**Top K Shortest Path Problem with MPI & OpenMP:**

**Serial vs Parallel Analysis**

*Muhammad Anas Farooq, Daniyal Ahmed, Usman Azam*

*BS(CS)(2021), Fast NUCES University, Islamabad, Pakistan*

***Abstract---In this project, we use MPI (Message Passing Interface) and OpenMP (Open Multi-Processing) to solve the Top K Shortest Path Problem with parallel computing. The Top K Shortest Path Problem, which has applications in a variety of fields like network routing, telecommunication, and transportation, entails determining the K shortest pathways between a pair of nodes in a weighted graph. The objective of our methodology is to effectively investigate several possible routes concurrently by utilizing the computing capacity of shared memory systems (OpenMP) and distributed memory systems (MPI).***

**I. INTRODUCTION**

A basic problem in graph theory, the Top K Shortest Path Problem has many practical uses. Finding the K shortest paths between a given source and destination node pair is the goal given a weighted/un-weighted graph. These paths could stand for different routes that connect different places, each with varying prices or durations.

For the Top K Shortest Path Problem, conventional sequential methods might grow unnecessarily slow, especially when dealing with big graphs or important values of K. As a result, there is growing interest in parallel algorithms that make use of contemporary computer systems. In our project, we use OpenMP for shared memory parallelism and MPI for inter-process communication in distributed systems. We effectively accelerate K shortest path computations by distributing path exploration across numerous processes and threads by combining these techniques.

**II. SERIAL IMPLEMENTATION**

To identify the K shortest paths from a source vertex to a destination vertex, a graph's paths are explored in serial fashion in order to solve the Top K Shortest Path Problem. The procedures to accomplish this work effectively are laid out in the offered algorithm.

**Algorithm 1** K Shortest Path Problem Serial Implementation

**Input:** edges, n, m, k

Initialize graph G with n + 1 vertices

**for** each edge in edges do

Add the edge to G

**end for**

Initialize 2D array dis with dimensions (n + 1) x k, filled with infinity

Initialize priority queue pq with (0, 1)

Set dis[1][0] to 0

**while** pq is not empty do

u ← vertex with minimum distance in pq

d ← distance to u from the source

**if** dis[u][k - 1] < d then

Continue to the next iteration

**end if**

**for** each neighbor v of u do

dest ← v

cost ← cost of edge (u, v)

**if** d + cost < dis[dest][k - 1] then

dis[dest][k - 1] ← d + cost

Sort the k-th row of dis in non-decreasing order

Push (d + cost, dest) to pq

**end if**

**end for**

**end while**

In Algorithm 1, the problem first initializes a graph G with n + 1 vertices and a 2D array, dis, with dimensions (n + 1) x k, filled with infinity. Initialized with the source vertex and distance 0, pq is a priority queue. The method extracts vertices (u) and explores pathways iteratively across pq. Distances are compared with dis for each u to update in the event that a shorter path is discovered. Once all potential pathways have been investigated, the k-th row of dis is sorted, and the revised distances and destination vertices are sent to pq for more research. This produces the K shortest paths from the source to the destination vertex.

The time complexity for this the serial implementation of the k shortest path problem is O((N+M)\*KlogK).

**III. MOTIVATION TO PARALLELIZE**

Performance and scalability gains are substantial when employing OpenMP and MPI to parallelize the Top K shortest path challenge. The computation time can be greatly decreased by utilizing several processors and threads, particularly for big graphs and a high number of pairs for which shortest paths need to be found. By distributing the workload among several nodes, MPI enables scalability to effectively address large-scale issues. However, OpenMP allows for parallel processing on every node, which improves performance even further. By combining MPI with OpenMP, distributed and shared-memory architectures may be used effectively, optimizing resource use and reducing idle time. Therefore, the goal of our study is to apply this Top K Shortest Path method in many paradigms by using:

* MPI
* MPI with OpenMP

Additionally, to clarify the method's parallelizability, our analysis compares the times required by the serial algorithm and its different parallel versions.

**IV. SYSTEM SPECIFICATIONS FOR EXPERIMENTS**

All the code execution & timing are done on the following system. Tables I summarizes the specifications of the execution environment.

|  |  |
| --- | --- |
| **Architecture**  **CPU op-mode**  **CPU(s)**  **Thread per core**  **Model Name**  **CPU GHz**  **GPU Model**  **GPU Bus Type**  **GPU DMA Size** | X86\_64  64 bits  4  2  Intel(R) Core (TM) i5-8350U CPU @ 1.70GHz  1.90  Intel(R) HD Graphics 520  PCIe  32 bits |

TABLE I: Hardware Specification for Code Executing laptop.

**V. PARALLEL APPROACH**

**A. USING MPI**

MPI is typically used in the code for process communication and workload distribution. Every process independently manages a subset of pairings and determines the shortest pathways between them. To make sure every process has access to the relevant graph data, MPI\_Bcast is utilized to broadcast the number of nodes (N) and edges (M) to every process. To enable each process to build its own graph, MPI\_Bcast is used to broadcast the edge data (edges) to all processes. MPI\_Scatterv is used to distribute node pairs among processes so that each process handles a subset of pairs for shortest path computation. Following receipt of pairs and graph data, each process builds its own section of the graph and determines the shortest pathways for the pairs that it has been allotted. With no coordination needed during processing, this independent computing guarantees parallelism across processes.

TABLE II AND TABLE III Below show the comparison between the serial execution of the problem and parallel execution using MPI (**4-Processes**):

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset Name** | Email-EuAll | Email-Enron | Doctor  Who |
| **Average Execution Time** | 1.5810s | 1.0990s | 0.0370s |

TABLE II: Serial execution of the problem on different datasets

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset Name** | Email-EuAll | Email-Enron | Doctor  Who |
| **Average Execution Time** | 0.5496s | 0.3085s | 0.0058s |

TABLE III: Parallel execution of the problem with MPI (**4-Processes**) on different datasets

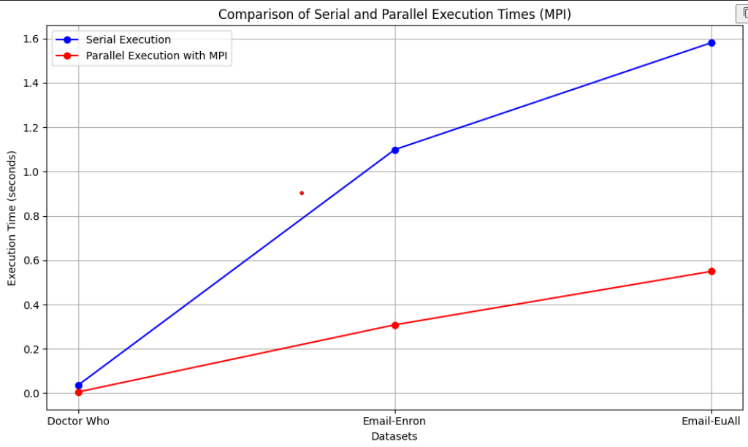


FIGURE 1: Comparison between the serial execution of the problem and parallel execution using MPI

Speedup = (Serial Execution Time / Parallel

Execution Time)

Speedup for Email-EuAll dataset =

1.5810/0.5496 = 2.875

Speedup for Email-Enron dataset =

1.0990/0.3085 = 3.561

Speedup for Doctor Who dataset =

0.0370/0.0058 = 6.379

All things considered, the MPI parallel execution yields encouraging results, cutting down on execution time when compared to serial execution on all datasets, proving that parallelization is a useful tool for expediting the solution of the Top K Shortest Path Problem.

**B. USING MPI WITH OPENMP**

Hybrid parallelism, which maximizes efficiency by utilizing both distributed and shared-memory parallelism, is made possible by the combination of MPI and OpenMP. While OpenMP optimizes intra-process parallelism for effective computing, MPI manages workload distribution between inter-processes. Each process uses OpenMP directives, like "#pragma omp parallel for," to parallelize the computation of shortest paths for numerous pairings. Each process uses several threads to execute shortest path computations concurrently by setting the necessary number of threads. OpenMP parallelizes computations inside each process to allow for the effective use of shared memory systems. A process's multiple threads can work together to find the shortest pathways for designated pairs by using shared memory for synchronization and communication.

TABLE IV below shows the execution time we got by using MPI with OpenMP and Figure 2 shows the comparison if all three-execution method, serial, parallel with MPI only, parallel with MPI and OpenMP (**4-Processes**):

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset Name** | Email-EuAll | Email-Enron | Doctor  Who |
| **Average Execution Time** | 0.5652s | 0.2362s | 0.0045s |

TABLE IV: Parallel execution of the problem with MPI with OpenMP (**4-Processes**) on different datasets

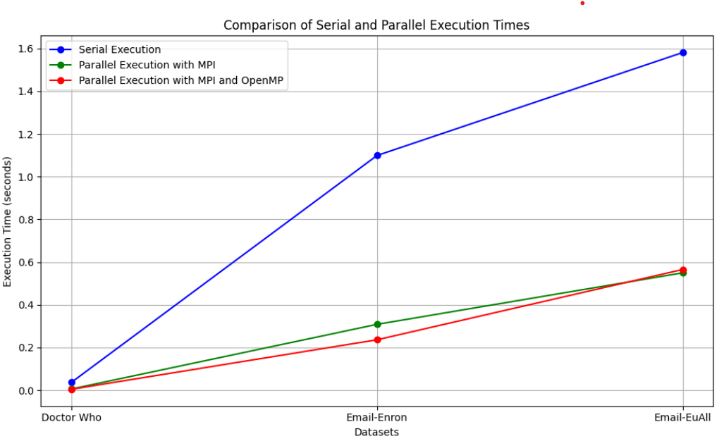


FIGURE 2: Comparison between the serial execution of the problem, parallel execution using MPI and parallel executing with MPI and OpenMP

Speedup = (Serial Execution Time / Parallel

Execution Time)

Speedup for Email-EuAll dataset =

1.5810/0.5652 = 2.795

Speedup for Email-Enron dataset =

1.0990/0.2362 = 4.652

Speedup for Doctor Who dataset =

0.0370/0.0045 = 8.222

Considering Table IV and Figure 2 and the Speedup we calculated we can conclude that MPI with OpenMP is better than serial execution of the problem but only using MPI is better than using MPI with OpenMP.

**VI. CONCLUSION**

The problem's serial execution produced the longest average execution times across all datasets. Longer execution durations result from its sequential based course, despite its simplicity of implementation. Execution times were significantly reduced when MPI was used for parallel execution as opposed to serial execution. It effectively distributes the burden among several processes by utilizing distributed memory systems. This led to a observable acceleration, particularly for bigger datasets such as "Email-EuAll" and "Email-Enron." By combining shared and distributed memory parallelism, MPI with OpenMP greatly improved the parallelization. compared to MPI alone, this hybrid technique performed much better, especially for smaller datasets like "Doctor Who." For larger datasets, the effect was less noticeable.

Overall, the size of the dataset and the available processing power determine which of the three options—serial, MPI, and MPI with OpenMP—to use. Serial execution could be sufficient for systems with constrained resources or for smaller datasets. Nevertheless, parallel execution with MPI or MPI with OpenMP turns out to be more effective for larger datasets and high-performance computing settings; in some circumstances, MPI with OpenMP even provides somewhat higher performance.

VI. REFERENCES

<https://www.geeksforgeeks.org/1st-to-kth-shortest-path-lengths-from-node-1-to-n-in-given-graph/>

Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems (Sol Ji Kang, 2015)