COL380 Assignment 2

Chinmay Mittal (2020CS10336) and Kartik Sharma (2020CS10351)

Sequential Algorithm and Data Structures

We first implemented the sequential algorithm with the *Prefilter, Initialize, and FilterEdges* routines.
We have maintained a *Graph class* at each process to store the graph and all graph processing functions are internal to the Graph class.
We store the adjacency lists in the graphs as an *unordered_set,* since it provides fast deletes (O(1)) and fast traversal (O(n)) which are operations used many times for maintaining the graph.

For handling multiple values of k, we process them in a sequential order and keep deleting edges if they cannot be part of any k-truss. For the k+1 truss we start from the reduced graph from computation of k-truss. Once the graph becomes disconnected it will always remain disconnected and we stop processing for subsequent k values.

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 with itself. This helps in saving a lot of extra space for each process in our distributed algorithm. We also *never do triangle storage* and hence the space complexity of our parallel program is O(n+m) which does not scale with the number of the 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.

We store the support count of all the edges for which we are responsible; this is initialized once and is 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 potential problem is that the call is blocking and that some processes might be choked by other processes. To remove this we do a *faster periodic synchronization* between the processes when any of them deletes a fixed number of edges without keeping other processes waiting for it to process all its edges.

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 there exists a k truss. 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, one approach is to repeatedly gather together local parts of the graph with every process and then do regular dfs to search for connected components. We also intend to 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 process with the last rank finally knows the connected components of the global graph and outputs it in the file.

Function Level Analysis

We evaluated the execution time of different functions in our algorithm to determine bottlenecks and scope of improvements. Following is an example of the time taken by different functions.

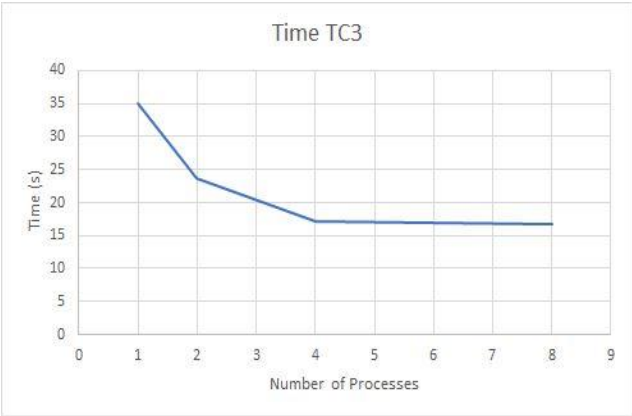
```
Elapsed Time ReadFile      p:     140.419 ms
Elapsed Time Neighbour Graph p:   813.715 ms
Elapsed Time Prefilter     p:3    0.103735 ms
Elapsed Time K-TRUSS       p:3    4076.05 ms
Elapsed Time G-SYNC        p:3    0.006692 ms
Elapsed Time Prefilter     p:4    0.094037 ms
Elapsed Time K-TRUSS       p:4    491.778 ms
Elapsed Time G-SYNC        p:4    0.001443 ms
Elapsed Time Prefilter     p:5    0.047459 ms
Elapsed Time K-TRUSS       p:5    1815.58 ms
Elapsed Time G-SYNC        p:5    0.000982 ms
Elapsed Time Prefilter     p:6    0.058259 ms
Elapsed Time K-TRUSS       p:6    18345.3 ms
Elapsed Time G-SYNC        p:6    0.11668 ms
```

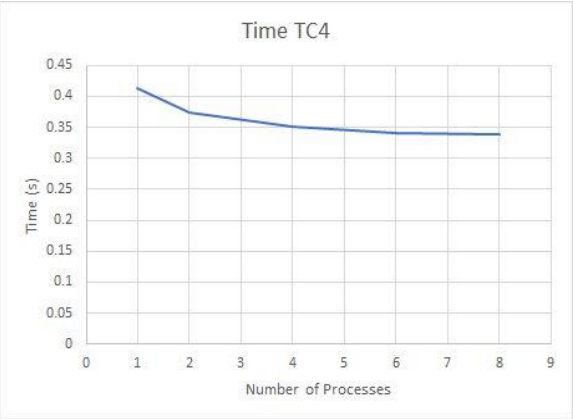
The (file read time * the number of processes) remained same, indicating file being read parallelly. The majority of the time was taken to compute the ktruss. Once we received a 0 value for a particular k, the output should be 0 for subsequent values of k, thus saving time. The G-Sync function syncs the partial graph stored in the processes and calculates the connected components of the graph parallelly. We used this information to remove bottlenecks by using efficient data structures like sets, making small functions inline, and removing redundant computations from our program.

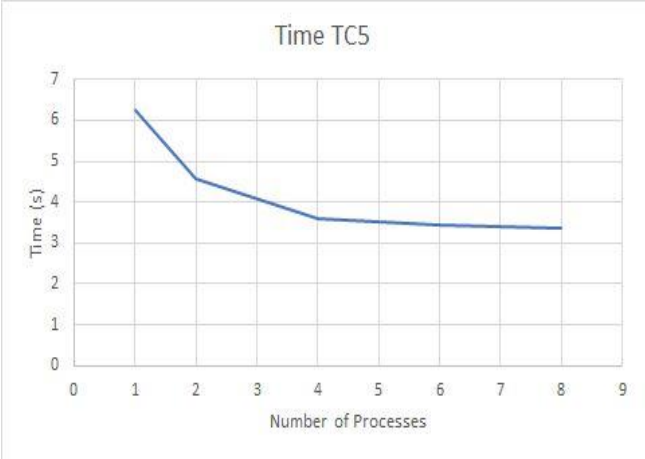
Speed-Up Analysis

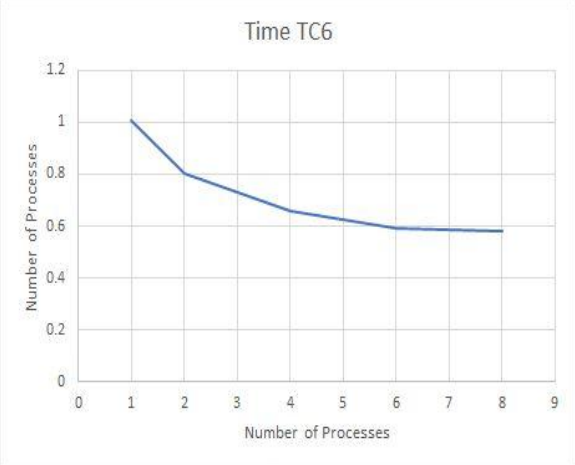
The following speed-up analysis has been done on our *local machine*. We ran our tests on CSS also which was performing better than our local machine, but later on, because of excessive load on the server, the output was coming out to be very slow.
We tested our implementation successfully on the large graph as well. It took *~10 minutes to run on the CSS cluster.*

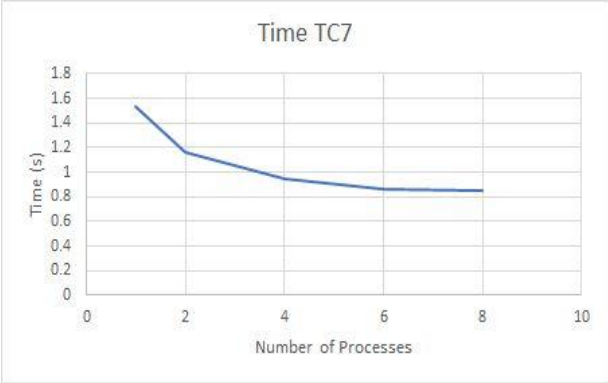
Speedup Graphs

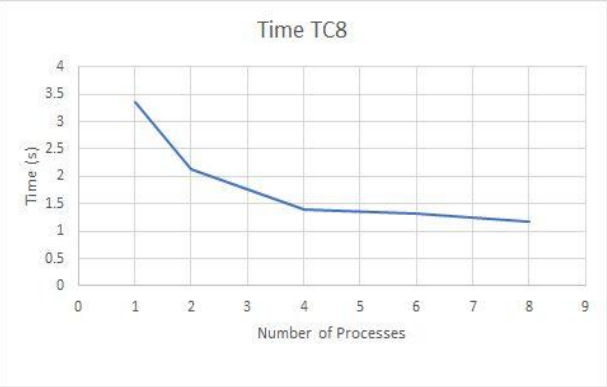












Speedup and Efficiency Tables

TEST CASE 3

Number of Processes (p)	**Time (T_p)**	**Speedup $S_p = \frac{T1}{Tp}$**	**Efficiency $E_p = \frac{Sp}{p}$**
1	34.98	1	1
2	23.76	1.47	0.735
4	17.16	2.03	0.5075
6	16.88	2.07	0.345
8	16.69	2.09	0.258

TEST CASE 4

Number of Processes (p)	**Time (T_p)**	**Speedup $S_p = \frac{T1}{Tp}$**	**Efficiency $E_p = \frac{Sp}{p}$**
1	0.414	1	1
2	0.374	1.106	0.553
4	0.35	1.18	0.295
6	0.34	1.21	0.20
8	0.338	1.22	0.152

TEST CASE 5

Number of Processes (p)	Time (T_p)	Speedup $S_p = \frac{T1}{Tp}$	Efficiency $E_p = \frac{Sp}{p}$
1	6.25	1	1
2	4.59	1.36	0.68
4	3.58	1.74	0.435
6	3.45	1.81	0.301
8	3.35	1.86	0.232

TEST CASE 6

Number of Processes (p)	Time (T_p)	Speedup $S_p = \frac{T1}{Tp}$	Efficiency $E_p = \frac{Sp}{p}$
1	1.009	1	1
2	0.8	1.26	0.63
4	0.66	1.52	0.38
6	0.59	1.71	0.285
8	0.58	1.73	0.216

TEST CASE 7

Number of Processes (p)	Time (T_p)	Speedup $S_p = \frac{T1}{Tp}$	Efficiency $E_p = \frac{Sp}{p}$
1	1.53	1	1
2	1.16	1.31	0.65
4	0.95	1.61	0.402
6	0.86	1.77	0.295
8	0.85	1.8	0.225

TEST CASE 8

Number of Processes (p)	Time (T_p)	Speedup $S_p = \frac{T1}{Tp}$	Efficiency $E_p = \frac{Sp}{p}$
1	3.35	1	1
2	2.13	1.57	0.785
4	1.404	2.308	0.577
6	1.326	2.52	0.42
8	1.183	2.83	0.35

References

1. Truss Decomposition on Shared-Memory Parallel Systems Systems Shaden Smith, Xing Liu†
2. Parallel Triangle Counting and k-Truss Identification using Graph-centric Methods Chad Voegele, Yi-Shan Lu, Sreepathi Pait and Keshav Pingali
3. C. Voegele, Y. S. Lu, S. Pai, and K. Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In 2017 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–7, Sept 2017.
4. Roger Pearce. Triangle counting for scale-free graphs at scale in distributed memory. In 2017 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–4. IEEE, 2017