COL380: Parallel and Distributed Programs
Assignment 2: Again Viral Marketing

Kartik Sharma
2020CS10351

Chinmay Mittal
2020CS10336

Task 1

Approach

Dividing the Graph

We first divide the graph between the processes, we do this by *dividing the nodes* amongst the processes.
We create almost equal continuous-sized chunks, and each process gets one chunk of nodes for which it is responsible.
We also *divide the edges*, consider an edge A ↔ B, and if A are B are nodes assigned to different processes, we make this edge a directed edge and assign it to only one process. This prevents duplicate work, and only one process processes this edge.
We first assigned edge A ↔ B to the process which is responsible for the smaller node. This is ineffective since the first process is responsible for all the smaller nodes and hence will be accountable for a lot of edges; on the other hand, the last process is responsible for all the larger nodes and will be assigned very few edges.
We used a *heuristic (degree ordering)* where we assigned the edge to the process responsible for the node with a smaller degree (deciding ties by the node value). This effectively does load balancing where all processes will be responsible for an almost equal number of edges.

Reading the Graph

We *read the graph parallelly*, using *MPI parallel File Handlers* where every process only reads the part of the graph for which it is responsible (as discussed above). We implemented this using the header file given to us using which we find the offsets in the main file to read only the adjacency lists for which we are responsible. Each process thus stores the adjacency list of the nodes for which it is accountable in its local graph object. This is thus *memory efficient*.

Memory Efficiency

We have parallelly distributed the graph to all the processes, and no process stores the entire graph within itself. This helps in saving a lot of extra space for each process in our distributed algorithm. We also *never do triangle storage*; hence the space complexity of our parallel program is O(n+m), which does not scale with the number of processes.

Parallel Prefilter

Each process is responsible for deciding whether to prefilter the nodes assigned to it. The algorithm essentially remains similar just that it is parallelized.
Each process knows the degree of its vertices and can decide which nodes need to be prefiltered.
We then use the *MPI_Allgatherv* collective to synchronize amongst the processes all nodes that are deleted. Now each process knows which are the global nodes that are deleted. It uses this to update the adjacency list of the nodes for which it is responsible. This might potentially lead to more node deletes, this continues happening till all processes delete no nodes.

Deleting Edges parallelly

Now each process maintains correct information about its nodes and their neighbors.

Figuring out which edges to delete was a challenge for us. Using perf tools, we figured out that the *find_common_neighbour* is a bottleneck in our algorithm. So we used openMP threads to parallelize these computations for different edges which are to be deleted. This significantly boosted our performance from our previous best. In order to ensure that 2 different threads don't process 2 edges sharing the same vertex, we gave *locks* for each vertex at each node. Whenever an edge was being processed, the thread held locks for the vertices in a critical region to parallelize the execution. We tried to parallelize every part except those which had to be done sequentially.

We store the support count of all the edges we are responsible for; this is initialized once and maintained throughout *(one edge is only processed and maintained by 1 process)*.
Using the support count, we know which edges are to be deleted. This information is to be sent to all processes. We use *MPI_Allgatherv* which is a collective synchronization step. One thread is responsible for the communication and all threads process the deleted edges parallely. As explained above we must use locks for each vertex to ensure that two edges of the same vertex are not processed parallely potentially leading to *race conditions*.

We get all edges that are deleted in the graph. Note that this is not particularly ineffective since each edge is only deleted at most once from the graph, also we discard an edge if it does not affect the support count of the edges we are responsible for maintaining.
We use the information on deleted edges to update the part of the graph we are responsible for and also update the support count of the edges we are responsible for.
Some more edges might get deleted since their support decreases.
This way we keep iterating, and the stopping criterion is that no edges are deleted in any process.

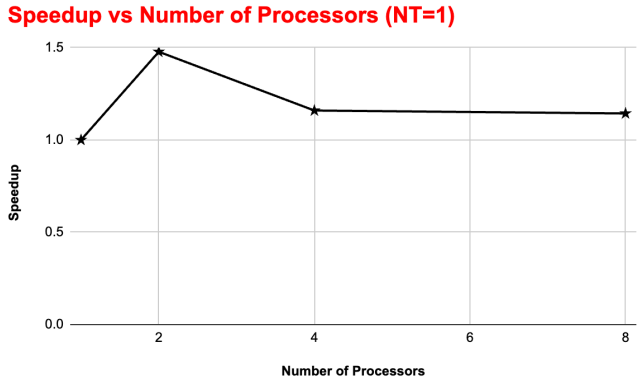
Getting The Output

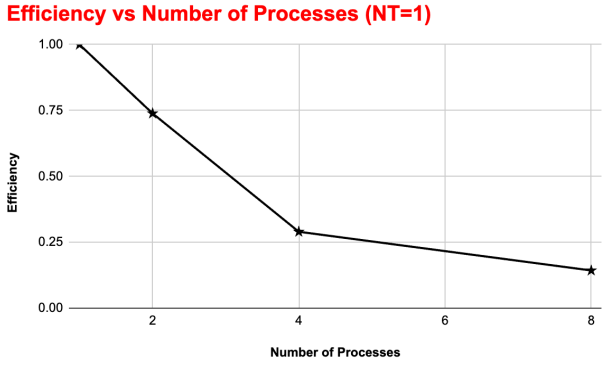
If the verbose is 0, we only need to check if a k truss exists. We query this by checking if there exists any edge with any process. We don't need to get the graph together in this case.
Thus verbose 0 is handled more efficiently.

When the verbose is 1, We use *parallel DSU* to calculate the connected components of the graph. Each process calculates its locally connected component graph, thus getting an array of size equal to the number of nodes. The graphs then share this information in a tree structure, thus requiring log(n) send and receive. The merged information gives a global notion of graph connectivity in the form of a parent array. We also used size as rank while doing DSU, which helped us reduce the time complexity of the *union* operations and determine the size of the connected component in O(1) using the *path compression and union by rank algorithm*. A single process then writes the connected components in the desired format.

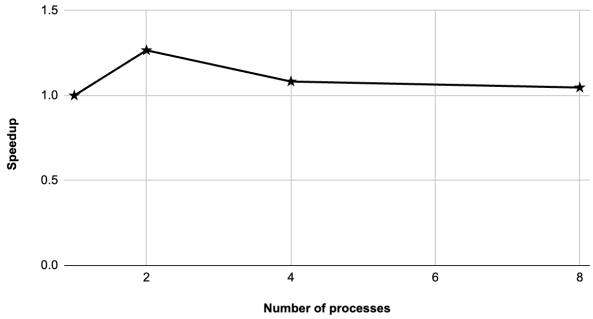
Graphs for speedup and efficiency for different core counts

Test case 3

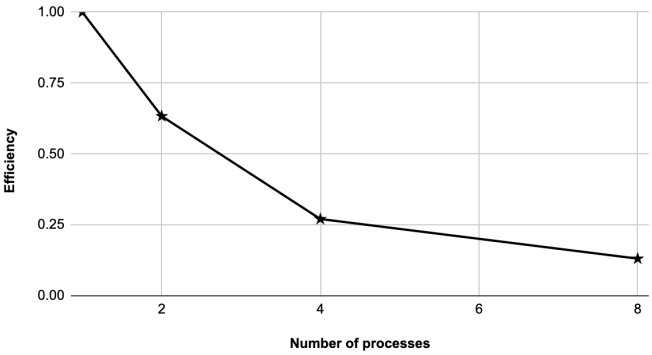




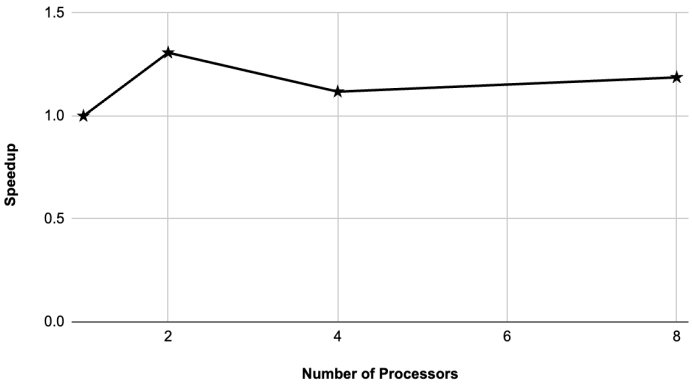
Speedup vs Number of processes (NT=2)



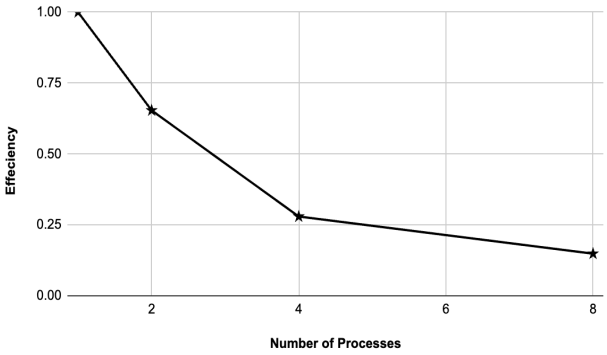
Efficiency vs Number of Processes (NT=2)



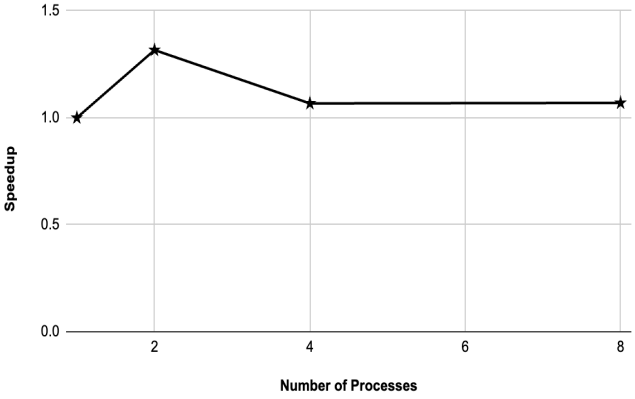
Speedup vs Number of Processors (NT=4)



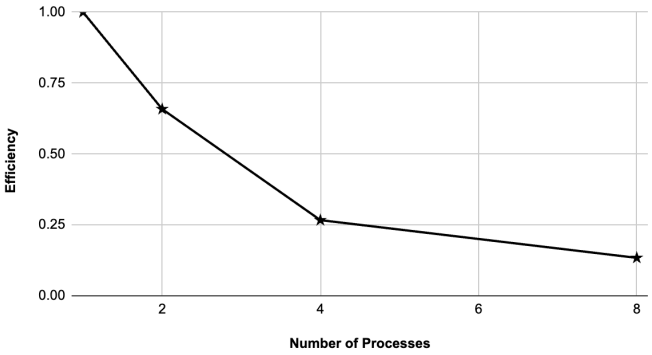
Efficiency vs Number of Processes (NT=4)



Speedup vs Number of Processes (NT=8)



Efficiency vs Number of Processes



Iso-efficiency

We estimate out problem size to be $O(n^3 \log n)$ in the sequential case. Also interpretting the above graphs, we see that the speed up is more or less linear of the form ($S_p = a*p + b$). Using this we calculate the $t(n, p) = t(n, 1) / S_p$.
Thus $t(n, p) = t(n, 1)/(a*p + b)$. Now we calculate the overhead function, i.e., $t(n, p)*p - t(n, 1)$.
$o(n, p) = n^3 \log n \, (p(1-a) + b/(ap + b))$
Now we know that iso efficiency funtion is $\mathcal{F}(p) = \Omega(o(p))$
Therefore $\mathcal{F}(p) = \Theta(p)$

Sequential fraction

$$Sp = \frac{t1}{tp}$$

We calculate the sequential fraction using the amdahal's law, which states:

$$Sp = \frac{1}{(f + \frac{1-f}{p})}$$

Where Sp is the speed up obtained on using p processes and f is the sequential part of the code. We calculated *f* as:

$$f = \frac{(\frac{1}{Sp} - \frac{1}{p})}{(1 - \frac{1}{p})}$$

Method 1

We fix the number of threads and vary the number of processors to compute the sequential fraction. The reported sequential fraction is for number of threads 1.

Method 2

In our case since we are changing both the number of processes and the number of threads. We compute the speedup by comparing to the time when both the number of processes and number of threads are one. In these formulats p = (Number of Processes * Number of Threads).
For a given test case and a given task. We compute f for different values of p and then take the average to estimate the sequential fraction.

Task 1

Test Case	Estimate of Sequential	Estimate of Sequential

	Fraction (Method 1)	Fraction (Method 2)
1	0.46	0.62
2	1.00	1.01
3	0.67	0.75

Some Observations Regarding Estimating Sequential Fraction

For extremely small test cases the problem is inherently sequential (test case 2). Larger test cases have more scope of parallelism and hence have lower sequential fraction.

Why is our solution scalable?

For large graphs, reading the graph can be a bottleneck, which we have paralelized using MPI file handlers. As the number of processes increases, each process is responsible for fewer vertices and fewer edges. We have used degree ordering of edges to balance the load between different processes. Each edge or vertex has been assigned to a unique process and stored to at most 2 processes. This makes our implementation memory efficient as memory requirements don't scale with the number of processes. We have used parallel disjoint set union (DSU). Hence we don't need to share the graph among the entire graph to compute the k-truss groups. This keeps the communication overhead low as we increase the number of processes. Each edge is communicated at most once among the processes. We used perf tools to identify the bottle-neck function i.e.,"find_common_neighbour" and parallelized that part. We also prevented deadlocks by acquiring locks and critical sections.

Task 2

Approach

Computing k-truss

We used the same above-mentioned parallel and distributed algorithm for reading the input and computation of the k-truss from the input graph.

Finding Influencers Vertices

Now, each node processes its vertices for influencer vertex check. For every vertex, we iterate over all its neighbors and store the representative vertex of their connected component (DSU-find) in a set. If the size of the final set is greater than p, we select it as an influencer vertex. These operations have been parallelized by openMP. For a given vertex all its

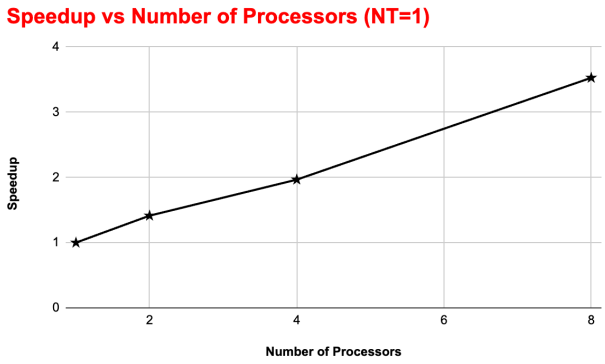
neighbours are processed parallely. Also storing the representative vertices in the DSU ensures that finding k-trusses is efficient (O(1) with DSU optimizations). Our algorithm is memory efficient since we never share information apart from the DSU array amongst nodes. For the reasons mentioned above our algorithm is also time efficient.

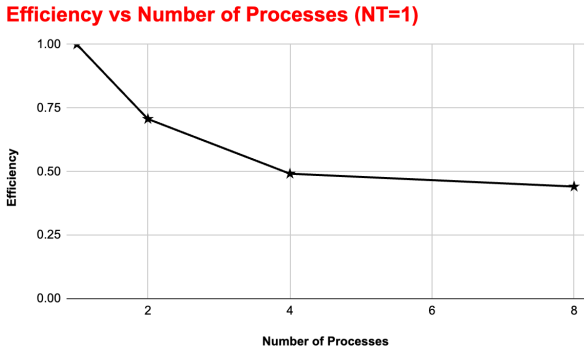
Getting The Output

Starting from node 0, we pass a token to each node allowing it to write its influencers and other group members (if verbose set). After finishing with its output, it passes the token to the next process until all processes have been covered. This way we don't have to share the influencer information between processes.

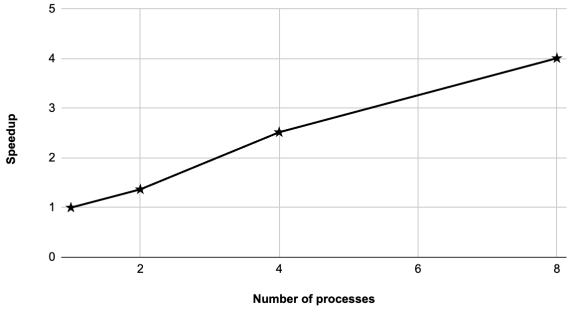
Graphs for speedup and efficiency for different core counts

Test Case 3

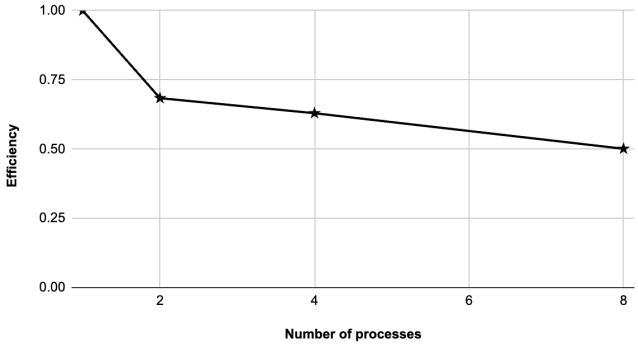




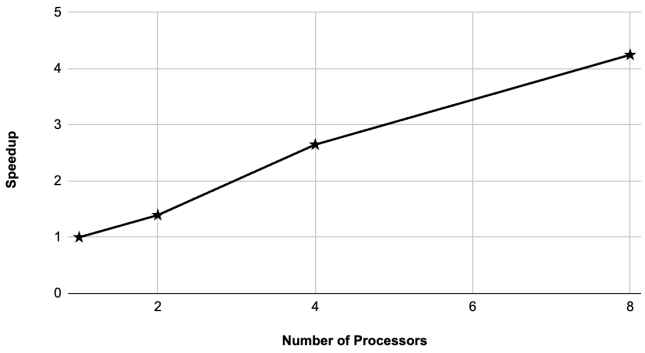
Speedup vs Number of processes (NT=2)



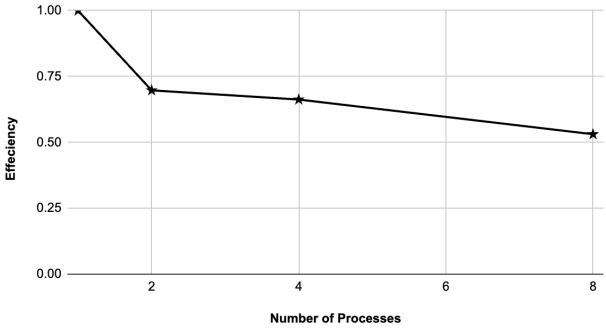
Efficiency vs Number of Processes (NT=2)



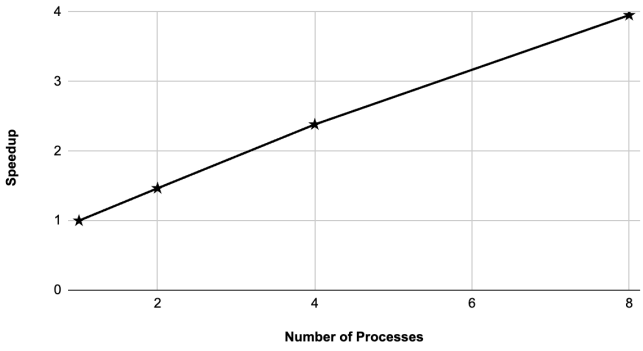
Speedup vs Number of Processors (NT=4)



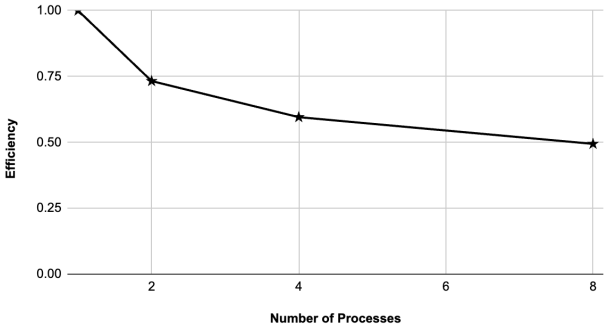
Effieciency vs Number of Processes (NT=4)



Speedup vs Number of Processes (NT=8)



Efficiency vs Number of Processes (NT=8)



Iso-efficiency

We estimate out problem size to be $O(n^3 \log n)$ in the sequential case. Also interpretting the above graphs, we see that the speed up is more or less linear of the form ($S_p = a*p + b$). Using this we calculate the $t(n, p) = t(n, 1) / S_p$.
Thus $t(n, p) = t(n, 1)/(a*p + b)$. Now we calculate the overhead function, i.e., $t(n, p)*p - t(n, 1)$.
$o(n, p) = n^3 \log n \, (p(1-a) + b/(ap + b))$
Now we know that iso efficiency funtion is $\mathcal{F}(p) = \Omega(o(p))$
Therefore $\mathcal{F}(p) = \Theta(p)$

Sequential fraction

Details mentioned above.

Task 2

Test Case	Estimate of Sequential Fraction (Method 1)	Estimate of Sequential Fraction (Method 2)
1	0.19	0.446
2	0.2	0.418
3	0.31	0.52
6	0.1	0.36

Some Observations Regarding Estimating Sequential Fraction

Task 2 is inherently more parallelizable than task 1 since computing influencer vertices can effectively be parallelized without any synchronization overheads unlink k-truss computation. Computing Influencer vertices is an expensive operations and hence task 2 becomes more parallelizable (smaller sequential fraction) than task 1.

Why is our solution scalable?

The k-truss computation is scalable as discussed above. The computation of the influencer vertices is also distributed among processes. Each process having nearly equal number of vertices. Using DSU, we could find the groups and their sizes for all neighbours in O(1) operations, without having to dfs again and again. For each vertex, the processing of all its neighbours is parallelized using openmp. Only the file output is sequential by token passing

among different processes. This make our algorithm scalable for larger number of nodes and cores.

We have reported the csv file for test case 3.
Details of other test cases can be found here:
https://docs.google.com/spreadsheets/d/1mmKMOF854hyk6HUGprcC4HIbPFzxyH85EDX5oV-RN0M/edit?usp=sharing

---

References

1. https://www.openmp.org/resources/refguides/
2. https://www.open-mpi.org/doc/
3. https://chat.openai.com/chat
4. https://ieeexplore.ieee.org/document/8091049/;jsessionid=Lpxaocz9pjUk3uoHEFToiAwE1S9Vqm0xzncrNzyF0IVRXKZ_8iYa!-1842855163
5. https://www.cs.utexas.edu/~yishanlu/papers/HPEC_2017.pdf
6. https://ieeexplore.ieee.org/document/8547572
7. https://link.springer.com/chapter/10.1007/978-3-319-96983-1_50
