IIT DELHI

COL380: Introduction to Parallel and Distributed Programming

Assignment-3 Report

Prof. Subodh Kumar

Ankit Mehra (2019CS10329)
Souvagya Ranjan Sethi (2019CS10405)
Department of Computer Science & Engineering

April 13, 2023

1 Task-1

1.1 Algorithm and Approach

We are given a simple, undirected and unweighted graph G=(V,E), and were asked to find the maximal k-truss for different k values in a given range. a k-truss of graph G is a subgraph, H in which each edge is part of at least $(k-2)$ triangles within H. In other words, every edge in the k-truss is part of at least (k - 2) triangles made up of edged and vertices of that truss. We attempted numerous algorithms for this task given in the assignment statements and in different papers, but finally we implemented a modified approach, which is a combination of different approaches. Detailed approach is given bewlo.

1.1.1 Approach for this task

1. **Division of vertices:** We read the graph in distributed settings i.e., In process - 0, we first read all the vertices and their respective degrees and then sorted the vertices according to their degrees. Then according to number of process, we divided the vertices to different process in a round robin fashion (using their degree), so that workload assigned to each process should be almost equal. We used MPI_Bcast to send this information to all processes.

2. **Reading Subgraph:** Then, from each process, we read all the neighbours of all the vertices that are assigned to each process. So, each process have a subgraph, on which it will do all the computations and information about the parent of each vertex and each edge. we ensured that each edge will be processed by only one processor (by using the heuristic that is given in this paper).

3. **Support Computation:** We found the support and triangle(i.e. all supporting vertices) for an edge in distributed manner. Since, each process have a subgraph, so each processor will find the support of all the edges assigned to it only. The algorithm used to find the support for an edge is given below:

Algorithm 1 Algorithm to Find support and Triangles for an edge

Require: subgraph assigned to this process, G'

// Each process will perform this for their respective edges.
for All monotonic edge = (u,v) in G' **do**
    Initialize an empty vector to store all queries of form(u,v,w).
    Add all w's (neighbour of u) to vector such that edge(u,w) is monotonic.
    Use MPI_Alltoallv to communicate all queries for edge(u,v) and to recv the corresponding response.
    According to recvd response, update the support and triangles
end for
OUTPUT: support and triangle for each edge

4. **Truss Computation:** After computing support and triangles for all processes, we have used **MPI_Gatherv** to collect support and triangles from each process into process-0. Now, Since process-0 Then, the remaining computation of finding K-Truss for all k values has been done by process-0 only. For finding K-truss for a particular value of K, following approach has been used:

    - Firstly, we are reading the whole graph in process-0, which is then modified using different functions to find k-truss.
    - We found all the edges having support value less than (k - 2) and marked them as *deletable*. Pseudo code for this step is given in ***Initialize*** function.
    - Then we filtered the edges by taking *deletable* as the starting point. Filtering of edges is done using **OpenMP**. We used *#pragma omp for* to divide the iteration of the for loop among the threads. Since, code inside the for loop is modifying some shared variables, so to prevent the ddeadlocks, we used locks by *critical sections pragma*. In the ***FilterEdges*** function, we run the infinite loop which run till there is an edge to delete. Code snippet is given below.
    - Finally, we deleted all the redundant vertices i.e., removed all the vertices who has degree 0 from the graph resulting the k-truss for a particular k-value.

```
void FilterEdges(graph, k, deletable, support_map, triangle_map)
    while (True)
        if (deletable.empty()) : break
        Initialise newDeletable
        #pragma omp parallel for
        for (int i = 0; i < deletable.size(); i++):
            edge (u,v) = deletable[i]
            if (u,v) not in graph : continue
            #pragma omp critical{
                G = G - edge (u,v)
            }
            vertices_UV = Supporting Vertices of (u,v)
            for all w in vertices_UV :
                #pragma omp critical
                {
                    if edge (u,w) in graph and in supportMap :
                        supportMap[uw]--
                        triangleMap[uw].erase(v)
                        if (support_map[uw] < k-2) :
                            newDeletable.push_back(uw)
                    if edge (v,w) in graph and in supportMap :
                        supportMap[vw]--
                        triangleMap[vw].erase(u)
                        if (support_map[vw] < k-2) :
                            newDeletable.push_back(vw)
                }
        #pragma omp critical {deletable = newDeletable};
```

1.2 Analysis

We have done the analysis using testcase-1 which was given in piazza. Also, we have used hpc server for running the code and finding the time.
Description of the test case:

Nodes (V) = 50,000

Edges (E) = 5,48,220

range of K = [3, 10]

1.2.1 num_t = 2, Varying p

No. of Processor	Time (s) [verbose=0]	Time (s) [verbose=1]
2	55.303	57.691
4	40.790	43.786
6	35.132	38.164
8	32.094	35.032
10	29.913	32.704
12	28.209	30.917

Table 1: No. of Process vs Time

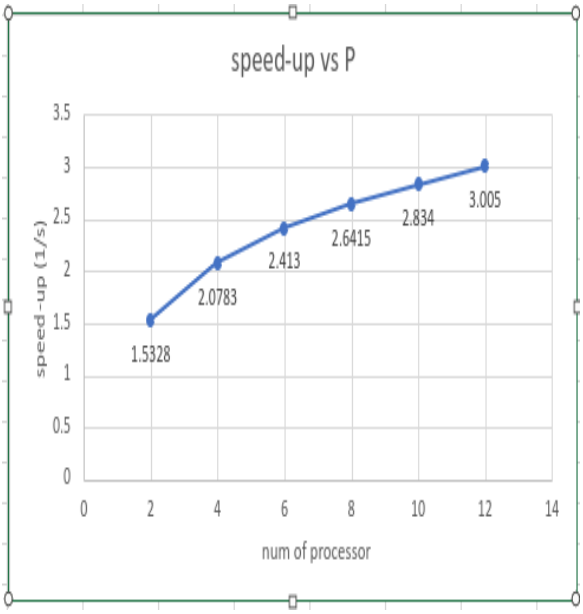

Figure 1: Verbose=0

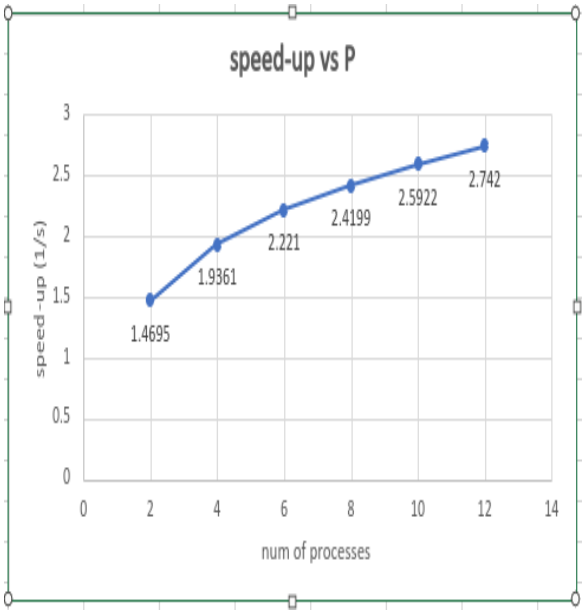

Figure 2: verbose=1

Figure 3: Constant num_t=2

1.2.2 Constant p=8, Varying num_t

Description of both test cases:

test case-1
Nodes (V) = 50,000
Edges (E) = 5,48,220
range of K = [3, 10]

test case-2
Nodes (V) = 1,00,000
Edges (E) = 6,25,317
range of K = [4, 8]

No of Threads	Time (s) testcase-1	Time (s) testcase-3
2	37.045	24.519
4	36.783	24.325
6	36.540	24.283
8	34.488	23.519

Table 2: No of Threads vs Time

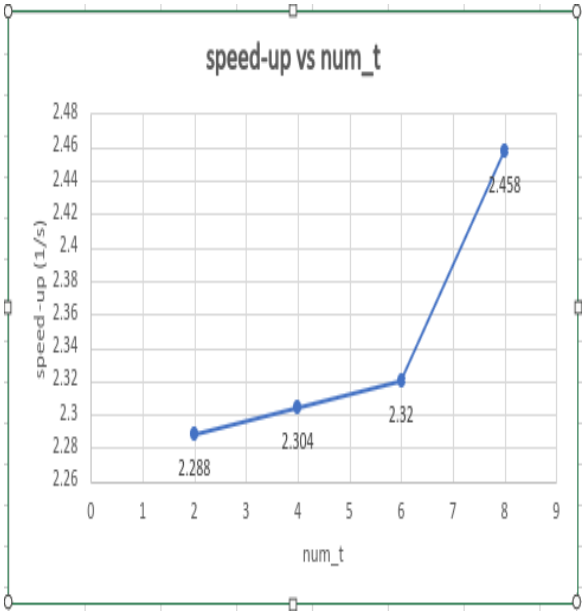

Figure 4: testcase-1

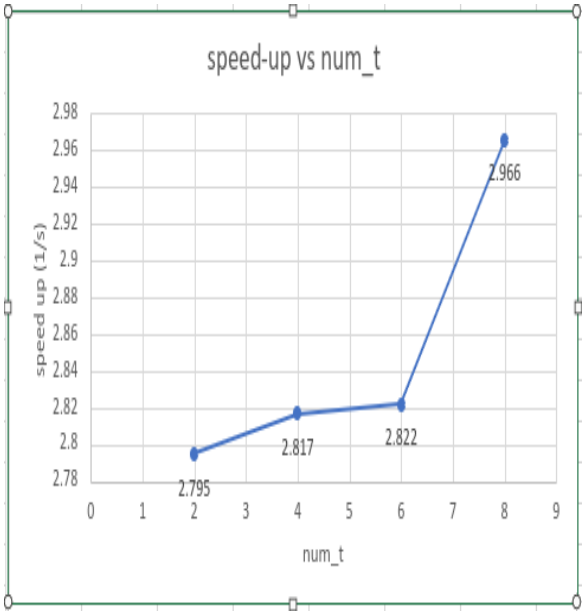

Figure 5: testcase-3

Figure 6: Constant p=8

1.3 Observations

- **Constant num_t=2 :** In first case, we kept the num_t constant and varied number of processors(p) and found the execution time. From the graph 3, we can see that as the number of processors are increasing, the execution time is continuously decreasing, which is expected because more number of processor will result in less workload to individual process.

- **Constant p=8 :** In this case, we kept the number of processor constant throughout and varied num_t from 2 to 8 and noted the execution time. From graph 6, one

can observe that for both test cases, execution time is continuously increasing with increasing num_t, this could be because in the portion of code where *openmp* is used, we also used used critical

1.4 Iso-Efficiency

According to our estimate, the iso-efficiency of this is $O(p^2)$. This is because on running the experiments, when we were doubling the number of threads, we had to double the size of input as well. Overall, isoefficiency is an important consideration when designing parallel algorithms, as it can help determine the maximum number of processors that can be effectively used to solve a given problem.

1.5 Sequential section

For calculating the sequential portion of the code, we use the formula

$$f = \frac{\frac{1}{s} - \frac{1}{p}}{1 - \frac{1}{p}}$$

In our implementation, we have accumulated the support map by process with rank0 and used it to find to the k-truss for the whole graph. So, this part is only done by one process. Also, during writing the output into the file, we are doing this by only one process which increases the sequantial portion of the code.

1.6 Scaling of the code

Our implementation scales on increasing the size of the input graph. Also, we have distribution the computation among processes. So, as the number of processes increase, the computation time decreases. Thus the computation load is distributed among the processes making the code scalable.

2 Task - 2

2.1 Algorithm and Approach

For this task, we were asked to find the influencer vertices. The influencer vertices refer to the vertices that are connected to at least p (input argument) k-truss components. The influencer vertex can itself be part of a k-truss component. For this task, we need to find the Ego-network for each influencer vertex. The Ego-network of an individual influencer vertex v is a subgraph of G formed by all v's neighbors. In this task, we find the k-truss for endK for the given problem.

2.1.1 Approach for this task

1. First, we find the K-truss for k = endK. as described in task1 in rank0. The truss is stored in the form of connected components (vector¡set¡int¿¿).
2. Then we Send the Connected Components to all processes by using MPI_Send.
3. All the process have their respective sub-graphs. So, on receiving the K-truss connected components, they create the influencerMap which stores the index of the connected component(group) that is influenced by the vertex.
4. We then filter out the vertices which influence more than P connected components.

2.2 Analysis

We have done the analysis using testcase-1 which was given in piazza. Also, we have used hpc server for running the code and finding the time.
Description of the test case:
Nodes (V) = 50,000

Edges (E) = 5,48,220

range of K = [3, 10]

2.2.1 num_t = 2, Varying p

No. of Processor	Time (s) [verbose=1]
2	48.791
4	36.512
6	30.773
8	27.744
10	25.910
12	25.022

Table 3: No. of Process vs Time

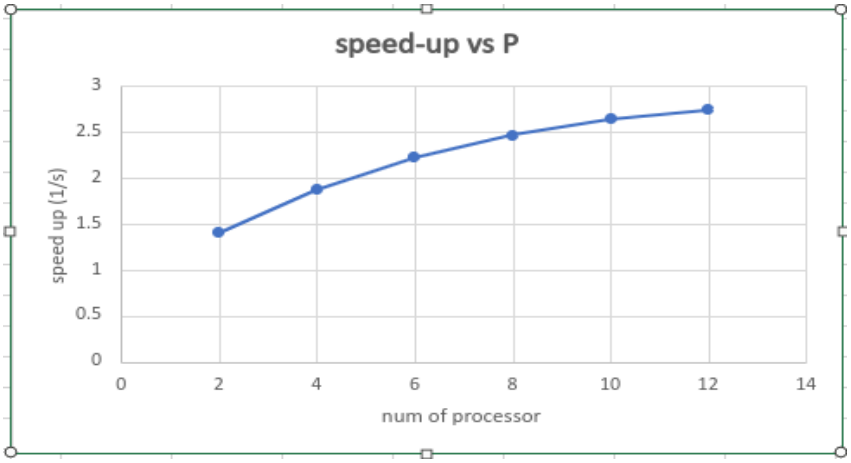

Figure 7: $\text{num}_t = 2$

2.3 Iso-Efficiency

[h] According to our estimate, the iso-efficiency of this is $O(p^2)$. This is because on running the experiments, when we were doubling the number of threads, we had to double the size of input as well. Overall, isoefficiency is an important consideration when designing parallel algorithms, as it can help determine the maximum number of processors that can be effectively used to solve a given problem.

2.4 Sequential portion of Code

In task2, again similar to the task1, we calculate the k-truss in process with rank0. It accumulates to the sequential portion of the code. Also, each process after finder the influencer map send them to rank0. Then rank0 processes them and find what to write to the output file.

$$f = \frac{\frac{1}{s} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Using this function, the sequential portion of the code was found to be

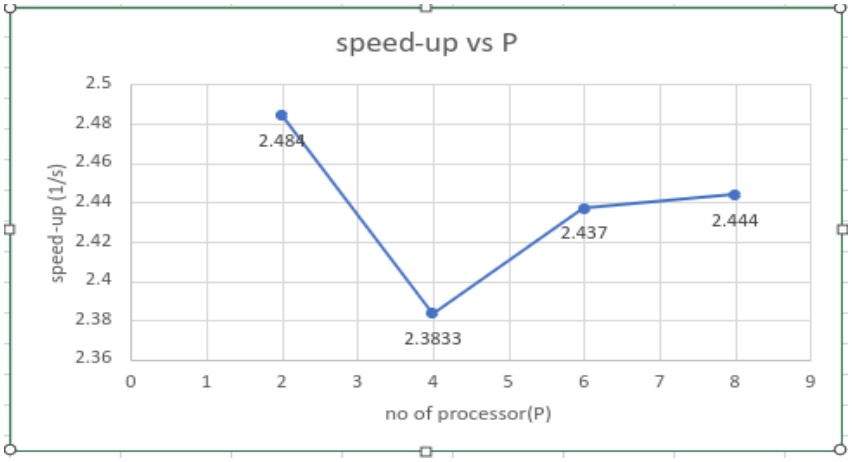

Figure 8: no of P=8

No of Threads	**Time (s) testcase-1**
2	27.601
4	28.774
6	28.140
8	28.057

Table 4: No of Threads vs Time

2.5 Scalability of the Code

Our code scales well we increase the size of input. Also this will give better efficiency as we are increasing the size of the input. With increasing number of processors the load will also get distributed in efficient manner and this will increase the effect of distrubuting the load among different processors. All in all our code scales well.