COL380 Assignment-3

Tanish Gupta, Rishi Jain

7th April 2023

1 Introduction

In this assignment, we implemented task 2 - i.e. found out influencer vertices. After that, we used OpenMP to speedup the sequential part of code on each processor - for both tasks 1 and 2.

2 Task-1

2.1 Approach

For task 1, the following approach was followed:

1. We first distributed the graph into p parts - each processor owns some of the n nodes. For each edge (u, v) - we distribute it to some processor p that either is the owner of u or is the owner of v. We decide this using a relation on both nodes (Assign it to the node with less degree).
2. After distrbuting the nodes, we calculate the support of each edge in the suppCalc function.
3. Then we use the minTruss algorithm (just like assignment 2), which iteratively deletes the edges with support value that is less than k. This minTruss algorithm is implemented in our code in the propLight() function.
4. After these steps, we have final nodes on different processors, but we need connected components. So, we run a distributed BFS algorithm. We wrote this algorithm ourselves, by maintaining a frontier, and assigning colors to the same connected component. (This is needed only for verbose = 1).

2.2 Graphs for Speedup and Efficiency

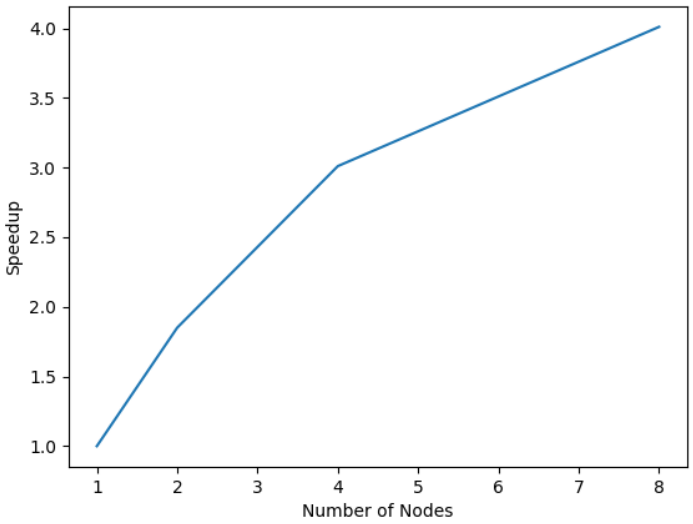

Figure 1: Speedup vs Number of Processors (Task 1)

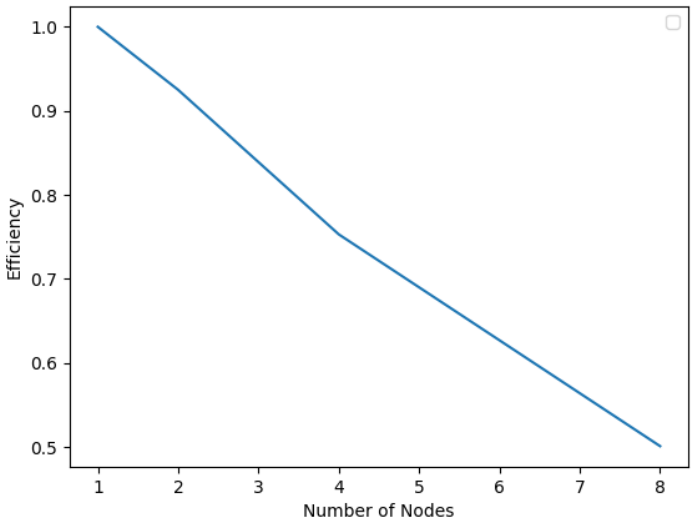

Figure 2: Efficiency vs Number of Processors (Task 1)

2.3 Iso-efficiency

The iso-efficiency of a scalable program indicates how (and if) the problem size must grow to maintain efficiency on increasingly large computing systems. A rapid growth in I with increasing p means that the only much larger problems can be efficiently solved on larger machines (which is not desired).
We can relate I to the overhead of parallelization $\bar{o}(n,p)$: the computation that is not required in the sequential solution. In other words, $\bar{o}(n,p)$ is the 'extra' time collectively spent by the parallel processors compared to the best sequential program. So,

$$\bar{o}(n,p) = t(n,p)p - t_1(n,1)$$

and,

$$I(p) = t_1(n,1) = t(n,p)p - \bar{o}(n,p)$$

In terms of efficiency, this becomes

$$I(p) = [\frac{\epsilon(n,p)}{1-\epsilon(n,p)}]\bar{o}(n,p)$$

Since for iso-efficiency, we need efficiency to be constant, so the term inside [] is constant, and hence iso-efficiency is proportional to the parallelization overhead. We thus, plot a graph of overhead vs the number of cores.

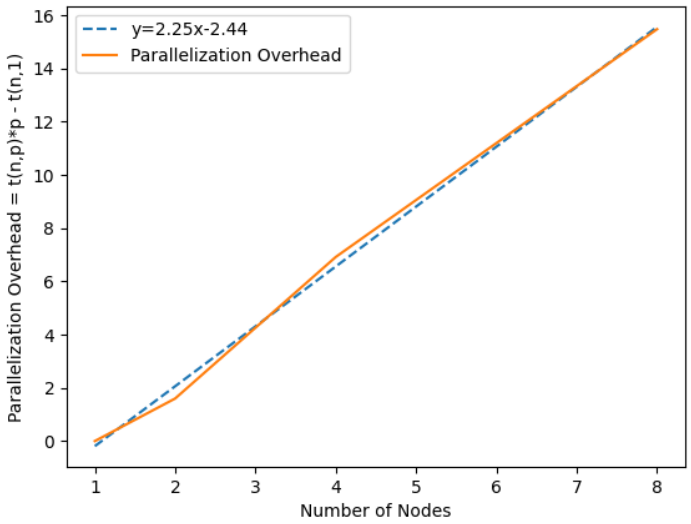

Figure 3: Overhead vs Number of Processors

2.4 Sequential Fraction

We use the Karp-Flatt Metric to estimate the unparallelizaed part f in the program, given the measured speed-up over the sequential execution S:

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

This metric is consistent with the Amdahl's law. We already know all the variables values here, and hence calculate the sequential fraction. Since these values will be slightly different for every pair of input variables, we include different values for sequential fraction and not just 1 value. We include these values in metric.csv file.

2.5 Scalability

Since we distribute the graph into multiple processors, and all computation on each node is fast, the solution is scalable to a large number of processors. Since we employ the distributed BFS algorithm, and we do not gather the vertices of the graph on any one processor, therefore, the code should improve in time complexity with increasing number of processors.

3 Task-2

3.1 Approach

For task 2 we did the following steps:

1. We first found out the connected components of the graph using Distributed BFS after the process done in task 1 verbose 0. After this step, all the nodes have the connected components of the graph. This is done by sending the connected components calculated by each node to all the other nodes using MPI_Allgatherv.

2. We iterate over all the nodes and find out the nodes which are connected to more than p components. This process is done in a distributed manner. Each processor iterates over its own nodes and finds the nodes connected to more than p components.

3. To find out the nodes which are connected to more than p components, we use pragma omp parallel for to do the task in parallel. We use a critical section to avoid data race.

4. After this step, each processor has a list of nodes that are connected to more than p components. After this step, we use MPI_Allgatherv to send this list to processor 0. After this step processor 0 has the list of all the nodes which are connected to more than p components.

5. Processor 0 the writes this list to the file.

3.2 Graphs for Speedup and Efficiency

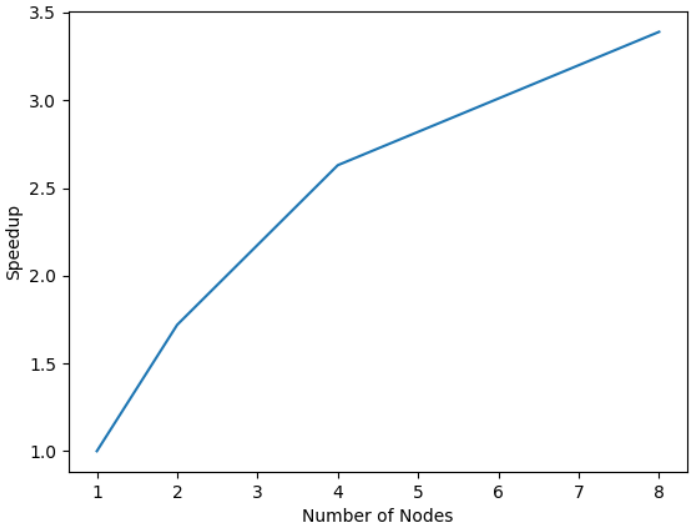

Figure 4: Speed vs Number of Processors for Task 2

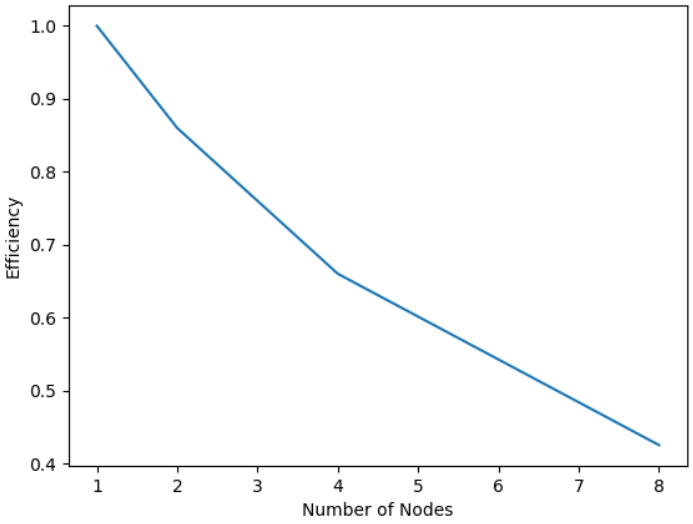

Figure 5: Efficiency vs Number of Processors for Task 2

3.3 Iso-efficiency

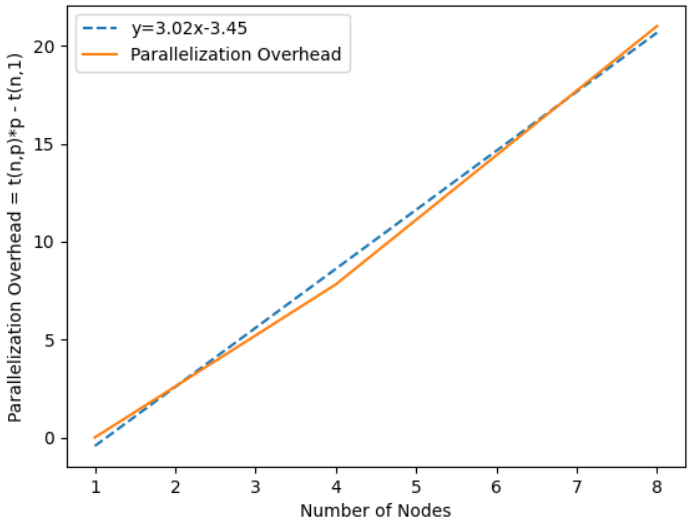

Figure 6: Overhead vs Number of Processors for Task 2

3.4 Sequential Fraction

Just like the task 1, we include all values of sequential fraction in metric.csv.

3.5 Scalability

For task 2, the part of calculating connected components is same as task 1. Thus, that part is scalable. Once we have the connected components, every processor finds the influencer vertex. This is done in parallel using openmp. This is again scalable since with more threads, the vertices will be divided among those threads (to find if it is an influencer vertex). Hence, our code is scalable.

4 Conclusion

In this assignment, we learnt how to integrate OpenMP and OpenMPI to leverage maximum speedup. We determined the k-truss of a given graph for some given value of k, and the influencer vertices in the graph. This has real-life applications as given in the assignment-2 PDF.