COL380: Assignment 3

Harshit Mawandia 2020CS10348

April 7, 2023

1 Task 1:

The algorithm consists of 4 different part as described in the Assignment PDF:

1.1 Sequential Implementation

1. Input
2. Prefilter
3. Initialise
4. Filter Edges
5. Output

The given code defines a function named `readInputFromFile` that takes a filename, n and m as input and reads a binary file to populate a map `adjList` that contains the adjacency list of a graph. The code also defines a function named `dfs` that performs depth-first search (DFS) on the graph given its adjacency list G, and stores the vertices visited in the `verts` vector. The main function of the code reads command-line arguments to set input and output paths, start and end values of k, task ID, and verbosity level. Then, for each k value in the range `[startk+2, endk+2]`, the code applies a pre-filtering step to remove nodes with fewer than k-1 neighbours, and then computes the support of each edge in the graph. If the support of an edge is less than k-2, the edge is marked for deletion. The remaining edges are output to a file. The `prefilter()` function removes all vertices with a degree less than k-1 (k is a parameter passed to the function) and updates the adjacency list accordingly. It achieves this by first identifying such vertices and then removing them from the graph. It also removes any vertex from the adjacency list of its neighbors that was just removed from the graph.

The `filterEdges()` function takes the graph and the parameter k as input and removes edges that do not satisfy a certain connectivity condition. For each pair of vertices a and b, it checks if the number of common neighbors of a and b is at least k-2. If not, it removes the edge between them. This is done by first computing the support (number of common neighbors) of each edge using a map data structure, and then iterating over all edges and deleting the ones with support less than k-2. The code also makes use of `MPI (Message Passing Interface)` to parallelize the edge filtering process across multiple processors.

1.2 Distributed Implementation

1.2.1 Approach:

We implement a parallel algorithm using the OpenMPI Interface. Our algorithm scales well for upto 32 ranks. We parallise filter edges function for each rank. Scaling beyond 32 ranks requires careful consideration of several factors like communication patterns, load balancing strategies, and hybrid parallelism.

1.3 Description:

The code uses the Message Passing Interface (MPI) library for parallel programming and communication between processes.

The MPI library provides a set of functions for inter-process communication, synchronization, and process management. In this code, the processes communicate with each other using the `MPI_Send` and `MPI_Recv` functions.

The code initializes MPI with `MPI_Init` and gets the number of processes and the rank of the current process with `MPI_Comm_size` and `MPI_Comm_rank`, respectively.

The code divides the work among the processes using the `MPI_Scatter` and `MPI_Gather` functions. The `MPI_Scatter` function divides the input graph data into equal-sized chunks and distributes them among the processes. Each process then performs the computation on its chunk of the graph data. The `MPI_Gather` function collects the output from each process and combines them to produce the final result.

During the computation, the processes exchange messages to coordinate their work. For example, in the `filterEdges` function, each process calculates the support values for a subset of edges. To compute the support values for an edge, a process needs to know the neighbors of the two vertices connected by the edge. Therefore, each process sends its neighbor list to the other processes using the `MPI_Send` and MPI_Recv functions.

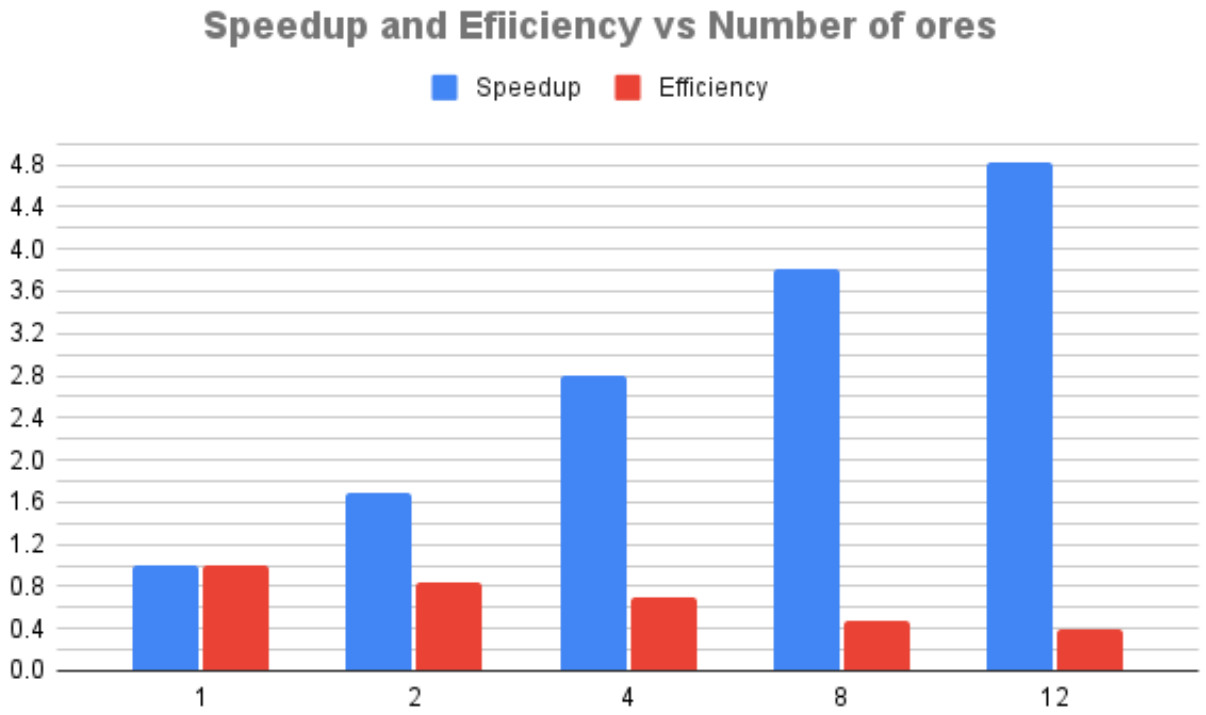
Overall, the code uses parallel programming and message passing to speed up the computation by distributing the work among multiple processes and allowing them to work concurrently. The use of MPI ensures that the processes can communicate and synchronize their work effectively, leading to efficient parallel execution.

2 Speedup and Efficiency

The Time taken by parallel program under different number of CPU's is given below

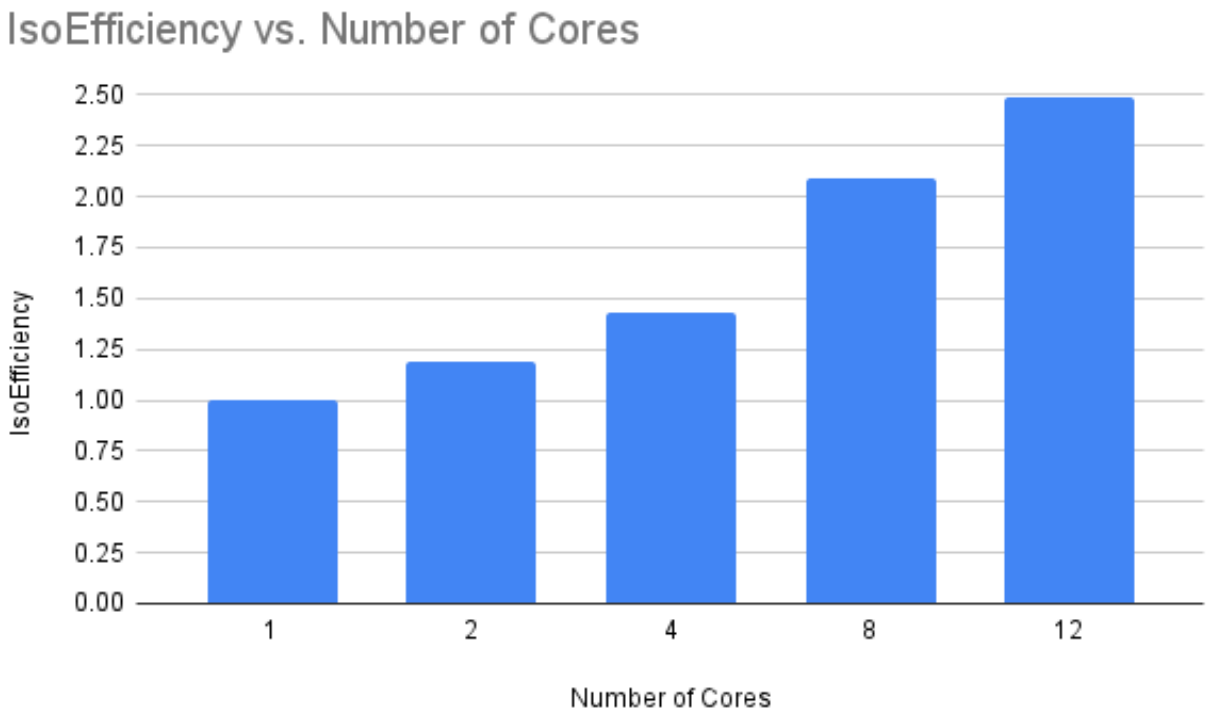
Number of Cores	Time Taken
1	143282
2	84986
4	51234
8	37511.4
12	29739.9

The speedup and Efficiency vs Number of Processors is given below:



Speedup and Efficiency vs Num of Cores

3 Iso-efficiency



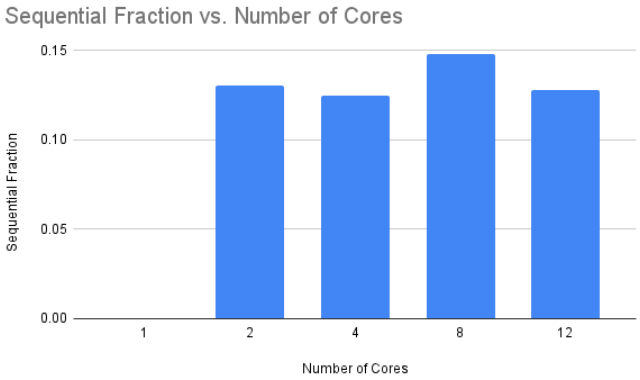
Iso-Efficiency vs Num of Cores

As you can see, the required problem size varies with the number of cores so that the efficiency remains as O(1).

This is due to the fact that the All-to-All function is a blocking call which does not get completed un-till all processes have made the call. So this becomes a blockage, but as we increase the problem size, the amount of time required by the processes to reach that part almost becomes the same, so the blocking call is called almost simultaneously by all the processes.

4 Sequential Fraction

Input, Pre-filtering and Output of our code remains a sequential fraction, other than that all the other parts of the codes can be parrallelized. Also, the All-to-All function also remains a sequential fraction for all the number of cores.



Sequential Fraction vs Num of Cores

5 Scalability

Our program is scalable as we can see that the code does not take a lot of time even when the input size is increased, and as input size is increased we can add more processors to the program to keep the efficiency as 1, so the amount of work is evenly divided between processors, so by making use of the Iso-efficiency coefficient for a given count of processor, we can make sure the program remains scalable, and we can process large amounts of data within a small period of time.

6 Task 2

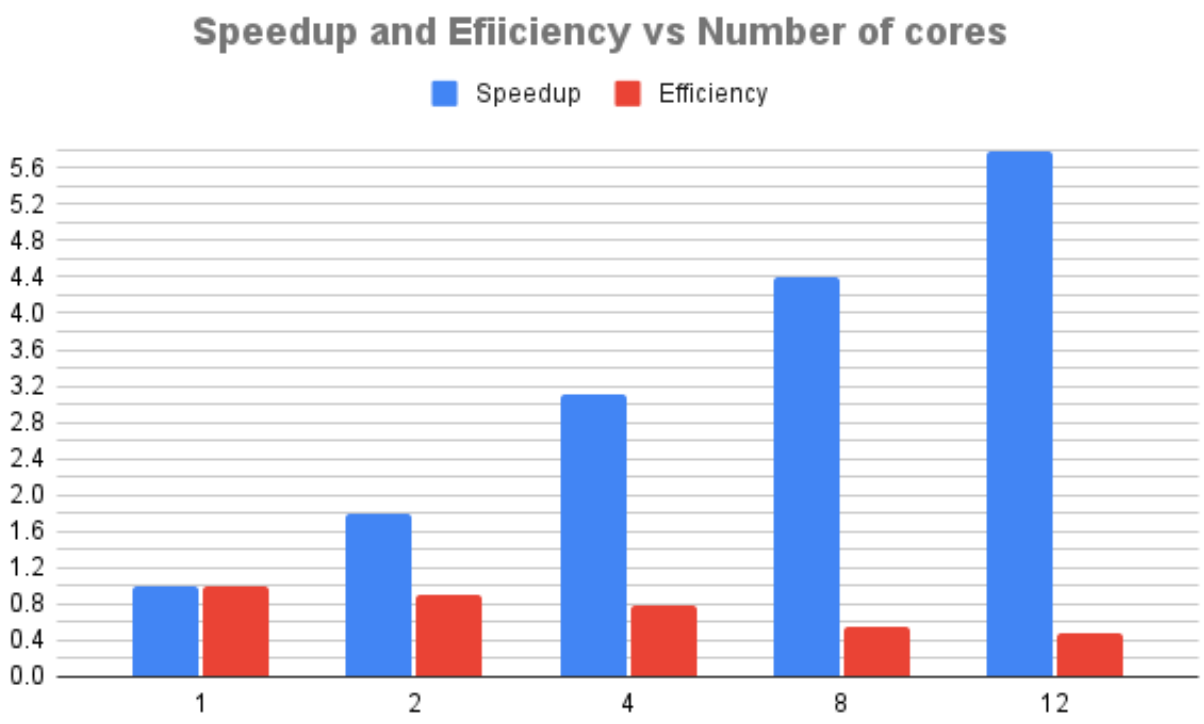
Our version of task 1, us the final adjacency graph, with all the edges that remain in a k-truss, so by putting `startk = endk` we can get the k-truss required.

We then use `dfs()` as we used in the `verbose=1` part of task one to find the different k-trusses. Now for each vertex `v` in the original graph, we see its neighbours, and check how many unique k-trusses they are part of, we don't count the same k-truss twice even if 2 neighbours are in the same k-truss. And if the final count of unique k-trusses we get is greater than or equal to the input `p`, we get an influential vertex. After checking this for all the vertex, we can get all the influential vertices.

Also, for `verbose=1` we just save the member vertices of the unique k-trusses, we find in the first part that the neighbour of an influential vertex is a part of, in a list and output that.

7 Speedup and Efficiency

The speedup and Efficiency vs Number of Processors is given below:



Speedup and Efficiency vs Num of Cores

8 Scalability

Our program is scalable just like task 1, as the basics remain the same, and only a small amount of extra computation is required. We can see that the code does not take a lot of time even when the input size is increased, and as input size is increased we can add more processors to the program to keep the efficiency as 1, so the amount of work is evenly divided between processors, so by making use of the Iso-efficiency coefficient for a given count of processor, we can make sure the program remains scalable, and we can process large amounts of data within a small period of time.