COL380
Assignment 2

Kushagra Rode(2020CS10354) Shreyansh Singh(2020CS10385)

March 15, 2023

Our Approach

We'll describe both approaches first the sequential one(which we wrote as an initial step before parallelizing the code) and then we'll talk about how we have made the algorithm to run parallelly. In both cases, the graph is stored as an adjacency list so that the total memory usage would be O(V + E).

Sequential

The algorithm we have used is a modified version of the naive implementation. We have made use of filtering vertices as well as edges which do not contribute to k-truss finding. For filtering out vertices, we have the prefilter function which removes all those vertices which have a degree less than k-1. We then look at all the neighbors of this vertex and then proceed in a breadth-first search fashion.
For filtering the edges we have the Filteredges function. Now, this function initially computes the edges which have low support. This is done by our intersection function described in our code. So these edges would be the deletable edges. Now again we proceed in a breadth-first-search fashion while deleting the edge in deletable which is being considered. Next, we find the common neighbors of the endpoints of our edge. These 3 points form a triangle and if we remove the edge, the support of the other two edges should be decremented. If their support becomes less than k-2, we add this edge to our deletable edge.
After this we have our graph which contains all the k-trusses as connected components. Now for getting all the k-trusses, we use breadth-first search. Finally, we collect all these k-trusses and output them.

Parallelising our code

The main computation of the k-trusses takes place in the Filteredges part. Hence it makes sense to parallelize this section so as to improve efficiency. Say we have n processes. Each process takes care of vertices in a round-robin fashion, i.e process 0 has vertex 0, vertex n, vertex 2n, and so on (where n is the number of processors). Each process computes its own deletable edges. And then we do an initial AllGather so that each process gets to know the edges deleted by other processes. Now since this deletable list could be of very large length, hence we divide this list into chunks of size 50 at every process and send this through Allgather. Also, since Allgather expects the sizes of send buffers same, we do an Allreduce with MPI_MAX so as to get the maximum size of the deletable present at each process. Now each process does its own computation as described in the sequential section. At each iteration, each process has its local deletable, which again needs to be sent through Allgather as described above.
Finally, we get the graph with all the k-trusses. Next, we use breadth-first search to get all the k-trusses. Note that for getting the k-truss we make use of the graph which got modified while finding the k-1 truss, hence making it efficient.

Graphs of Time Taken for different graph sizes

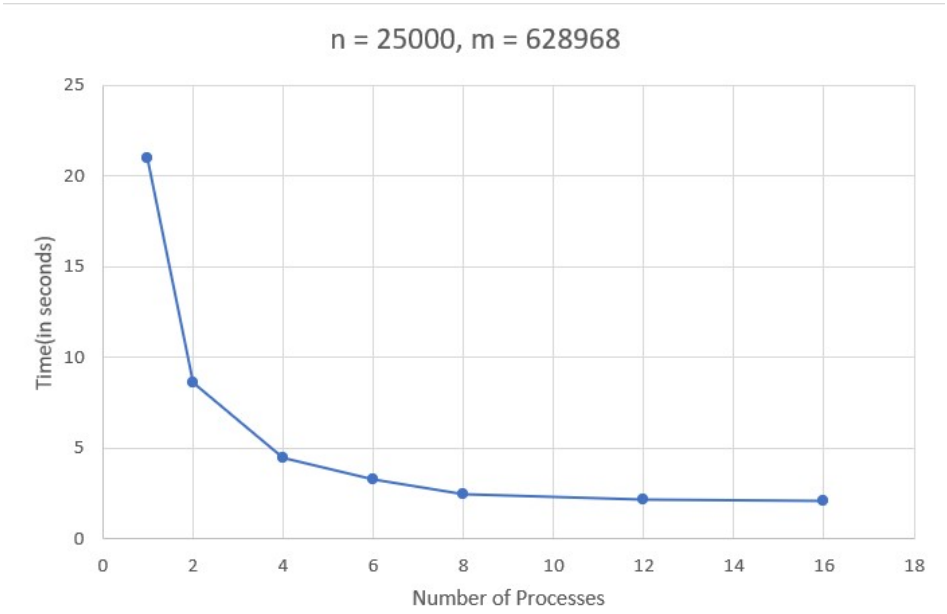

Figure 1: Time vs Number of processes for test 5

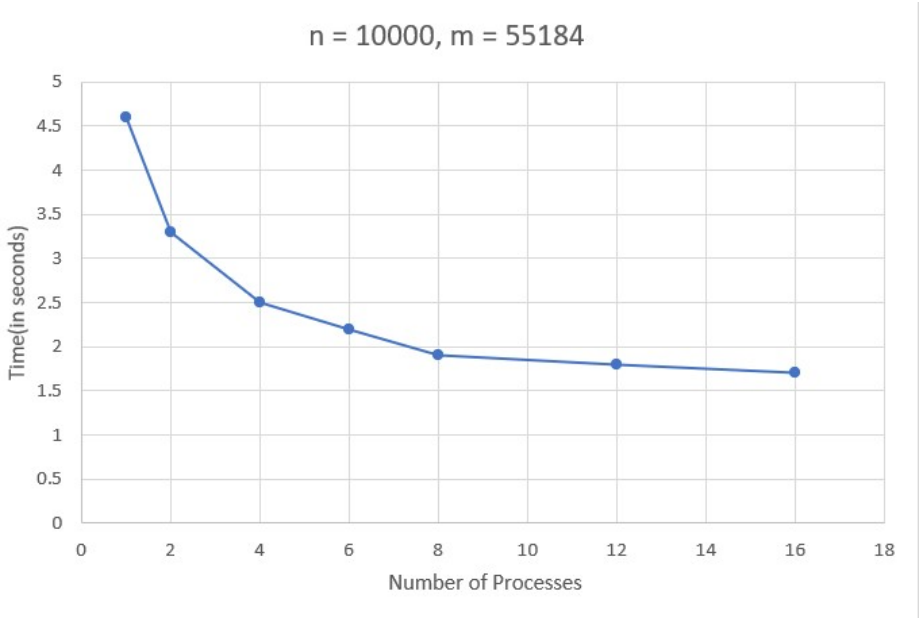

Figure 2: Time vs Number of processes for test 6

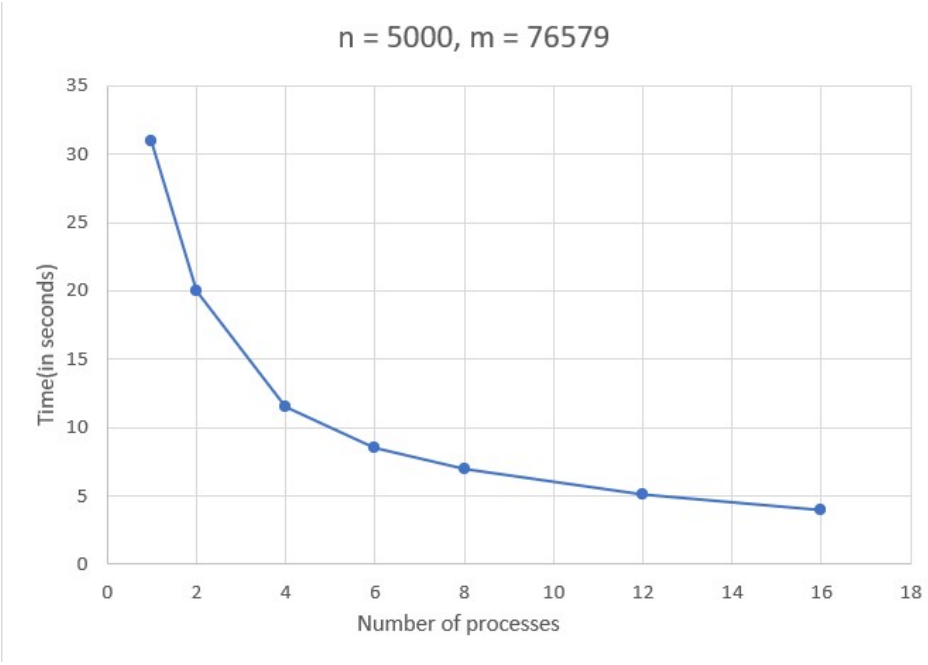

Figure 3: Time vs Number of processes for test 7

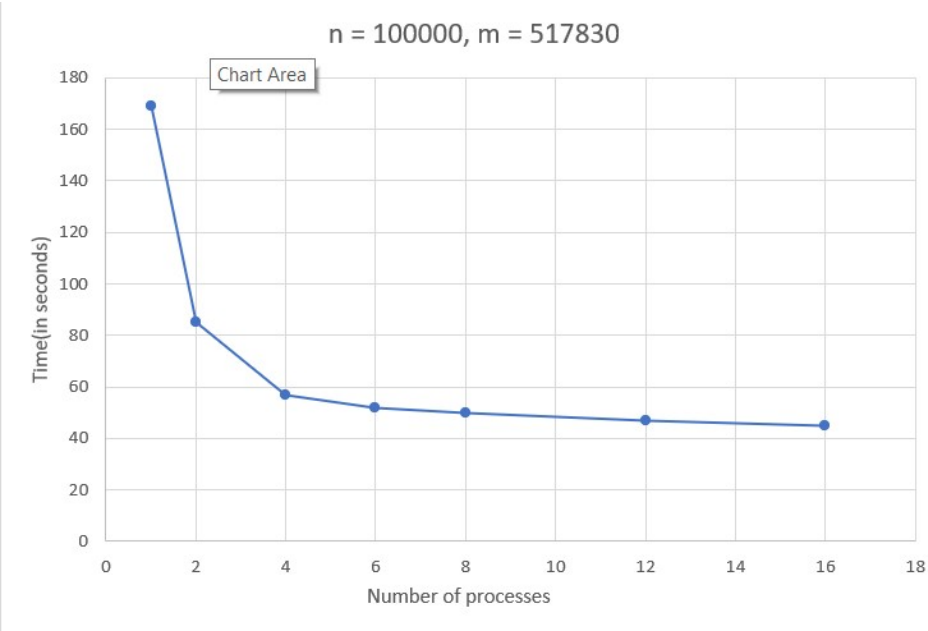

Figure 4: Time vs Number of processes for test 8

Table for the times taken by different test cases

All times are mentioned in seconds

Testcase	1	2	4	6	8	12	16
Test 5	15	8.6	4.5	3.3	2.5	2.2	2.1
Test 6	4.6	3.3	2.5	2.2	1.9	1.8	1.7
Test 7	31	20	11.5	8.5	7	5.1	4
Test 8	169	85	57	52	50	47	45

Figure 5: Time taken by various test cases

Speed-Up and Efficiency

Testcase	1	2	4	6	8	12	16
Test 5	1	1.74	3.33	4.54	6	6.81	7.14
Test 6	1	1.39	1.84	2.09	2.42	2.55	2.70
Test 7	1	1.55	2.69	3.64	4.42	6.07	7.75
Test 8	1	1.88	2.80	3.07	3.2	3.40	3.55

Figure 6: Speedup for various test cases

$$Speedup\ S_p = \frac{t_1}{t_p}$$

where t1 = time taken by 1 process and t_p is time taken by p processes

Testcase	1	2	4	6	8	12	16
Test 5	1	0.87	0.83	0.75	0.56	0.55	0.44
Test 6	1	0.69	0.46	0.34	0.30	0.21	0.16
Test 7	1	0.77	0.67	0.60	0.55	0.5	0.48
Test 8	1	0.94	0.7	0.51	0.4	0.28	0.22

Figure 7: Efficiency for various test cases

$$Efficiency \;\; E_p = \frac{S_p}{p}$$

where S_p is the speedup which we calculated above and p is the number of processes.

Legend for the graphs below -
Blue Line - Test 5
Orange Line - Test 6
Grey Line- Test 7
Yellow Line - Test 8

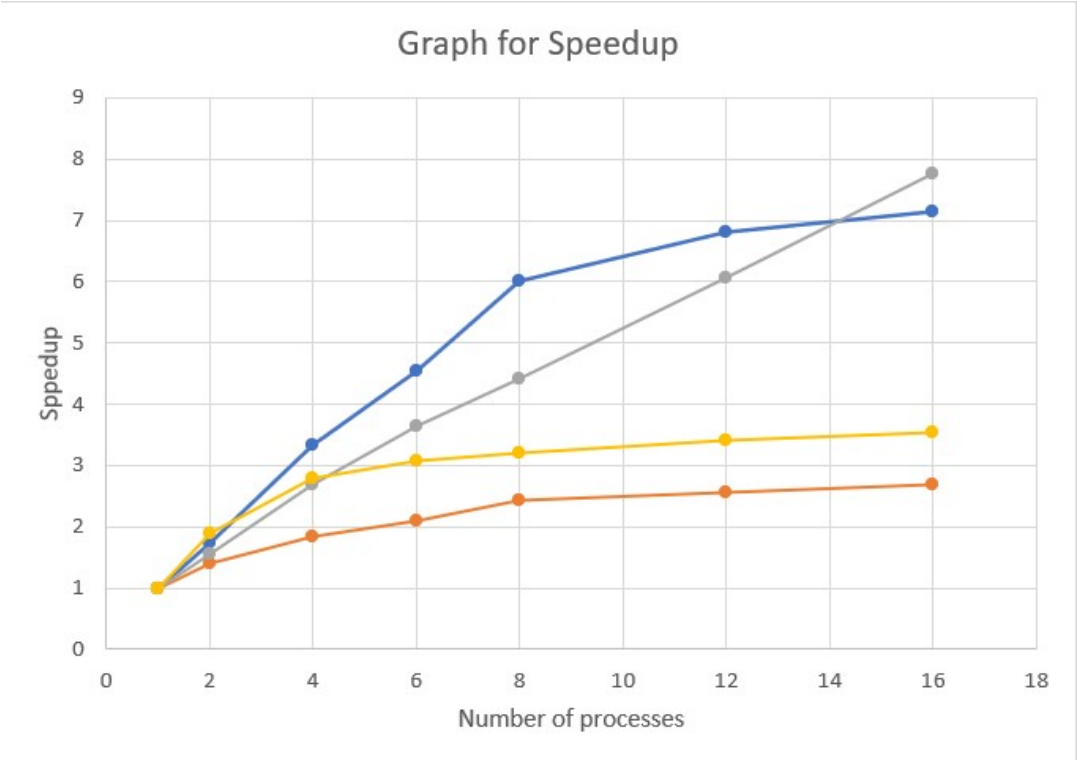

Figure 8: Speedup vs Number of processes for test 5 to 8

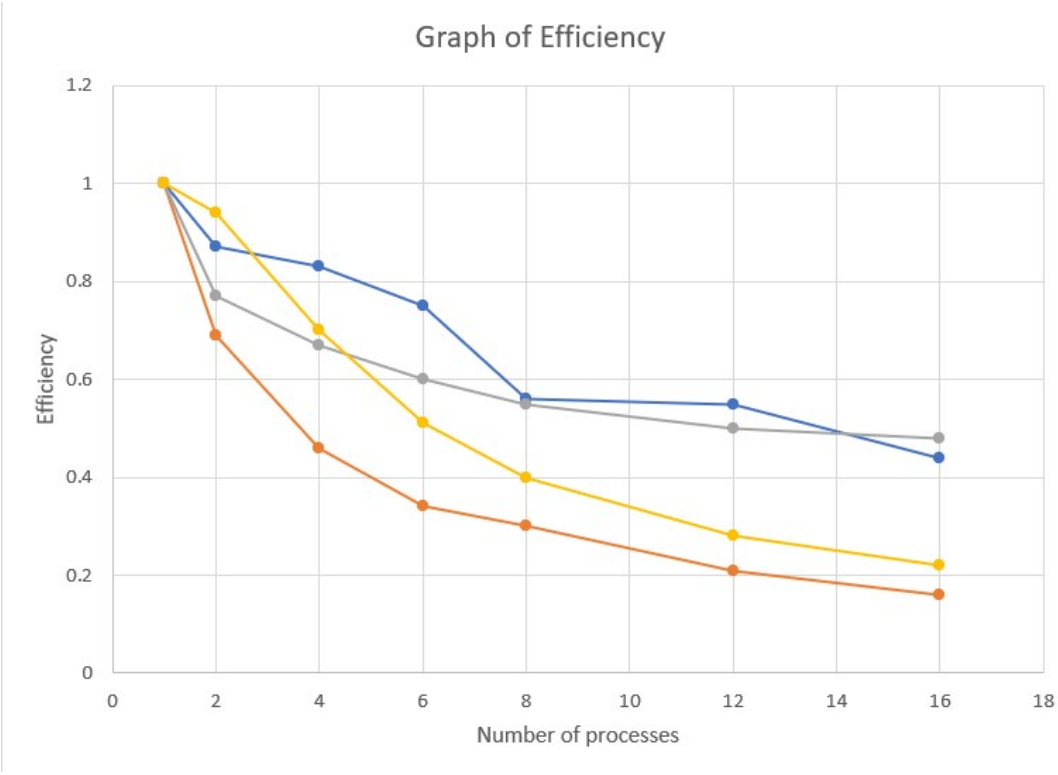

Figure 9: Efficiency vs Number of processes for test 5 to 8

Observation

One of the key observations which are obtained is that the running time of the code gets reduced by many times as we increase the number of processes. This can be clearly seen in the execution time plots for test cases 5, 6, 7 and 8 shown below.

Further, it can be seen as we increase the number of processes from say 12 to 16 the change in the execution time is not much. This is because as the number of processes increases the overhead increases which somewhat counters the effect of parallelizing the code.

Speed-up as is seen in the graph increases as we increase the number of processes for each of the test cases. The speedup is directly linked with the execution times hence the trend. Further for test 5 and test 7, we have a high speed-up as compared to tests 6 and 8 which also have a decent speed-up.

The efficiency decreases as we increase the number of processes as the overhead of maintaining and managing processes increases. The efficiency is less in test cases where the time taken is less, because the overhead time is significant in comparison to the computation time.