COL380: Assignment 2

Viral Marketing

Ishita Chaudhary
2019CS10360
March 16, 2023

Introduction

We need to find groups such that each pair of friends in the group must have at least k common friends for some fixed values of k. The social network of Insta-book is represented as a graph where each vertex represents a person and each edge represents the friendship between the vertices connected by the edge.

$$G = (V, E) \text{ with } n = |V| \text{ vertices and } m = |E| \text{ edges}$$

In graph theory, a truss is a type of subgraph that is particularly well-connected. Specifically, a k-truss of graph G is a subgraph H in which each edge is part of at least k − 2 triangles within H. In other words, every edge in the k-truss is part of at least k − 2 triangles made up of edges (and vertices) of that truss.
Observe that in a (k+2) truss, every edge is part of at least k triangles, which means every pair of friends will have at least k common friends. Thus, our problem reduces to finding (k+2) truss for every fixed value of k.

Approach

We divide our end-to-end algorithm of finding the k-size groups from the given input into 4 key parts which are explained as follows -

1. **Reading the graph**
   First of all, all the ranks read the values of n, m, node start offsets and degrees of all the nodes. Then we divide the nodes equally among the ranks and each rank reads the adjacency list of its assigned partition of nodes. The ownership of any edge (u,v) is given to the node with lesser degree. In case of tie, the ownership is given to u if u < v or vice-versa. Observe that each rank knows the owner rank of all the nodes and thus also the owner of all the edges.

2. **Triangle Enumeration**
   Consider any node u on rank A and two edges (u,v) and (u,w). Now without loss of generality assume that node v on rank B is the possible owner of (v,w) edge (if it exists at all). Now, rank A sends a (u,v,w) query to rank B to check for the existence of (v,w) edge. If (v,w) edge exists, rank B increments the triangle count of edge (v,w) and sends back (u,v,w) triple to rank A which then increments the triangle count of edges (u,v) and (u,w). For every edge, we also store the list of nodes which make a triangle with that edge.
   The above queries and their subsequent responses are sent and received using the *MPI_Alltoallv* primitive because every rank might have queries for every other rank. We do this in batches so that individual *MPI_Alltoallv* calls are not very large. We use *MPI_Alltoall* primitive before every call to *MPI_Alltoallv*, to get the size of the receive buffer required.
   Let supp(e) be the number of triangles incident on edge e, we initialise $\tau(e) = supp(e) + 2$.

3. **Truss values computation**

We define τ(u,v,w) = min{τ(u,v), τ(u,w), τ(v,w)}, for every triangle (u,v,w). We use the proposition given in reference [1].

Proposition 1. *For any edge $e = \langle u, v \rangle$, we have that*

$$\tau(e) = \max\{j \ : \ |\{\Delta(u,v,x) \ : \ \tau(u,v,x) \geq j\}| \geq j-2\}$$

For each triangle, we maintain an upper bound on its truss value, initialising it to INT_MAX. To manage the load of computation, we maintain a window of values of k to be processed in the current iteration. We start with k_{min} and add the edges with the next value of k to the active set before the next iteration. In each iteration, we also consider all the edges whose truss value changed in the previous iteration.

We group the triangles incident on any edge e based on their truss values and maintain a histogram consisting of two components,$h_e(\cdot)$ and g_e . For $j < \tau(e)$, $h_e(j)$ stores the number of triangles with truss value exactly j, whereas g_e keeps track of the number of triangles with the truss values at least τ(e). We initialise $g_e = \tau(e) - 2$ and for all $j < \tau(e)$, $h_e(j) = 0$. This histogram strategy helps avoid expensive recomputations.

$$\forall j < \widehat{\tau}(e): \quad h_e(j) = |\{\Delta(u,v,x) : \widehat{\tau}(u,v,x) = j\}|$$

$$\text{and} \quad g_e = |\{\Delta(u,v,x) : \widehat{\tau}(u,v,x) \geq \widehat{\tau}(e)\}|$$

Consider any edge e in the current active set. For all triangles (u,v,w) incident on e, if the current truss number for the edge is less than that of the triangle, we update τ(u,v,w) = τ(e), and also send the new updated value of τ(u,v,w) along with its identity (u,v,w) to the other two edges (say e' and e'') of the triangle so that they can update their truss values to satisfy proposition 1. $h_e(\cdot)$ and g_e values are also updated for the other two edges, depending upon the old and new values for τ(u,v,w). When $g_{e'}$ falls below τ(e) − 2, we decrement τ(e) as well and update $g_{e'}$ using $h_{e'}(\cdot)$.

MPI_Alltoallv primitive is used to send the new updated truss values of triangles to the other two edges. We use *MPI_Alltoall* primitive before every call to *MPI_Alltoallv*, to get the size of the receive buffer required.

We end the algorithm when the size of the active set for all the ranks becomes 0. This is checked using the *MPI_Allreduce* primitive at the end of every iteration where each rank sends its size of the active set for the next iteration.

4. **Outputting the results**

For verbose = 0, every rank sends its maximum truss value among all the edges owned by it to the process with rank 0. Rank 0 then finds the maximum truss value over the entire graph, by taking the maximum of the maximums. We know that if k-truss exists for some k = K, then k-truss will exist for all k<K. Existence of k-truss implies the existence of (k-2)-group.

For verbose = 1, every rank sends the edges involved in at least one triangle along with their corresponding final truss values, to rank 0. Rank 0 then uses Depth First Search algorithm to find k-size groups for all $k \in [k_1, k_2]$.

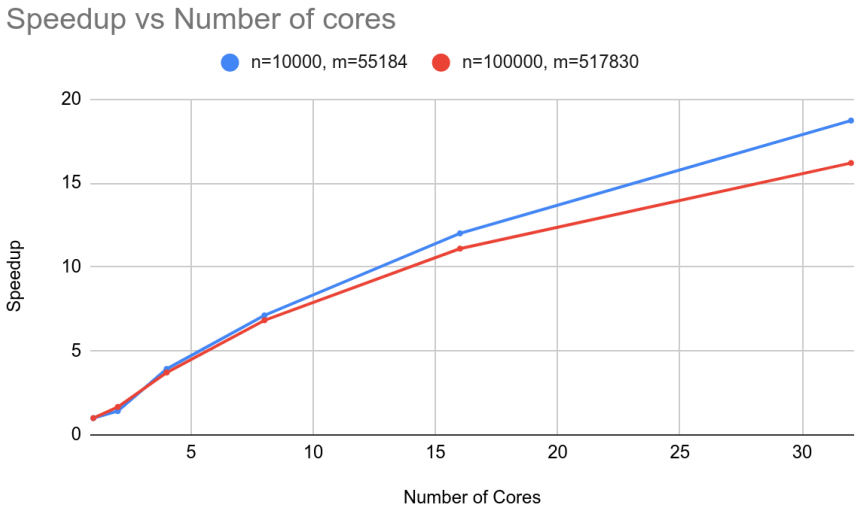
Rank 0 then writes the output to the *outputpath* text file depending upon the value of verbose.

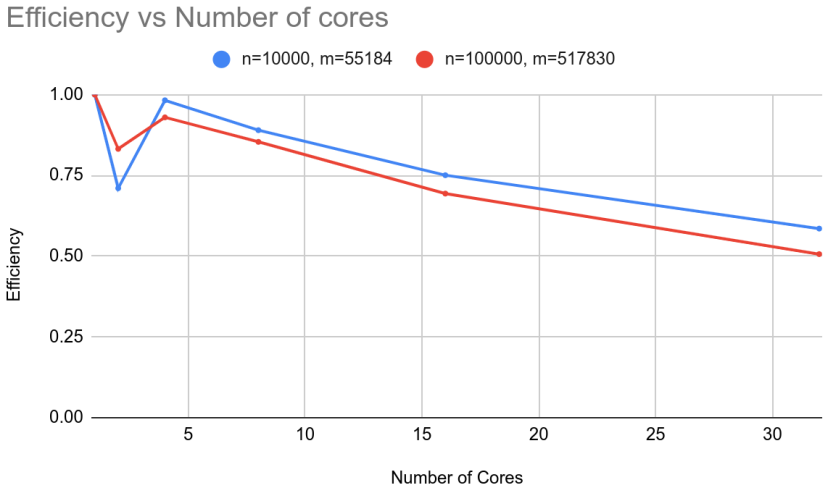
Results

Given below are the runtimes in milliseconds(ms) for graphs of different sizes and densities (Verbose = 0)

Cores / Testcase	n=100 m=1489	n=1000 m=12481	n=25000 m=628968	n=10000 m=55184	n=5000 m=76579	n=100000 m=517830
1	1925	807	15348	91043	469677	106737
2	1249	634	9092	64088	287287	64137
4	879	539	4370	23165	148064	28701
8	709	507	2663	12787	73656	15626
16	606	510	1759	7581	46328	9616
32	633	545	1405	4861	30355	6588
Maximum Speedup	3.176567657	1.591715976	10.92384342	18.72927381	15.47280514	16.20173042
Corresponding Efficiency	0.1985354785	0.198964497	0.3413701068	0.5852898066	0.4835251606	0.5063040756

Let us consider two of the large test cases in more detail.





Observations

- We observe that the algorithm is scaling well for large input sizes where the runtimes with 32 cores are significantly less than that for a single core.We observe upto 18X speedup in one of the large test cases.
- For large inputs, the runtimes decrease monotonically as we increase the number of cores which is as expected. For smaller inputs, we observe that the runtimes saturate as we increase the number of cores because the cost of synchronisation balances out the benefits of parallelization.
- While the speedup is fairly good with 32 cores, the efficiency values are in the range of 50-60% for large test cases, which is also not bad. The efficiency with 16 cores is about 70% and with 8 cores, the efficiency is as high as 88% in one of the large test cases with 7X speedup which is a fairly good tradeoff.
- Another interesting observation is that the runtime for some of the test cases with smaller input size is more than those with larger inputs. This might be due to differences in the density of the graphs and the distribution of dense parts of the graph among the ranks since we have done the distribution uniformly based on the nodes and not taken the density of edges into account.

References

[1] Chakaravarthy, V.T., Goyal, A., Murali, P., Pandian, S.S., Sabharwal, Y. (2018). *Improved Distributed Algorithm for Graph Truss Decomposition*. In: Aldinucci, M., Padovani, L., Torquati, M. (eds) Euro-Par 2018: Parallel Processing. Euro-Par 2018

[2] R. Pearce and G. Sanders, "*K-truss decomposition for Scale-Free Graphs at Scale in Distributed Memory*," 2018 IEEE High Performance extreme Computing Conference (HPEC), Waltham, MA, USA, 2018, pp. 1-6, doi: 10.1109/HPEC.2018.8547572.

[3] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini and G. Karypis, "*Truss decomposition on shared-memory parallel systems*" 2017 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2017, pp. 1-6, doi: 10.1109/HPEC.2017.8091049

[4] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. *Parallel triangle counting and k-truss identification using graph-centric methods*, In High Performance Extreme Computing Conference (HPEC), 2017 IEEE, pages 1–7. IEEE, 2017.