COEN 319 - Assignment 3 Report

Bhanu Prakash Naredla
1630571

I. Approach

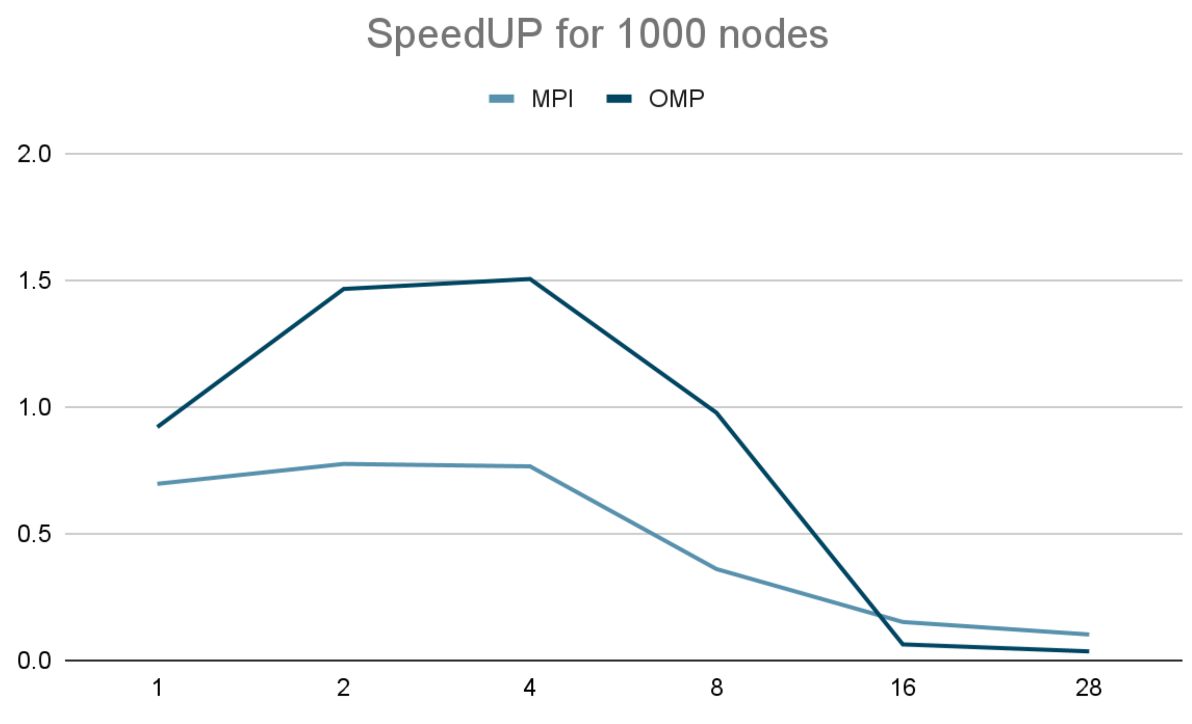
Parallelizing Dijkstra MPI Version

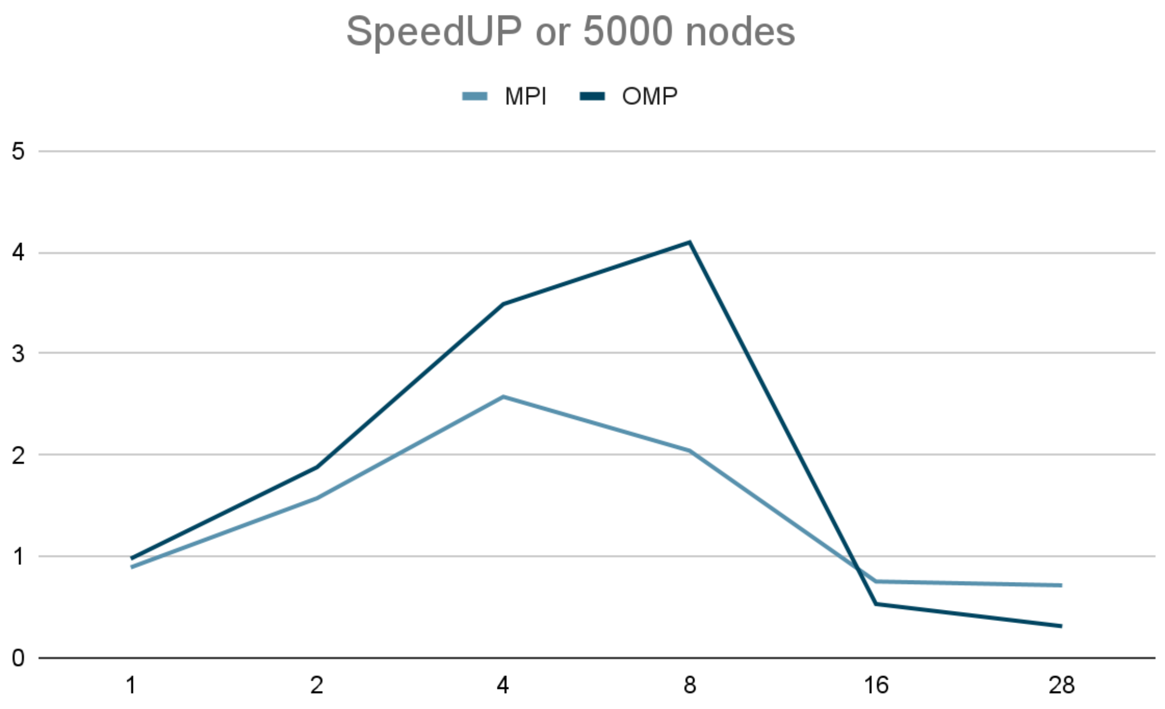
- ➔ As the full graph cannot be stored in a single process, I have splitted the graph equally among the number of processes and each process stores an equal number of nodes.
- ➔ Process 0 is responsible to read the adjacency matrix from the input file and send it to the corresponding process.
- ➔ Process 0 reads a part of the adjacency matrix from the input file and sends the data to its corresponding process with blocking MPI_Send. It first sends the graphInfo for a process which has totalNumNodes, StartingNodeId, EndingNodeId. Then, sends the part of adjacency matrix rows from StartingNodeId to EndingNodeId.
- ➔ All other processes receive the graphInfo and allocates memory to store their part of the adjacency matrix and receives adjacency matrix rows from StartingNodeId to EndingNodeId from process 0.
- ➔ Now all the process are ready to execute MPI version of dijkstra algorithm, which goes as follows
- ➔ Process 0 broadcasts sourceNode to every other process and all other processes receive sourceNode.
- ➔ Each processes initializes its distance array to store the shortest distance of each of its respective nodes from the sourceNode.
- ➔ Each process finds the node in its respective nodes which is not yet included in the shortest distance calculated set and also closest to the sourceNode. This information is sent to process 0.
- ➔ Process 0 receives the current shortest distance nodes from each process and computes the global minimum distance node and broadcasts it to every other process.
- ➔ Every other process receives the global minimum distance node and updates their current shortest distances from their respective nodes.
- ➔ Process which stores information of global minimum distance node Id will mark that node as shortest distance calculated.
- ➔ We repeat this until the shortest distance is calculated for every node.
- ➔ Now, each process sends their shortest distance array to process 0, which writes it to the output file.

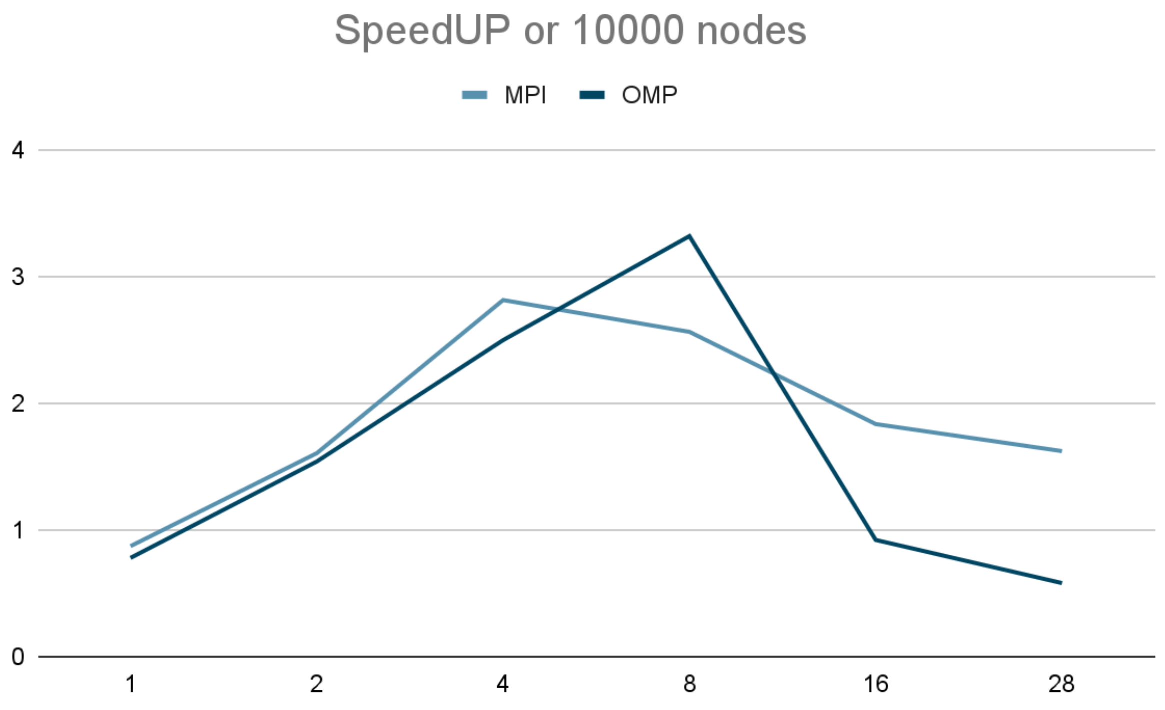
Parallelizing Dijkstra OMP Version

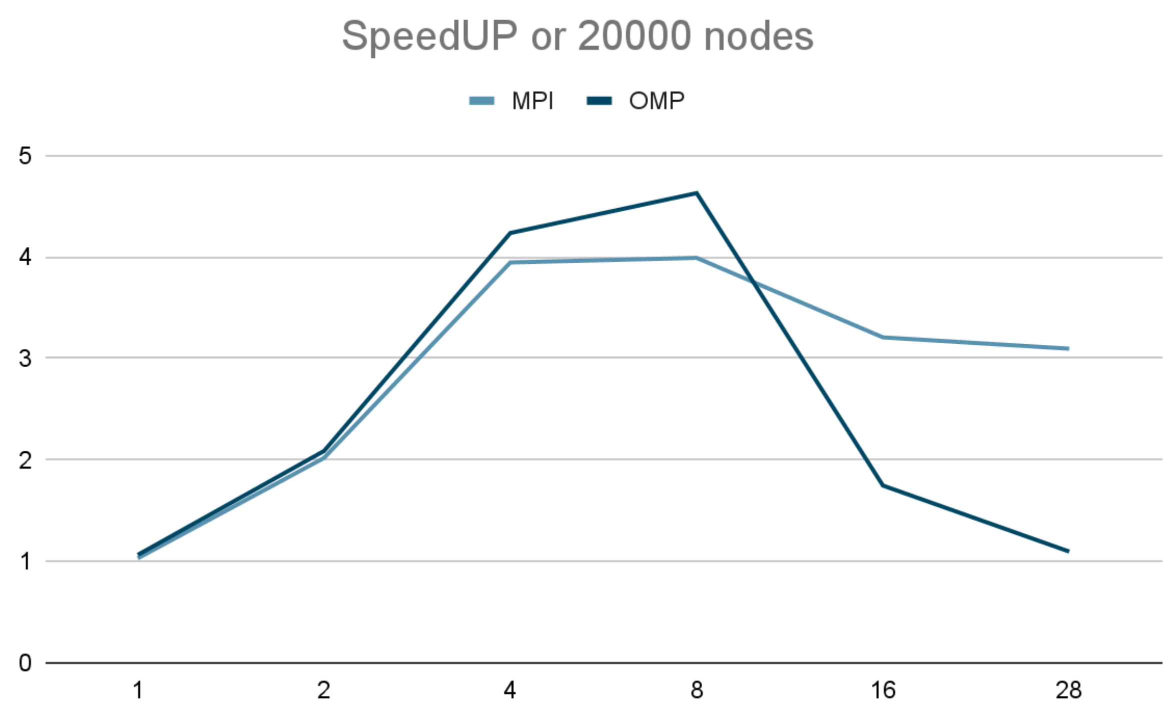
- ➔ Full adjacency matrix is read from file and stored and ready to perform dijkstra.
- ➔ Nodes are equally divided among the number of threads and each thread is responsible to compute any information from its respective nodes
- ➔ Each thread initializes the distance of its respective nodes from source in the shared distance array.
- ➔ Each thread computes the minimum distance node from its respective node and updates the global minimum distance where this update happens in a critical section using #pragma omp critical
- ➔ Once we got the global minimum distance node, any one of the thread marks it as shortest distance is calculated, used #pragma omp single
- ➔ Now, each thread updates the distances of its respective nodes in the shared distance array.
- ➔ Repeat until the shortest distance for all the nodes are calculated and write them to the output file.

II. SpeedUp Results:




SpeedUP or 5000 nodes
MPI
OMP
5
4
3
2
1
0
1
2
4
8
16
28


SpeedUP or 10000 nodes
MPI
OMP
4
3
2
1
0
1
2
4
8
16
28



III. Analysis of results

- ➔ For 1000 nodes, we didn't see much performance improvement in either of the OMP because the amount of work each thread does isn't significant and time taken to create these threads and context switch between threads will be an overhead for the program execution time.
- ➔ But, we got some performance improvement in the MPI version with a best speed up of ~1.5X with 4 processes.

- ➔ For 5000 nodes, we got the best performance in MPI version when using 4 processes and best performance in OMP version when using 8 threads. Best Serial time = 29.46 sec, best MPI time = 11.44 sec, best OMP time = 7.18 sec
- ➔ Best MPI speed up ~2.5X, Best OMP speed up ~4.2X
- ➔ For 10000 nodes, we got the best performance in MPI version when using 4 processes and best performance in OMP version when using 8 threads. Best Serial time = 117.56 sec, best MPI time = 41.79 sec, best OMP time = 35.43 sec
- ➔ Best MPI speed up ~2.8X, Best OMP speed up ~3.35X

- ➔ For 20000 nodes, we got the best performance in MPI version when using 8 processes and best performance in OMP version when using 8 threads. Best Serial time = 459.19 sec, best MPI time = 116.38 sec, best OMP time = 99.2 sec
- ➔ Best MPI speed up ~4.05X, Best OMP speed up ~4.6X

- ➔ If we observe in the test results as the number of nodes increases, we got increased overall speed up in both MPI and OMP versions.
- ➔ In most of the test results, we can observe that the OMP version gave slightly better performance than the MPI version because in the MPI version we are exchanging data between processes and synchronizing between them at some points in the algorithm, which affected the performance slightly.
- ➔ Moreover, the cost of creating the thread is far less when compared to the cost of creating a process. Also, the context switch between threads is much cheaper when compared to context switching between the processes which affected the performance.
- ➔ In the OMP version/ MPI version, With an increasing number of threads/processes we cannot guarantee that our program will run faster because it all depends on how many cores does the machine have and those many threads can run in parallel to improve performance.
- ➔ If we keep increasing the number of threads/ processes our program runs on, we may end up slowing it down, as threads/processes are not free i.e., OS takes up some memory and time for creating and maintaining threads. If the amount of work each thread does isn't significant, time taken to create these threads/processes and context switch between threads/processes will be an overhead for the program execution time.