Parallelization of Single Source Shortest Path Algorithm (Dijkstra) and All Pair Shortest Path Algorithm (Floyd Warshall) in OpenMp and MPI

Project Overview

This project aimed to enhance the speed of the Dijkstra single-source shortest path and Floyd-Warshall all-pair shortest path algorithms through parallelization using OpenMP and MPI. The biggest challenge is the inherent sequential nature of these algorithms, leading to substantial data dependencies among threads and processors. Attempting to distribute the workload among threads or processes introduced communication overhead, potentially nullifying or even worsening the time complexity compared to sequential implementations. This write-up explores the methods employed, challenges faced, and outcomes achieved in the endeavor to parallelize the Dijkstra and Floyd-Warshall algorithms, with a focus on the delicate balance between parallelization and communication overhead and its impact on performance.

Decomposition Techniques used to achieve speedup (OpenMp).

In our implementations of Dijkstra's single-source shortest path algorithm and the Floyd-Warshall all-pairs shortest path algorithm, distinct decomposition techniques leveraging OpenMP directives have been employed to harness parallelism and achieve enhanced computational efficiency.

For the Dijkstra algorithm, a combination of data and task decomposition strategies was adopted. Data decomposition was realized through thread-based division, where each thread was assigned a specific range of vertices for computation. Task decomposition was facilitated by parallelizing the outer loop and synchronizing critical sections to update global minimum distance and vertex values. The **#pragma omp barrier** directive played a pivotal role in synchronizing threads at crucial points, ensuring correct updates to shared variables. Additionally, the use of the **#pragma omp single** directive allowed for exclusive execution of certain sections by a single thread, preventing data corruption in critical operations. This hybrid approach effectively distributed the workload among threads, balancing the computational load, communication overhead and mitigating the challenges posed by the sequential nature of the Dijkstra algorithm.

Conversely, the Floyd-Warshall algorithm exhibited a different approach, combining both data and task decomposition with dynamic scheduling. Data decomposition was achieved by parallelizing the outer loop over intermediate nodes, assigning each thread the responsibility of computing shortest paths for a subset of nodes. Task decomposition was further realized by parallelizing the inner loop over source nodes, with dynamic scheduling to enhance load balancing. The **#pragma omp parallel for** and **#pragma omp parallel** directives facilitated thread-based and loop-based parallelization, respectively. This combination of decomposition techniques, along with the careful management of shared data and private variables, contributed to the efficient parallel execution of the Floyd-Warshall algorithm, yielding notable speedup over its sequential counterpart.

Data Distribution and Collective Communication Techniques used to achieve speedup in Dijkstra (MPI).

In the MPI implementation of Dijkstra's single-source shortest path algorithm, the parallelization is achieved through a combination of data distribution and collective communication, leveraging MPI directives for efficient parallel execution across multiple processes. The code adopts a Single Program Multiple Data (SPMD) model, where each MPI process is responsible for computing the shortest paths for a subset of vertices.

The technique employed here is data distribution using the **MPI_Scatter** function, which efficiently distributes the locally stored weight matrix among the processes. This ensures that each process has access to the relevant portion of the graph, allowing them to independently compute the shortest paths for their assigned vertices. The subsequent use of **MPI_Gather** helps collect the local distance vectors from each process at the source process, consolidating the final result.

MPI functions like **MPI_Allreduce** and **MPI_Scatter** facilitate efficient communication and coordination among processes. The **MPI_Allreduce** function, used in the main loop of the Dijkstra algorithm, helps find the global minimum distance vertex across all processes, enabling synchronized updates to the distance vectors. Additionally, the **MPI_Scatter** and **MPI_Gather** functions efficiently distribute and collect data, minimizing communication overhead.

This MPI implementation demonstrates a notable speedup over its sequential counterpart by parallelizing the computation across multiple processes. The ability to distribute the workload and efficiently communicate intermediate results allows the algorithm to scale effectively with the number of processes, leading to improved performance on large-scale graphs. The combination of data distribution, collective communication, and synchronization using MPI functions contributes to the overall success of the parallel implementation, showcasing the potential for significant speedup in single-source shortest path computations.

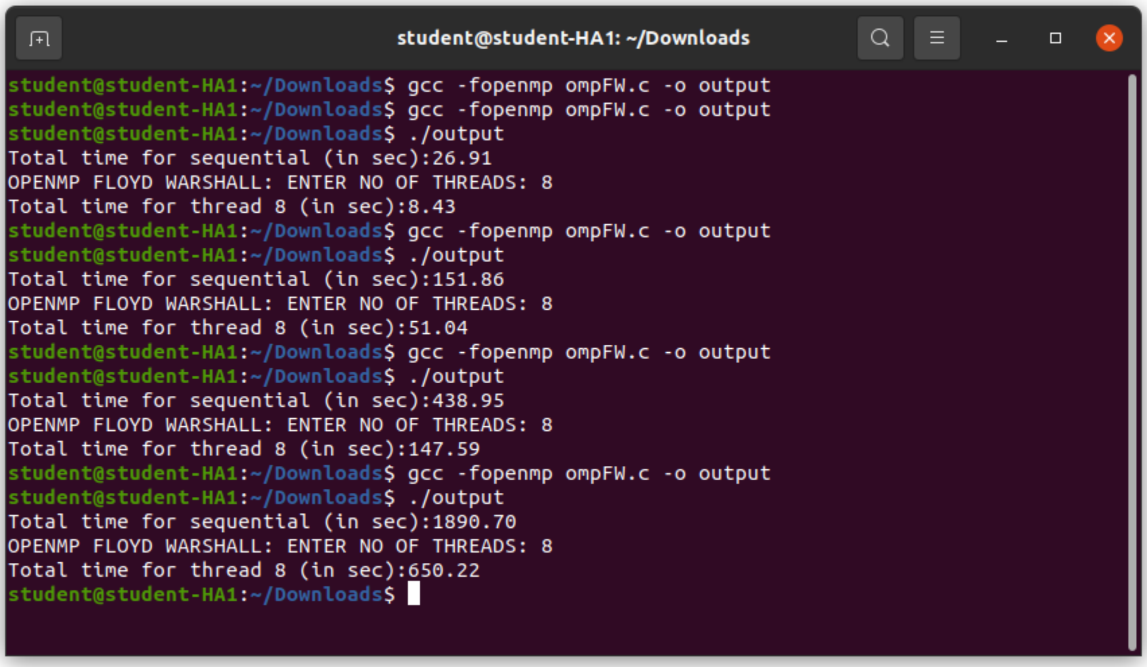
Data Generation and Test Run Screenshots.

In both the OpenMP and MPI implementations, the graph data is generated using the rand() function, which provides a random integer between 0 and rand()%100. This function is applied to initialize the weight matrix with random edge weights, ranging from 0 to 99 in the provided code. The randomness in edge weights ensures diverse graph structures for evaluating the performance and scalability of the parallelized Dijkstra and Floyd-Warshall algorithms**.**

Figure 01 shows multiple test run of openmp implementation of floyd warshall with various input sizes firstly 2000, then 3500, then 5000, and then 8000 nodes. With 8 threads, we can clearly observe the speedup is achieved as the parallel time taken is notably better than the time taken by the sequential time taken.

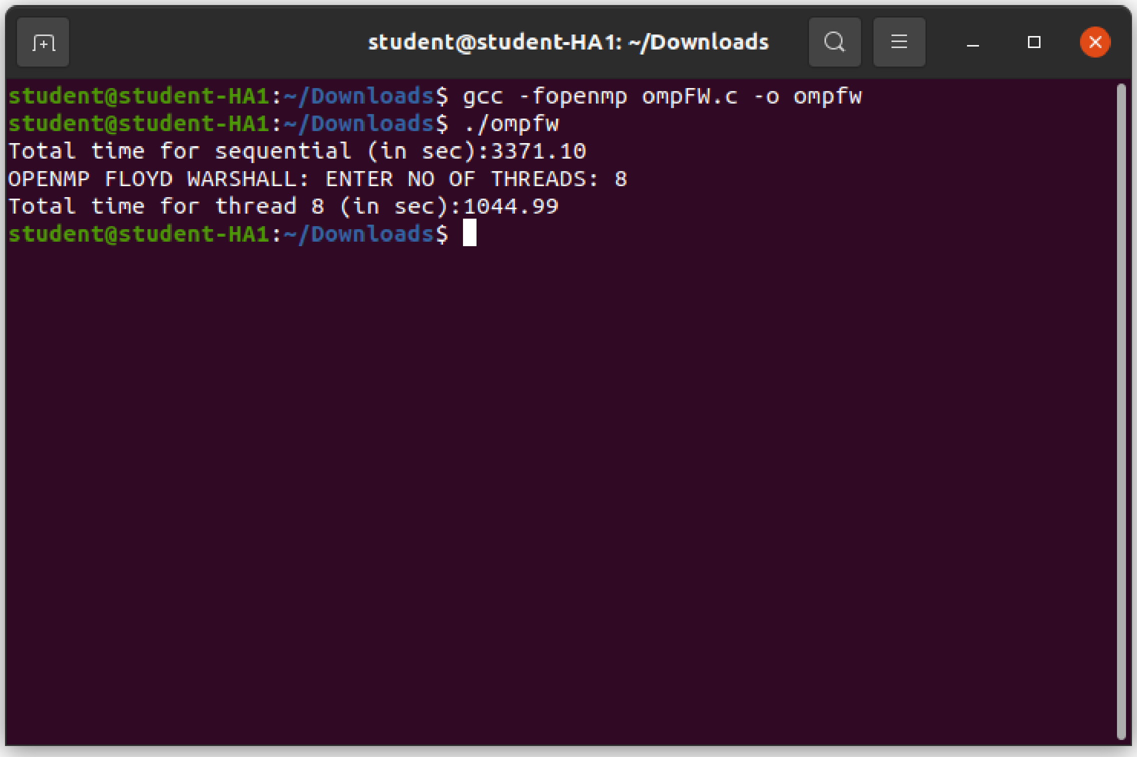
Finally, **Figure 02** shows a test run with 9500 Nodes as input and it took around **56 minutes** to run for the sequential code and as expected it achieved quite a speedup in the parallel time taken which is about **17.4 minutes**

Figure 01 (floyd warshall test run with 2000, then 3500, then 5000 and then 8000 nodes)



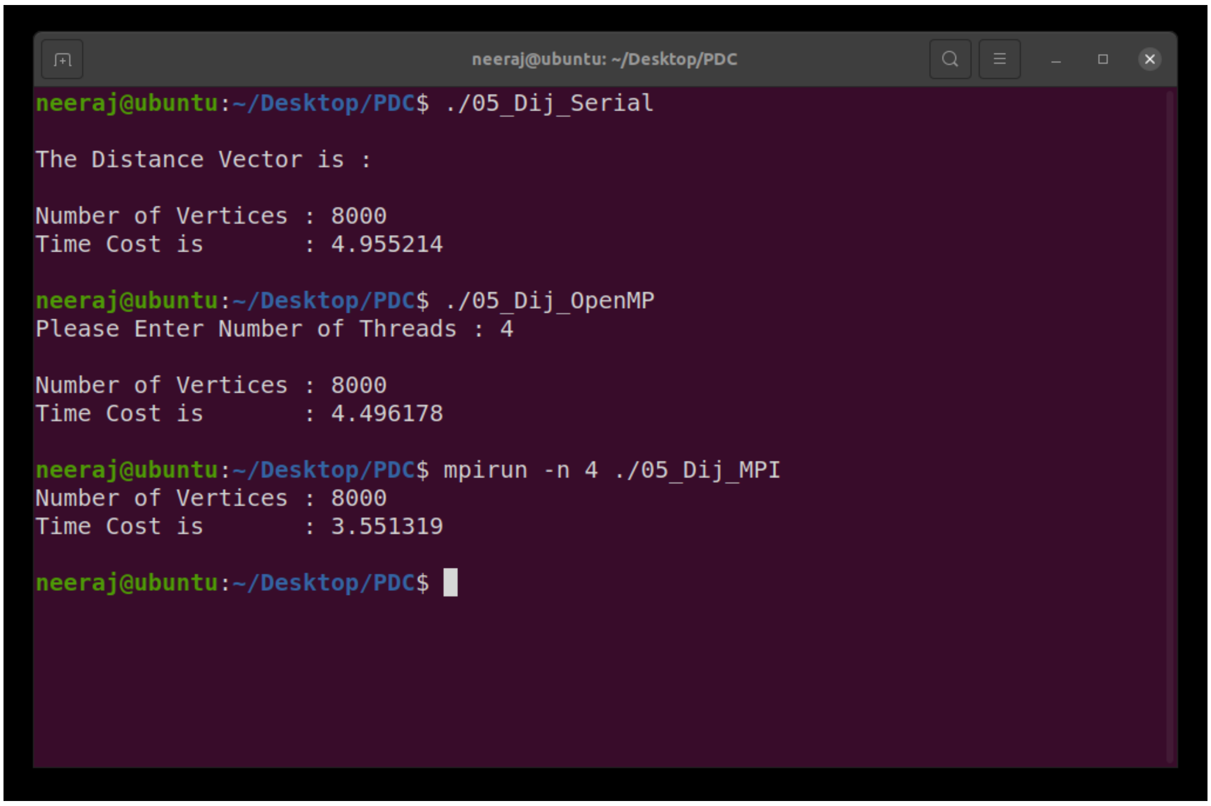
```
student@student-HA1:~/Downloads$ gcc -fopenmp ompFW.c -o output
student@student-HA1:~/Downloads$ gcc -fopenmp ompFW.c -o output
student@student-HA1:~/Downloads$ ./output
Total time for sequential (in sec):26.91
OPENMP FLOYD WARSHALL: ENTER NO OF THREADS: 8
Total time for thread 8 (in sec):8.43
student@student-HA1:~/Downloads$ gcc -fopenmp ompFW.c -o output
student@student-HA1:~/Downloads$ ./output
Total time for sequential (in sec):151.86
OPENMP FLOYD WARSHALL: ENTER NO OF THREADS: 8
Total time for thread 8 (in sec):51.04
student@student-HA1:~/Downloads$ gcc -fopenmp ompFW.c -o output
student@student-HA1:~/Downloads$ ./output
Total time for sequential (in sec):438.95
OPENMP FLOYD WARSHALL: ENTER NO OF THREADS: 8
Total time for thread 8 (in sec):147.59
student@student-HA1:~/Downloads$ gcc -fopenmp ompFW.c -o output
student@student-HA1:~/Downloads$ ./output
Total time for sequential (in sec):1890.70
OPENMP FLOYD WARSHALL: ENTER NO OF THREADS: 8
Total time for thread 8 (in sec):650.22
student@student-HA1:~/Downloads$
```

Figure 02 (floyd warshall test run with 9500 nodes, sequential time=56 minutes, parallel time = 17.4 minutes)



```
student@student-HA1:~/Downloads$ gcc -fopenmp ompFW.c -o ompfw
student@student-HA1:~/Downloads$ ./ompfw
Total time for sequential (in sec):3371.10
OPENMP FLOYD WARSHALL: ENTER NO OF THREADS: 8
Total time for thread 8 (in sec):1044.99
student@student-HA1:~/Downloads$
```

Figure 03 (Dijkstra Sequential + Openmp + Mpi Time Taken)



```
neeraj@ubuntu:~/Desktop/PDC$ ./05_Dij_Serial

The Distance Vector is :

Number of Vertices : 8000
Time Cost is       : 4.955214

neeraj@ubuntu:~/Desktop/PDC$ ./05_Dij_OpenMP
Please Enter Number of Threads : 4

Number of Vertices : 8000
Time Cost is       : 4.496178

neeraj@ubuntu:~/Desktop/PDC$ mpirun -n 4 ./05_Dij_MPI
Number of Vertices : 8000
Time Cost is       : 3.551319

neeraj@ubuntu:~/Desktop/PDC$
```