two Hadoop algorithms Cuda Partition algorithm Analyze Cache & Memory IO Results Presentation Slides Writeup Report

Table 5: Work Distribution

References

[1] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new parallel framework for machine learning," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, (Catalina Island, California), July 2010.

[2] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "Doulion: counting triangles in massive graphs with a coin," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, (New York, NY, USA), pp. 837–846, ACM, 2009.

[3] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis, "Efficient triangle counting in large graphs via degree-based vertex partitioning," *CoRR*, vol. abs/1011.0468, 2010.

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

[5] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th international conference on World wide web*, WWW '11, (New York, NY, USA), pp. 607–614, ACM, 2011.