Parallel Algorithm for Triangle Counting in an undirected graph

Pranali Yawalkar - CS11B046

Rishitha Dega - CS11B035

Siddhant Mutha - CS11B048

Ahana Chatterjee - CS14S002

pranali.yawalkar@gmail.com

rishitha.reddy.3@gmail.com

siddhantmutha@gmail.com

ahanacttrj@gmail.com

IIT Madras

Abstract

The number of triangles in a graph is an important metric in social network analysis. Two frequently used metrics in complex network analysis which require the count of triangles are the clustering coefficients and the transitivity ratio of the graph. Triangles have been used successfully in several real-world applications, such as detection of spamming activity, uncovering the hidden thematic structure of the web and link recommendation in online social networks. However finding the number of triangles sequentially in dense and huge graphs is computationally challenging.

I. Problem Definition

Because of the high complexity of sequential implementation of the problem, we are coming up with a parallel implementation of the problem of finding number of trianlges.

II. Problem Description - Deliverables - Done

Parallel implementation of the problem for finding triangles on *large datasets* based on Map Reduce strategy. Modified NodeIterator++ algorithm to be used to get the triangles parallely and **Metis** to be used for parallel graph partioning. Also tweaking the configuration parameters to get better run time and complexity.

III. Problem Description - Maybe - Done

I. Application of the Parallel implementation

To use the parallel algorithm to find triadic closures and clustering coefficients of Social Circles : Facebook network available on `http://snap.stanford.edu/data/egonets-Facebook.html`. Details about the data -

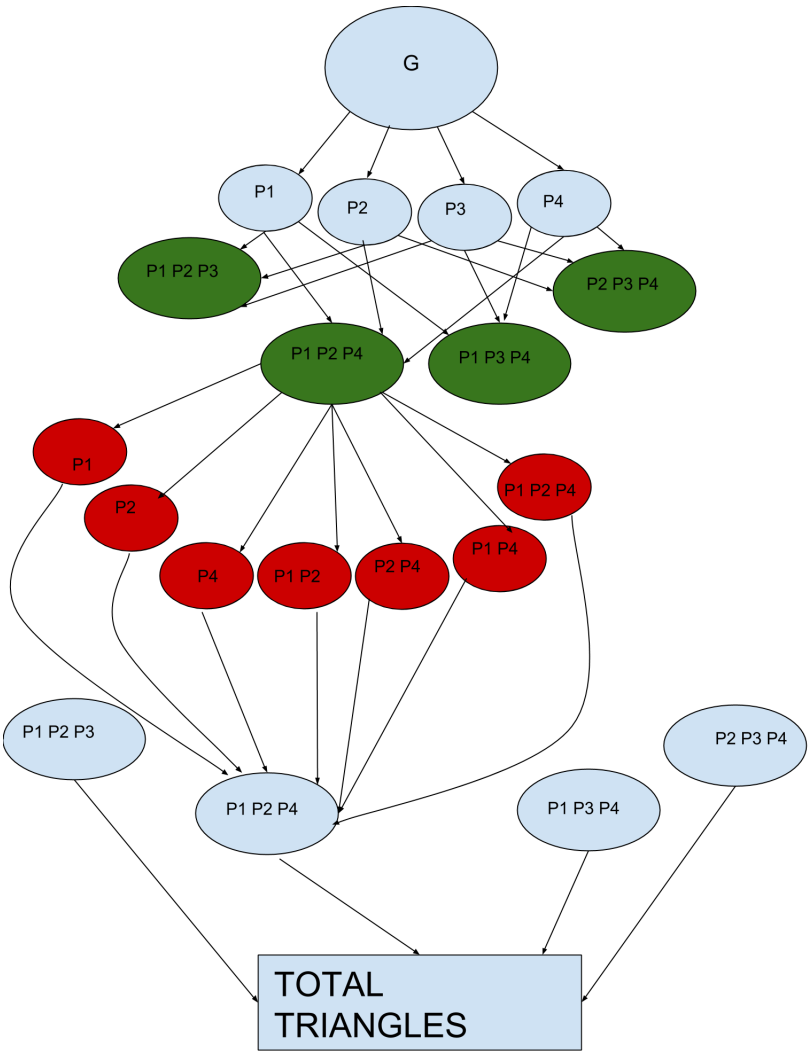
1. Nodes = 4039
2. Edges = 88234

IV. Problem Description - Maybe

To implement sequential BFS and parallel BFS for graph partioning and compare it with results of Metis based on time and parallelism metrics.

V. Pseudo Code

The architecture for partitioning, processing and combining the results can be seen in the following figure -



In the above graph, the green nodes are spawned parallely in the first level of parallelism and the red ones are spawned parellely by each of the threads spawned in the first level.

Listing 1: Graph Partition and Triangle Count

```
Function–Graph Partition and Triangle Count
Input – undirected unweighted G = (V,E) and k = total number of partitions
Output – Total number of triangles and the incident triangles for
each node.
Number_of_Triangles = 0
Split the graph naively (and later on using metis) into k
partitions = P1 P2 ...Pk
for every three partitions Px Py Pz such that x < y < z
        answer = Find_Triangles_Partitions(Px Py Pz)
        for all nodes n
                Number_of_Triangles(n) += answer(n)
return (∑forallnodes Number_of_Triangles) / 3
Function –Find_Triangles_Partitions(Px Py Pz)
Input – undirected unweighted Partitions Px Py Pz
Output – Total number of triangles in the subgraph {Px Py Pz}
Triangle_Count = 0
Triangle_Count += Node_Iterator1++(Px)
Triangle_Count += Node_Iterator1++(Py)
Triangle_Count += Node_Iterator1++(Pz)
Triangle_Count += Node_Iterator2++(Px, Py)
Triangle_Count += Node_Iterator2++(Px, Pz)
Triangle_Count += Node_Iterator2++(Py, Pz)
Triangle_Count += Node_Iterator3++(Px, Py, Pz)
return Triangle_Count
```

Listing 2: Total Triangles per node in the partition

```
Function – NodeIterator1++(Px)
Input – undirected unweighted Partition = (Vx,Ex)
Output – Total number of triangles in input partition and the
incident triangles for each node
Count_of_Triangles = 0
for every node n1 in Px
        for every node n2 in Px
                for every node n3 in Px
                        if( n1 < n2 < n3)
                        if(<n1,n2> ∈ E && <n2,n3> ∈ E && <n1,n3> ∈ E)
                                z = (p(n1) choose 2) + p(n1)(k − p(n1) − 1) + (k−p(n1)−1 choose 2)
                                Count_of_Triangles(n1) += 1/z
                                Count_of_Triangles(n2) += 1/z
                                Count_of_Triangles(n3) += 1/z

return Count_of_Triangles
```

Listing 3: Total Triangles per node across the 2 partitions

```
Function – NodeIterator2++(P_x , P_y)
Input – undirected unweighted Partition = (V_x ,E_x , V_y ,E_y)
Output – Total number of triangles across the two input partitions and
the incident triangles for each node
Count_of_Triangles = 0
for every node n1 in P_x
        for every node n2 in P_x
                for every node n3 in P_y
                        if( n1 < n2 )
                        if(<n1,n2> ∈ E && <n2,n3> ∈ E && <n1,n3> ∈ E)
                                z = k – 2
                                Count_of_Triangles(n1) += 1/z
                                Count_of_Triangles(n2) += 1/z
                                Count_of_Triangles(n3) += 1/z
for every node n1 in P_x
        for every node n2 in P_y
                for every node n3 in P_y
                        if( n2 < n3 )
                        if(<n1,n2> ∈ E && <n2,n3> ∈ E && <n1,n3> ∈ E)
                                z = k – 2
                                Count_of_Triangles(n1) += 1/z
                                Count_of_Triangles(n2) += 1/z
                                Count_of_Triangles(n3) += 1/z

return Count_of_Triangles
```

Listing 4: Total Triangles per node across the 3 partitions

```
Function – NodeIterator3++(P_x, P_y, P_z)
Input – undirected unweighted Partition = (V_x,E_x, V_y,E_y, V_z,E_z)
Output – Total number of triangles across the two input partitions and
the incident triangles for each node
Count_of_Triangles = 0
for every node n1 in P_x
        for every node n2 in P_y
                for every node n3 in P_z
                        if(<n1,n2> ∈ E && <n2,n3> ∈ E && <n1,n3> ∈ E)
                                z = 1
                                Count_of_Triangles(n1) += 1/z
                                Count_of_Triangles(n2) += 1/z
                                Count_of_Triangles(n3) += 1/z

return Count_of_Triangles
```

Listing 5: Parallel BFS

```
Function – PBFS(v_0)
Input – root node v_0
Output – BFS Partitions of the graph
cilk_for each vertex v in V (G) – {v_0}:
     v.dist = –1
v_0.dist = 0
d = 0 // depth
V_0 = Create a new cilk::reducer_list
V_0.insert(v_0) // insert v0 into the V0 list
V_p = V_0 // Vp is the pointer to the list for the current layer in the BFS
while not V_p is empty:
    V_child = new cilk::reducer_list // children list
    PROCESS–LAYER(V_p, V_child, d)
    d = d + 1
    add V_child to global list of partitions
    V_p = V_child // set the current child as the new parent
```

Listing 6: PBFS Process Layer

```
Function – PROCESS–LAYER(parent–list, children–list, depth)
Input – parent–list, children–list, depth
Output – Fills the children–list with the children of the parent–list
if parent–list size < GRAINSIZE:
    for each u in parent–list:
        parallel for each v in Adj[u]:
            if v.dist == –1:
                v.dist = depth + 1 // benign race here, doesn't affect correctness
                children–list.insert(v)
     return
```

```
split children-list from the middle into children-list_part1 and children-list_part2
cilk_spawn PROCESS-LAYER(parent-list, children-list1_part1, d)
cilk_spawn PROCESS-LAYER(parent-list, children-list1_part2, d)
cilk_sync
```

VI. Testing Plan

- Input -
  1. Synthetic data for around 500 nodes.
  2. Dataset obtained from `http://snap.stanford.edu/data/wiki-Vote.html` with known value of number of triangles.
- Output - Number of triangles.
- Correctness - Known number of triangles in the synthetic data and known value of the same available on SNAP for the SNAP dataset.

VII. Division of Labor

- Graph Partioning - Ahana, Siddhant
- Finding number of triangles in each partition based on NodeIterator++ Algorithm and combining the results - Pranali, Rishitha
- Implementing Metis - Pranali
- Sequential BFS - Rishitha
- Parallel BFS - Siddhant , Ahana

VIII. Timeline

- Graph Partioning - 20^{th} April
- Finding the triangles on each partition - 21^{st} April
- Metis implementation for partioning - 23^{rd} April
- Sequential BFS - 26^{th} April
- Project Proposal - 26^{th} April
- Parallel BFS - 4^{th} May
- Comparitive study of all the techniques used for partitioning based on time and parallelism metrics. - 6^{th} May
- Final Report - 8^{th} May

IX. Implementation and Results

I. Algorithms Used

The overall implementation of the program is as per described in the above graph where we have a partioning phase, one processing phase and one merging phase where the triangles incident on all the nodes are found. The processing phase and the merging phase are common to all the implementations as given in the above code snippet (using Modified Node Iterator algorithm). However, because we are dealing with large graphs specifically, to reduce the latency in the partioning phase, we have tried a couple of algorithms for the partioning phase -

- Naive partioning - The graph is partitioned simple based on the node indices as given in the input graph. Though this is simple and easy to implement, it does not exploit the graph details to find out suitable partitions which would eventually improve the work of the processing phase.

- METIS partioning - Metis is a software package for partioning large graphs. Traditional partitioning algoeithms find the partitions by directly acting on the original graph. However, metis works in many phases to acheive good partitions. It reduces the size of the graph by collapsing the vertices and edges (coarsening phase), it then partitions these smaller graphs (initial phase) and then uncoarsens them to construct the partition of the original graph. The software has its in built algorithms which make the coarsening phase easy and novel. It is extremely fast and procudes high quality partitions.

- Sequential BFS (Breadth First Search) - Get the adjacency matrix of the graph and do a BFS on the graph. We then get distances of each node from the source node. Now the nodes at level "i" do not form any triangles with the nodes at level > "i+1" and level < "i-1". Exploting this, we can get rid of the nodeIterator3 function of the implementation. However we had expected the performance to improve because of the reduction in number of operations being done. But the skewed partitioning of the graph (which gives unequal load on the threads) reduces the performance largely. Advantage : Therefore we can reduce the number of operations being done in the processing phase. Disadvantage : Skewed partitioning, therefore uneven distribution of work among threads and reduced performance.

- Parallel BFS using reducer list data structures: A reducer_list is a concurrent list data structure which can be used by multiple functions in parallel without races and contention. The main PBFS function works as follows. For every layer in the BFS, we have a reducer_list of the nodes. For every parent layer, a new reducer_list corresponding to the child layer is created, which is filled up in parallel by the PROCESS-LAYER function. After this, the depth is incremented by one and the new set of children are set as the new parent. The above procedure is repeated for this new set, given that it is not empty – at which point the loop exits, and the PBFS is complete.
  The PROCESS-LAYER function takes a parent list and a child list which needs to be filled. If the size of the parent list is greater than the GRAINSIZE, the list is split and each part is processed recursively in parallel. When the parent list size is less than GRAINSIZE, each node in this list is processed in parallel, wherein, the children for each node are computed and inserted into the children-list. There are no races encountered here since the reducer_list data structure is being used. Finally, after all the spawned children have returned, the children-list is accumulated and the PROCESS-LAYER function returns, with the children-list filled.

II. Results

The results of various implementations are -
Seq BFS (on Facebook dataset)
Total number of triangles : 1.61201e+06
Avg clustering coefficient : 0.605547
elapsedTime: 4.27785
Number of Partitions : 6

For Parallel BFS (on Facebook dataset)
Total number of triangles : 1.61201e+06
Avg clustering coefficient : 0.605547
Elapsed time: 2.34941 sec
Number of Partitions : 6

For METIS
Total number of triangles : 1.61201e+06
Avg clustering coefficient : 0.605547
Elapsed time: 22.299 sec
Number of Partitions : 20

For Naive Implementation
Total number of triangles : 1.61201e+06
Avg clustering coefficient : 0.605547
Number of Partitions : 9
elapsedTime: 2.71821 sec
Number of Partitions : 5
elapsedTime: 1.5166 sec
Number of Partitions : 4
elapsedTime: 0.541072 sec

CilkView statistics for the implementation with Parallel BFS
1) Parallelism Profile
Work : 4,393,970,551 instructions
Span : 184,409 instructions
Burdened span : 16,994,409 instructions
Parallelism : 23827.31
Burdened parallelism : 258.55
Number of spawns/syncs: 16,314,635
Average instructions / strand : 89
Strands along span : 706
Average instructions / strand on span : 261
Total number of atomic instructions : 16,314,656
Frame count : 32,633,308
Entries to parallel region : 7
2) Speedup Estimate

2 processors: 1.90 - 2.00
4 processors: 3.80 - 4.00
8 processors: 7.60 - 8.00
16 processors: 14.56 - 16.00
32 processors: 26.58 - 32.00
64 processors: 45.25 - 64.00
128 processors: 69.75 - 128.00
256 processors: 95.64 - 256.00

X. Conclusion

From all the implementations we have come up with a conclusion that the naive implemention is better when the number of partitions is less. However, when the graph is huge and the partitioning phase is quite extensive, we have found that graph partitioning using Parallel BFS rules out the others and proves to be better among all. The parallelism obtained in the partition phase and the processing of the phase indeed reduces the time taken for the overall computation.

References

[S M Arifuzzaman, Maleq Khan and Madhav V. Marathe] Parallel Algorithms for Counting Triangles and Computing Clustering Coefficients. http://staff.vbi.vt.edu/maleq/papers/sc12-poster.pdf

[Charles E. Leiserson, Tao B. Schardl] A Work-EfiňĄcient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers) http://web.mit.edu/~neboat/www/papers/pbfs/pbfs-spaa10.pdf

[Siddharth Suri and Sergei Vassilvitskii] Counting Triangles and the Curse of the Last Reducer http://theory.stanford.edu/~sergei/papers/www11-triangles.pdf

[ ] Metis. http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview

[A. Pavan1, Kanat Tangwongan2, Srikanta Tirthapura1] Parallel and Distributed Triangle Counting on Graph Streams http://domino.watson.ibm.com/library/CYBERDIG.NSF/papers/9CAF168069AEB04285257B1F0054E742/$File/rc25352.pdf