National University of Computer and Emerging Sciences

Parallel and Distributed Computing

Project

21I-0523 Sheraz Sadiq

21I-0586 Huzaifa Tahir

21I-0737 Ali Hassan

Project Statement: Top K Shortest Path Problem with MPI and OpenMP

The Kth Shortest Path Problem involves finding the Kth shortest path between two nodes in a graph. Unlike the standard shortest path problem, which aims to find the shortest path, the Kth shortest path problem seeks the Kth shortest path, which may not necessarily be unique.

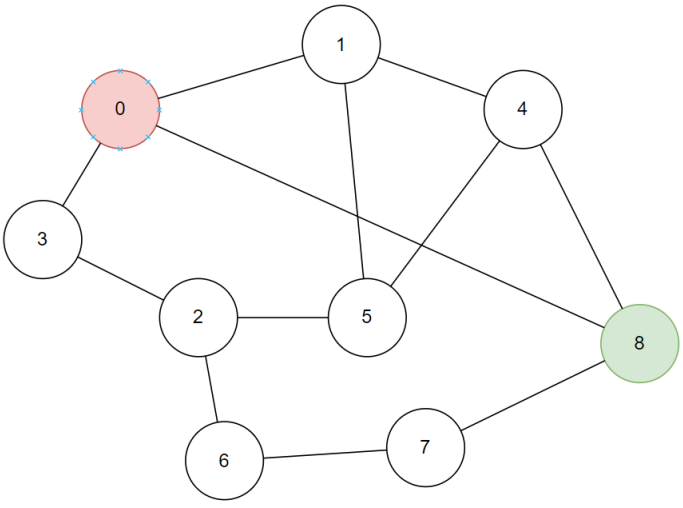
Experimental Setup

Preprocessing: The preprocessing step involved reading data from the provided CSV file, transforming it into appropriate data structures, such as arrays and matrices, and computing the necessary information for further processing. This included identifying unique cities, constructing an adjacency matrix, and initializing memoization tables.

> **Challenges Faced:** Ensuring correctness and efficiency in parsing the CSV data, handling edge cases such as duplicate cities, and efficiently constructing the adjacency matrix.

Implementation: The main implementation included algorithms for finding the shortest path and the kth shortest path between pairs of cities using parallel and distributed computing paradigms (OpenMP and MPI).

Let Consider this Graph



In our implementation, the *shortestPath()* function utilizes **Dijkstra's algorithm** to find the shortest path between two nodes in a graph. This algorithm efficiently computes the shortest paths from a single source node to all other nodes in the graph, providing a reliable method for determining distances between pairs of nodes.

Within the *findKthPath()* function, we call *shortestPath()* three times, each with different source and destination nodes. By doing so, you aim to explore different routes and connections within the graph, ultimately contributing to the computation of the kth shortest paths between the given source and destination pairs.

For the above example if we want distance between 0 and 8 Node, function is called as

- Source 0, Destination 8
- Source 0, Destination 1 + Source 1, Destination 8
- Source 0, Destination 3 + Source 3, Destination 8
- And so, on

By calculating the shortest path cost between each node, we add them and find the actual shortest path between 2 nodes.

Also used memoization table to backtrack the previous paths that we found during the execution of Dijkstra's algorithm. This allows us to reconstruct the shortest path between any pair of nodes efficiently, enabling us to not only compute the shortest distances but also retrieve the actual path taken to reach each destination. By storing the parent nodes along with the shortest distances in the memoization table, we facilitate the reconstruction of paths, contributing to the overall functionality and versatility of the algorithm.

> **Challenges Faced:** Implementing parallel and distributed algorithms while ensuring data consistency and minimizing communication overhead between processes. Optimizing parallel loops and task distribution to maximize performance.

Testing:

Testing involved running the implemented algorithms on different configurations, including varying numbers of processes and different input graph sizes.

> **Challenges Faced:** Ensuring correctness and performance scalability across different configurations, debugging synchronization issues in parallel and distributed environments.

Optimizations Applied:

1. **Parallelization:** Utilized OpenMP directives to parallelize loops and tasks, distributing computation across multiple threads to leverage multi-core processors effectively.
2. **Distributed Computing:** Employed MPI for distributed computing, allowing communication and coordination between multiple processes running on different nodes in a cluster.
3. **Memoization:** Implemented memoization techniques to cache previously computed results, reducing redundant computations and improving overall performance.
4. **Task Granularity:** Optimized task granularity in parallel loops to balance workload distribution and minimize overhead, enhancing parallel efficiency.

Experimental Results:

Run 1: Execution Time: 3.631735 seconds, K = 10

Source	Destination	K Paths	Execution Time
137	107	3 2 3 3 3 3 4 4 4 3	0.212314 seconds
257	561	4 4 4 4 4 4 4 4 4 4	0.625538 seconds
137	470	5 5 5 5 5 5 5 5 5 5	0.722760 seconds
261	282	4 4 4 5 5 5 5 5 4 5	1.081989 seconds
67	13	4 4 4 4 4 4 4 4 4 4	0.989134 seconds

Run 2: Execution Time: 2.293995 seconds, K = 3

Source	Destination	K Paths	Execution Time
648	595	5 6 6	0.197343 seconds
62	180	3 3 4	0.376711 seconds
451	105	4 3 5	0.598044 seconds
393	103	4 4 5	0.528263 seconds
613	659	3 4 4	0.593634 seconds

Run 1

```
└─$ mpiexec -n 5 ./mpi
{137 ,107}
{257 ,561}
{137 ,470}
{261 ,282}
{67 ,13}
Shortest K(10) Paths: 3 2 3 3 3 3 4 4 4 3
Execution time: 0.212314 seconds

Shortest K(10) Paths: 4 4 4 4 4 4 4 4 4 4
Execution time: 0.625538 seconds

Shortest K(10) Paths: 5 5 5 5 5 5 5 5 5 5
Execution time: 0.722760 seconds

Shortest K(10) Paths: 4 4 4 5 5 5 5 5 4 5
Execution time: 1.081989 seconds

Shortest K(10) Paths: 4 4 4 4 4 4 4 4 4 4
Execution time: 0.989134 seconds
```

Run 2

```
└─$ mpiexec -n 5 ./mpi
{648 ,595}
{62 ,180}
{451 ,105}
{393 ,103}
{613 ,659}
Shortest K(3) Paths: 5 6 6
Execution time: 0.197343 seconds

Shortest K(3) Paths: 3 3 4
Execution time: 0.376711 seconds

Shortest K(3) Paths: 4 3 5
Execution time: 0.598044 seconds

Shortest K(3) Paths: 4 4 5
Execution time: 0.528263 seconds

Shortest K(3) Paths: 3 4 4
Execution time: 0.593634 seconds
```

Insights and Analysis:

1. Parallel and distributed computing techniques effectively reduced computation time for finding shortest paths and kth shortest paths between cities.
2. Optimal performance was achieved by balancing workload distribution, minimizing communication overhead, and optimizing task granularity.
3. Scalability was observed with increasing numbers of processes, but diminishing returns were evident beyond a certain point, highlighting the importance of efficient parallelization strategies.
4. Using MPI to divide and then map code as different processes is an effective strategy and significantly reduces execution time, as demonstrated in the above experiment, where we created a SIMD model, where although the adjacency matrix was same, but the source and destination nodes for each process was different, whereas the instructions for each concurrent process were the same. In conclusion, using mpi parallelization for such models demonstrates the true essence of parallelization and its benefits.
5. Overall, the implementation demonstrated the effectiveness of parallel and distributed computing paradigms in solving large-scale graph problems efficiently.