COL380: Assignment 3

Again Viral Marketing

Ishita Chaudhary
2019CS10360
April 7, 2023

Detailed Approach for Task 1

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 τ(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 < τ(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 = τ(e) − 2 and for all j < τ(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 ∈ [k_1, k_2].

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

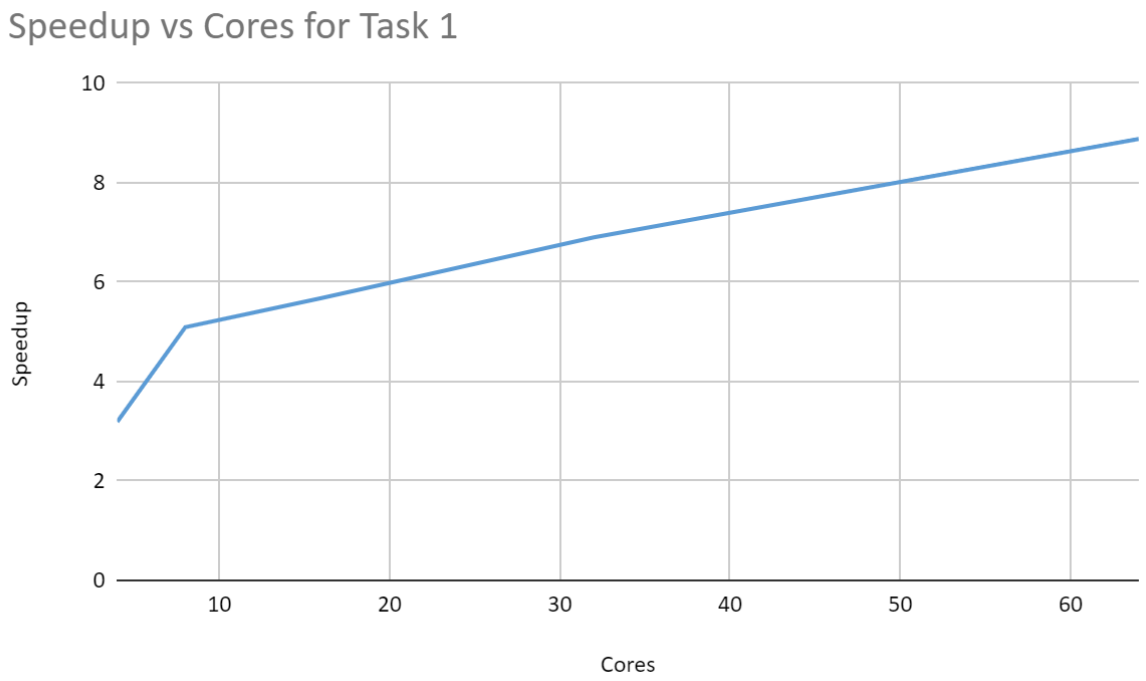
OpenMP tasks were used at certain places in the above algorithm to exploit threads available with every MPI process.

Plot Graph for speedup and efficiency for different core counts

We used the following formulas to calculate speedup and effeciency respectively:

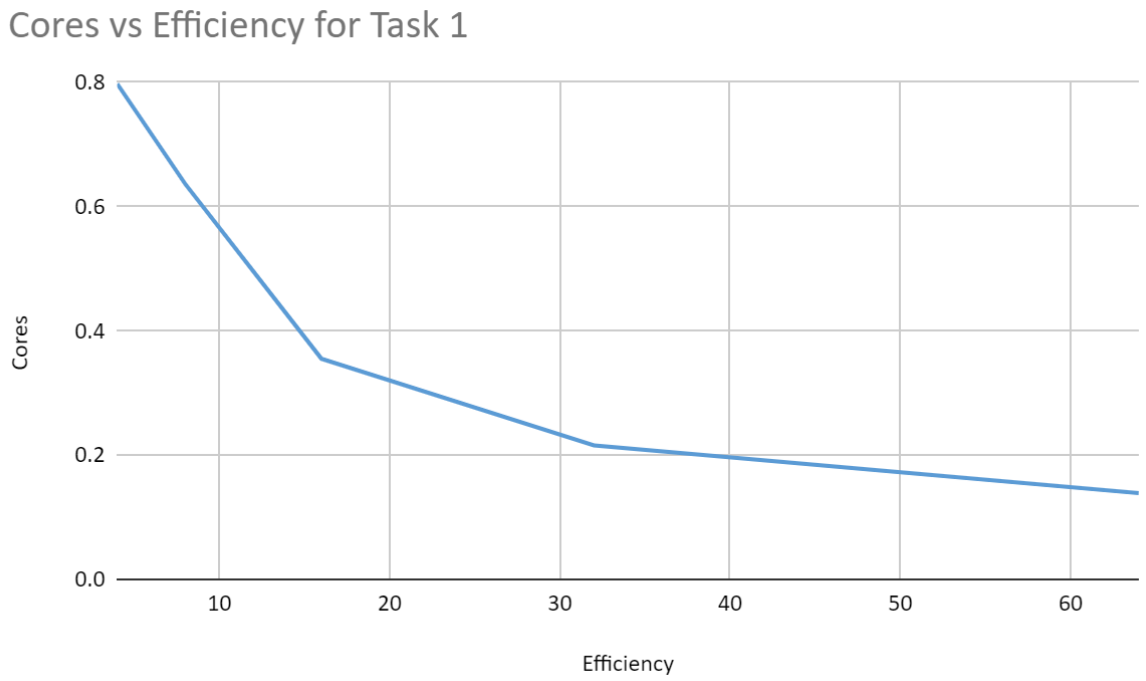
Speedup = $t_1(n, 1)/ t(n,p)$

where n denotes the problem size, $t_1(n, 1)$ denotes the time taken by the program using 1 processor and p denotes the number of processors, which is equal to (number of nodes x number of threads).



Efficiency (ε) = Speedup / p

this is simply the speedup acheived per processor.



Report Iso - efficiency and explain

Iso-efficiency is a metric to determine how a problem size should grow in order to maintain the effeciency. We know that:

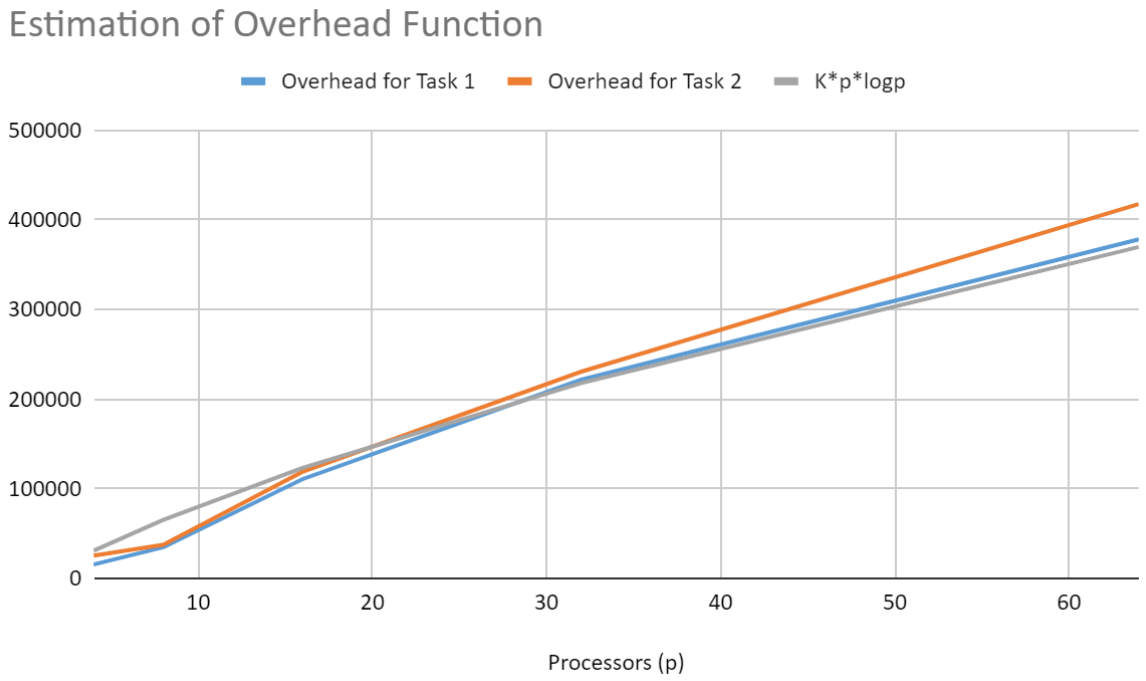
$$t_1(n,1) = \varepsilon(n, p)*p*t(n, p) = \text{Isoeffeciency } (I(p))$$

where ε(n, p) denotes efficiency of a program with p processors and n problem size.

Now, consider the parallelization overhead, the extra time needed in computation which is not needed in a sequential program, defined as:

$$\bar{o}(n, p) = t(n, p)*p - t_1(n, 1)$$

We can write I(p) as ε/(1-ε)*ō(n, p). SInce efficiency is constant, I(p)= K*ō(n, p). We estimated the overhead using the plot fitting method, and p*logp was the closest estimate. Hence, our I(p)= K*p*logp, where p is the number of processors.



Estimate sequential fraction for 8 -64 cores

We use Gustaffson's Law to determine the sequential fraction, f. Consider the following equation:

$$t(n, p)= t_{seq} + t_{par}$$

$$t(n, p)= ft(n, p) + (1-f)t(n,p)$$

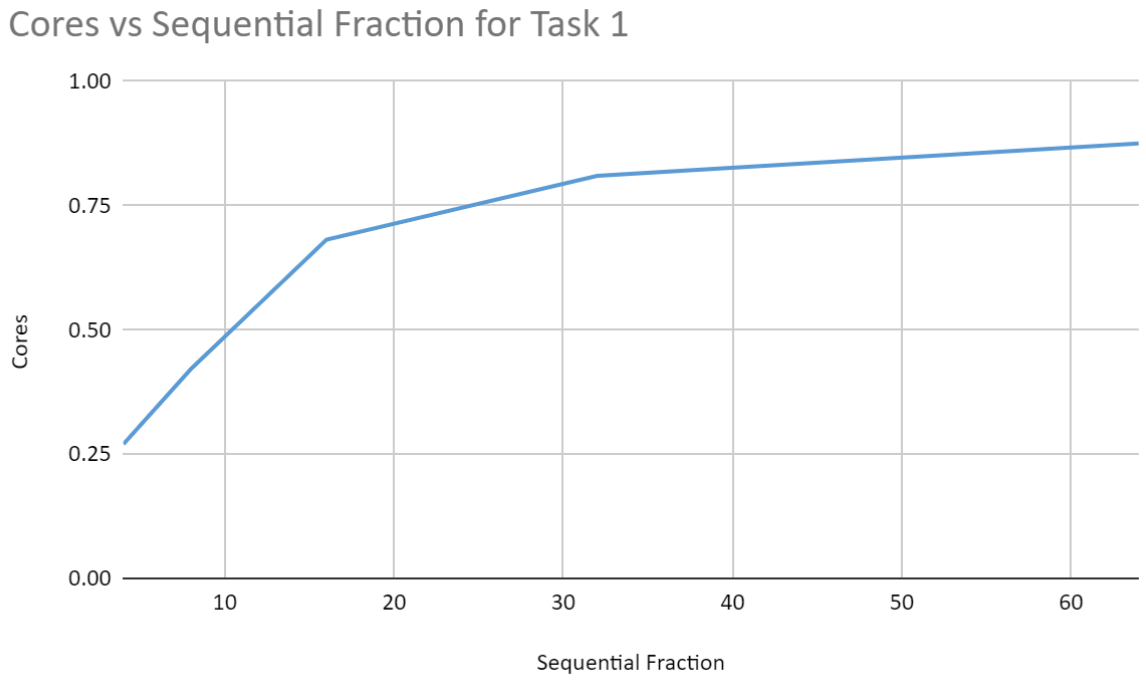
Now,

$$t_1(n,1)=ft(n, p) + p(1-f)t(n,p)$$

$$\text{Speedup}=f+(1-p)f$$

Rearrange the equation to get:

$$f= p-\text{Speedup}/(p-1)$$



Justify why your solution for task 1 is scalable

This solution for task1 is quite scalable because we have used MPI processes to divide the graph and perform subsequent calculations in parallel. Moreover, we have used OpenMP tasks to exploit parallelism at each node. The scalability is also verified by the metrics reported in *metrics.csv* .

We have divided the nodes of the graph equally among the MPI ranks and also assigned ownership of every edge to a specific rank, and hence, we can store much larger graphs which could otherwise not be processed by a single process, making the program highly scalable. OpenMP threads further help to speedup execution at each rank by using shared memory for running tasks in parallel.

Detailed Approach for Task 2

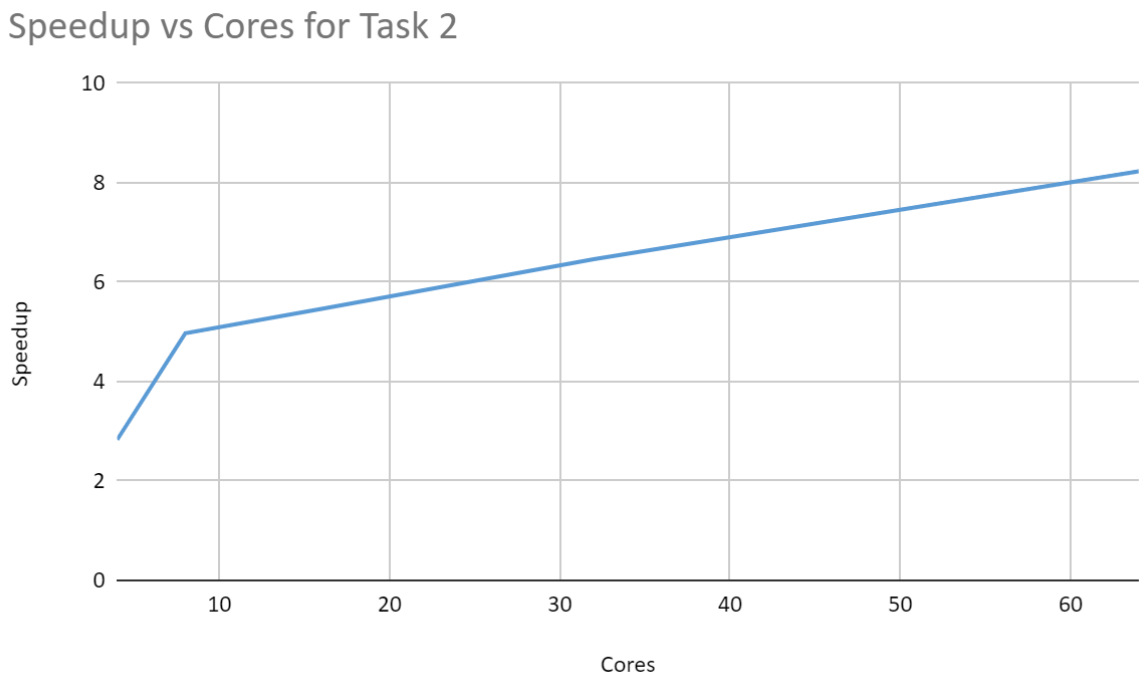
For Task 2, the first three steps, i.e. Reading the graph, Triangle enumeration and Truss values computation remain the same as that for Task 1 because we need all k-advertisement groups to find influencer nodes and their respective ego-network.

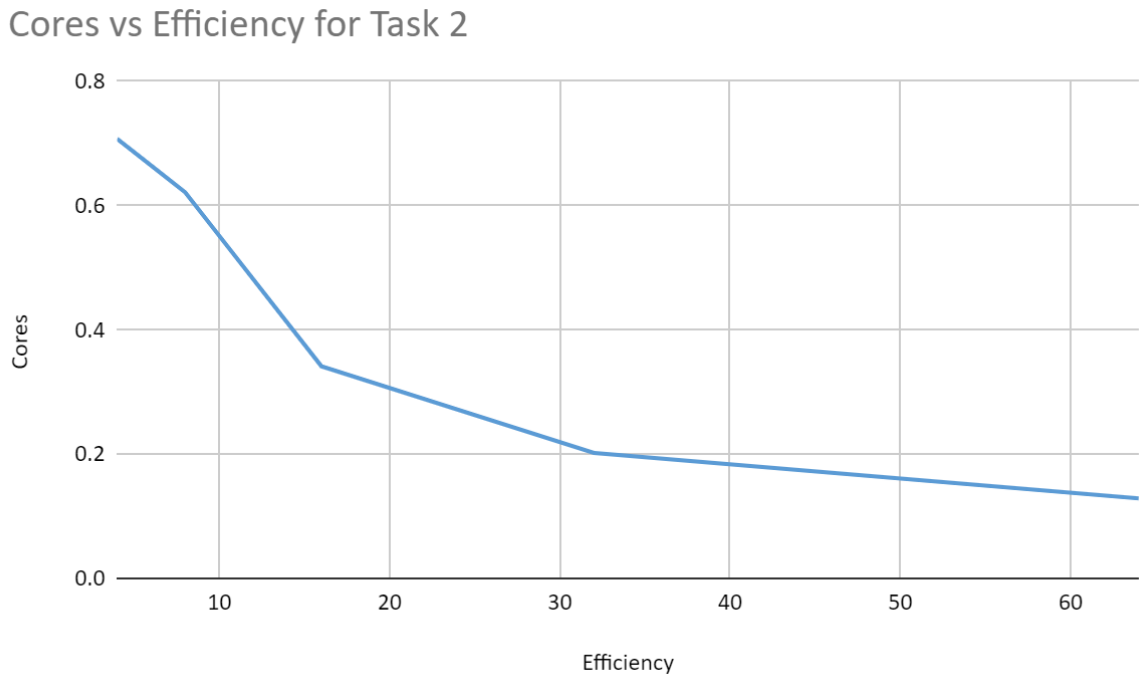
In the last step, every rank sends the edges involved in at least one triangle along with their corresponding final truss values, to rank 0. This involves the use of *MPI_Gather* and *MPI_Gatherv* primitives. Rank 0 then uses the Depth First Search algorithm to find k-advertisement groups for k = endk. Rank 0 also assigns a unique component number to all the vertices and then broadcasts this information to all the ranks using *MPI_Bcast* primitive.

Now every rank goes through the adjacency lists of the vertices that it owns and finds the number of unique components that the neighbours of each vertex belong to. If this number is greater than or equal to *p*, we have found an influencer vertex. Then each rank sends the

relevant information about the found influencer vertices to rank 0 depending upon the verbose level.
Rank 0 then writes the relevant output to the *outputpath* text file.
OpenMP tasks were used at certain places in the above algorithm to exploit threads available with every MPI process.

Plot Graph for speedup and efficiency for different core counts





Justify why your solution for task 2 is scalable

Most of the steps in task2 are similar to that of task1 and hence already have the benefits of shared-memory (OpenMP-based) and message-passing (MPI-based) programming.
For identifying influencers after finding k-advertisement groups, every rank can iterate through its partition of nodes in parallel to find the influencer nodes and then send the necessary information, depending upon the level of verbose, to rank 0 for output. The partition of nodes at each rank can be further partitioned into threads for faster execution. This makes the program highly parallel and fairly scalable.

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.