COL380 Assignment 3 - Report

Ujjwal Mehta | Entry No. : 2020CS10401

Solanki Divyamsinh Ratnasinh | Entry No. : 2020CS10388

Introduction:

In this assignment, we have performed the task of Truss decomposition of a given large graph ,given in a byte file format by computing the Trussity values of the edges of graph in a parallel distributed manner by the use of OpenMPI and OpenMP and then further we tested and experimented our implementation on the IIT Delhi hpc using which we solve for the Task 1 and Task 2 of this assignment.

Implementation for Task 1:

The implementation is referenced from the paper Improved Distributed Algorithm for Graph Truss Decomposition and we have implemented the minTruss algorithm in a parallel distributed way from this paper which is based on computing the trussity values of the edges of the graph. By trussity of an edge e being equal to k means that inside that graph G, there exists a k truss containing the edge e and that value of k is the maximum value of k whose corresponding k truss can have edge e. Now in order to compute the Trussities and solve for the Task 1, we have the used the following approach in a distributed manner using OpenMPI.

Approach:

1. Step 1 : We first read the graph from the byte file and distribute the edges and vertices of the graph among different processes in a following manner: Let p be the number of processes on which we are running our MPI code then distribute the vertices starting from index 0 to n-1 in a equally contiguous manner to all processes i.e. rank 0 process will get the vertices from 0 to $\lfloor n/p \rfloor$ then to rank 1 we give vertices from $\lfloor n/p \rfloor$ to $\lfloor 2 * n/p \rfloor$ and so on. Now as for the edges distribution, we distribute an edge (u,v) to the process containing u if the deg(u)< deg(v) or if the deg(u) = deg(v) and id(u) < id(v). After this our graph distribution has been done.
2. Step 2: Now for the implementation of minTruss, we initialise the trussity value of each edge as the supp(e) + 2 where supp(e) is the number of triangles in which edge e is lying and in order to calculate this, we first pre process the triplets in all the processes of the form (u,v,w) in which u vertex belongs to the process and (u,v) and (u,w) are edges of the process and then we send this kind of triplets to

the process of edge (v,w) and in order to achieve sending such triplets from a process and well as receiving such triplets from other processes we use **MPI_Alltoallv** primitive which does this job, and then once we get these triplets, which check if the triangle exists or not and if it exists then we store the triangles mapped from their edges and also send the confirmation of triangles in a similar manner. After this step, we have stored the triangles of all the edges in a form of a map in the corresponding process containing the edge.

3. Step 3: (Iteration) In this step we enter a while loop which in each iteration first collects from all the processes the minimum value of current trussity using the **MPI_Alltoall** primitive and then retire the correponding edges whose current trussity value is equal to this minimum then we finally exchange the triangles that need to be removed due to the retiring of these edges from each process and these triangles are again exchanged via the **MPI_Alltoallv** primitive in a distributed manner and then we delete the triangles from the corresponding edge and as well as update the current trussity value of each updated edge conditioned to that it cannot go below the current minimum.
4. Step 4: Finally we break out of the iterative loop once all the processes receive that the minimum trussity obtained is more then Kmax + 2 since after this value, all the remaining edges will be the part of Kmax+2 th truss which is the maximum truss that we need to find.
5. Step 5: Finally after each rank has done computing the trussity value of its edge, then we for each value of k, we do a parallel DSU (disjoint set union) using **MPI_Bcast** in order to find the connected components of the edge induced subgraph with edges whose truss value is greater then equal to k+2, in each rank then we finally write the output of connected component in our output file, using **MPI_File_write_ordered** to write from each rank in order.

Optimisations:

In order to improve the performance of our given parallel algorithm as well as reduce the memory usage for a single process, we did the following optimisations on our code:

1. In order to confirm for the triangles in preprocessing and initialisation of trussity, earlier we were sending the triangle processes of both the vertices of v and w, after now we only send them to the process containing edge (v,w) for confirmation of triangle (u,v,w) hence reducing the communication latency.
2. Earlier we used c++ maps in order to store the triangles and other things for each edge of a process, now we use unordered_map which led to significant decrease in runtime.
3. We clear all the send and receive buffers once the communication is over which inturn decreases our memory usage.
4. Used the directive

```
#pragma omp parallel for reduction
```

to find the minimum current trussity edge in each iteration of each rank.
5. Used the directive

```
#pragma omp parallel for nowait
```

For computing the partitions of a big vector, and merged the big vector using the critical section. This way we parallelized the computation in memory IO

Implementation for Task 2:

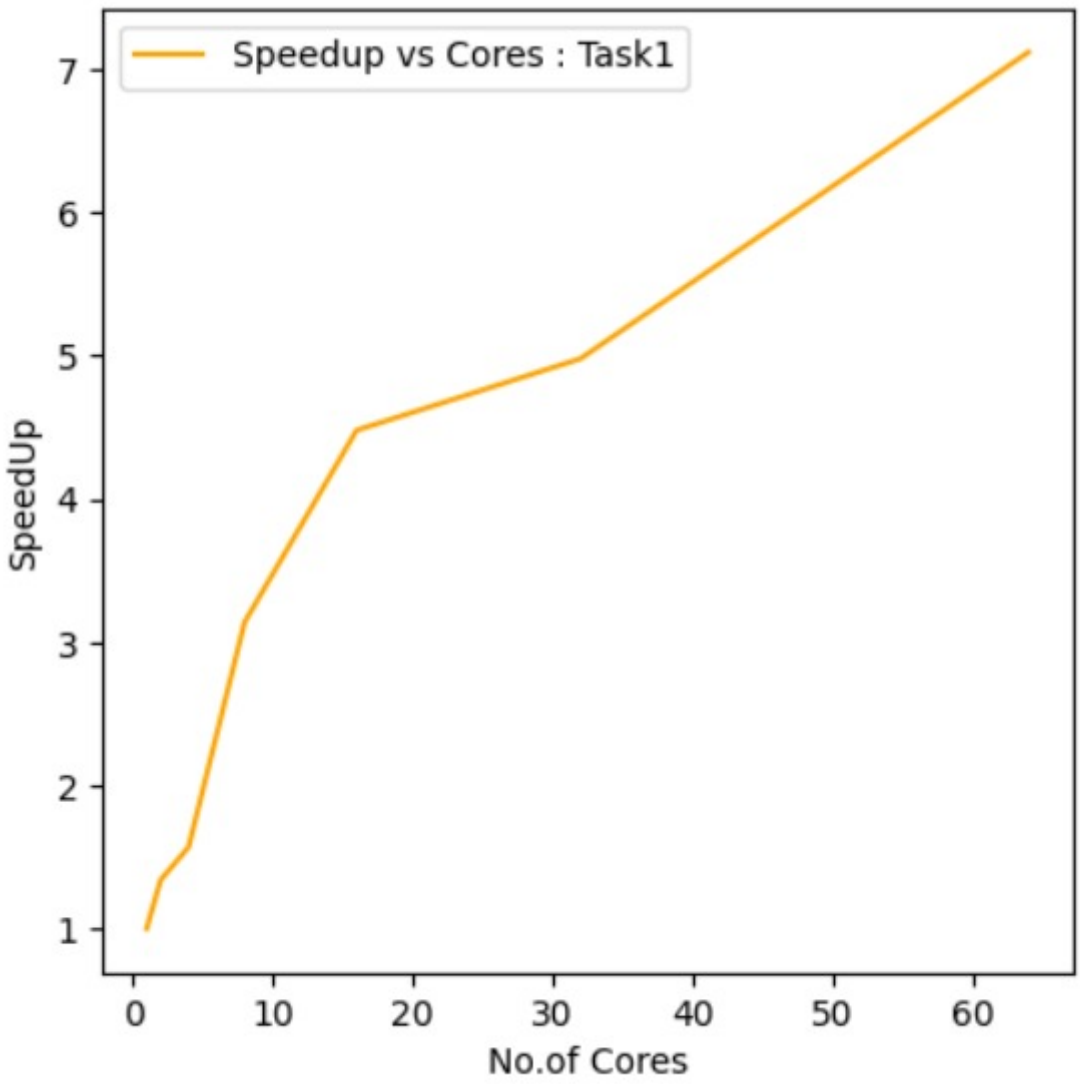
In order to implement task 2 we reused the same part of above approach to compute the truss value of each edge for all the ranks and then we apply DSU(disjoint set union) in parallel using **MPI_Bcast** in order to evaluate the connected components at each rank then finally each rank traverses its own set of vertices to check if it is an influencer vertex or not, then we finally output in parallel in the output file, using **MPI_File_write_ordered**.

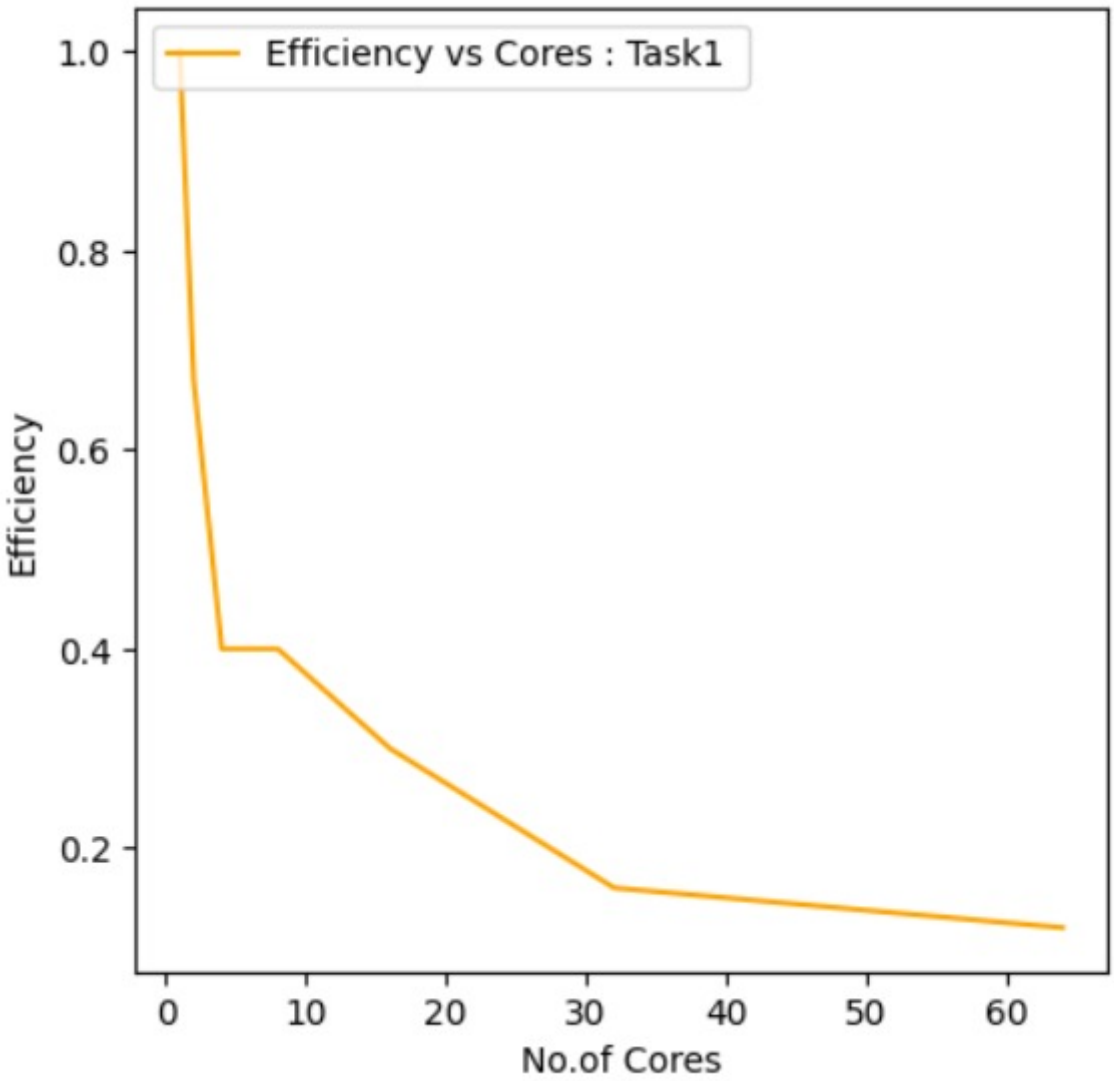
Approach:

1. Step 1: Compute the truss value of edges in a same way as mentioned in task 1
2. Step 2: Find the connected components in each rank using parallel DSU(disjoint set union) by using **MPI_Bcast** by broadcasting edges to each rank in order of ranks then computing the connected components in each rank parallelly.
3. Step 3: Each rank traverses its own set of vertices in order to check if that particular vertex is an influencer or not, by checking in how many connected components its neighbours lie.
4. Step 4: Finally we write in output file by each rank, the group it influences using the **MPI_File_write_ordered**.

Plot Graph of Speedup and Efficiency for Task 1:

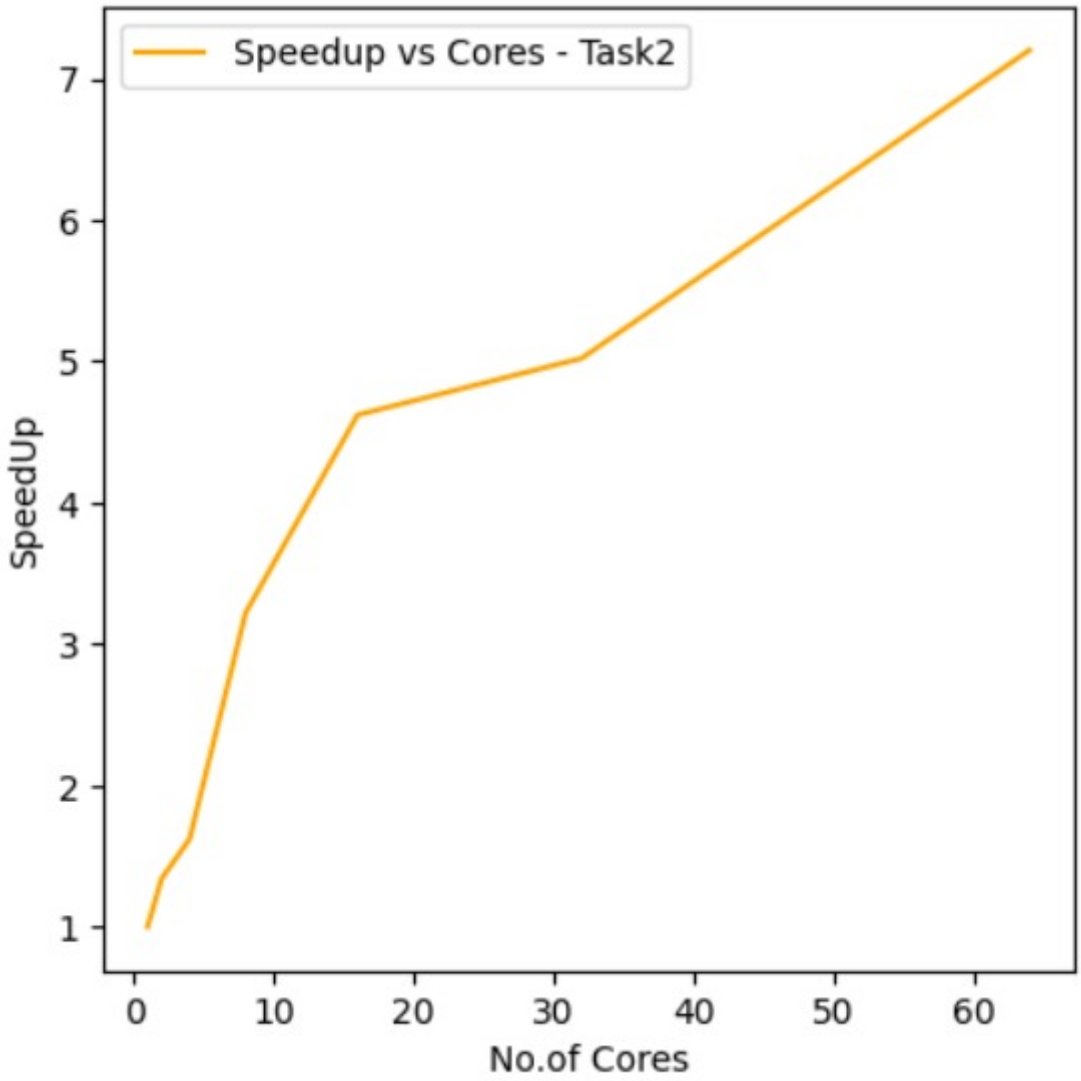
The plot for Efficiency and Speedup for varying cores for task 1 is as follows:





Plot Graph of Speedup and Efficiency for Task 2:

The plot for Efficiency and Speedup for varying cores for task 2 is as follows:



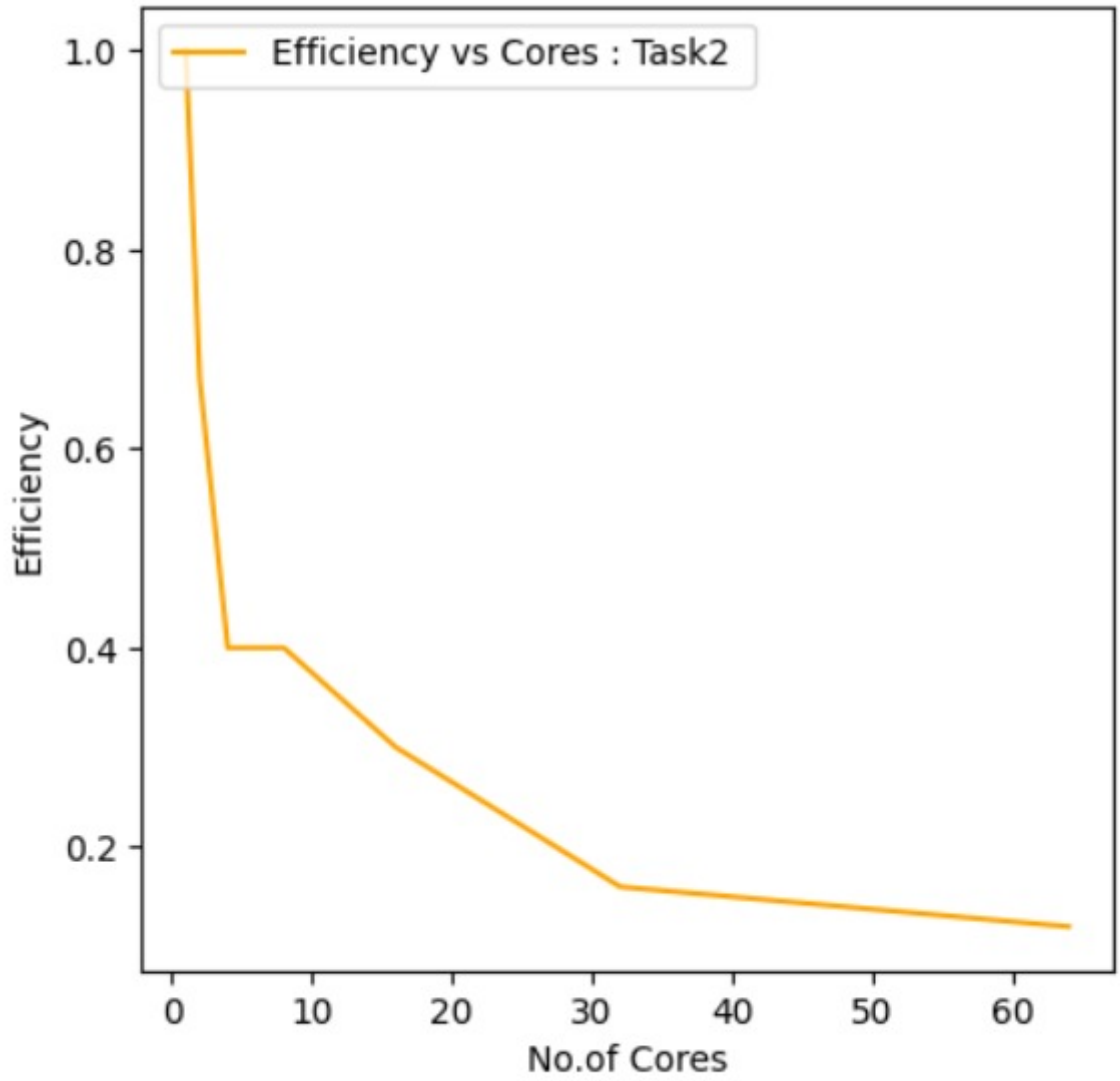


Table for Iso-Efficiency and Speedups (from csv file):

The table for iso-efficiency and speedup after experimenting for various test cases is as follows:

Task_id	1	1	2	2	1	2	1	2
Metric	Speedup	Efficiency	Speedup	Efficiency	Iso-efficiency	Iso-efficiency	Sequential Fraction	Sequential Fraction
nodes = 2, thread = 2	1.57	0.39	1.62	0.4	$p^{0.7}$	$p^{0.7}$		
nodes = 2, thread = 4	1.91	0.24	1.97	0.25				
nodes = 2, thread = 8	2.73	0.17	2.81	0.18				
nodes = 4, thread = 2	3.14	0.39	3.2	0.4				
nodes = 4, thread = 4	4.48	0.28	4.62	0.29				
nodes = 4, thread = 8	6.41	0.21	6.35	0.21				
nodes = 8, thread = 2	3.49	0.22	3.44	0.22				
nodes = 8, thread = 4	4.98	0.16	5.02	0.16				
nodes = 8, thread = 8	7.12	0.12	7.06	0.12			0.4	0.4

Here in order to get the iso-efficiency, we tested out our code with varying graph sizes and after changing the graph sizes by a certain function of p (given in form of

iso-efficiency $p^{0.7}$), we deduced that in order to keep the efficiency same we need to increase our input size by this function.

Justification of Scalability:

The code that we wrote has a parallel fraction of 0.6, and as we increase the number of ranks and threads, this part of code is parallelized in an efficient manner because the graph that we distribute is distributed equally with respect to vertices among all the ranks and even the truss computation happens parallely, making this part of code quite scalable.

Task 1 Scalability:

This task is scalable because of the truss computation part which as already explained is quite parallel, as well as we compute the connected components to be reported in output file, by using parallel DSU(disjoint set union) which further helps in parallelizing.

Task 2 Scalability:

This task is scalable because of the truss computation part which as already explained is quite parallel, as well as we find the influencer vertices in parallel way(each rank computes its own influencers) thus making it scalable.

References:

1. We thank HPC IIT Delhi for helping us test our code and check for the scalability and other optimisations.
2. We took the reference of MinTruss from the following link : https://link.springer.com/chapter/10.1007/978-3-319-96983-1_50
3. We took the help from official documentation of OpenMPI and OpenMP