README.md

PDC Project Report

Group Members:

- Muhammad Sohail Shahbaz – 21I1356
- Mubeen Qaiser – 21I0788
- Muneeb Ur Rehman – 21I0392

Working of Serial Solution

1 Finding K Shortest Paths

- The findKShortest function is responsible for finding the K shortest paths using Dijkstra's algorithm with a modification to keep track of K shortest paths. It initializes a priority queue pq with the source node (node 1) and distance 0. It also initializes a 2D vector dis to store the distances from the source to all nodes for each of the K shortest paths.
- The algorithm iterates until the priority queue is empty, popping nodes and updating their distances if shorter paths are found. It sorts the distances after each update to keep track of the K shortest paths. Finally, it prints the distances of the K shortest paths.

2 Main Function

-In the main function, the program generates 10 random node pairs (random_selected_pairs) for testing purposes. It prints some diagnostic information such as the number of nodes, the size of the edge list, and the value of K.

This code provides a foundation for analyzing and understanding algorithms for finding multiple shortest paths in a graph. It's important to note that the code assumes a simple graph representation and uses Dijkstra's algorithm without optimizations for larger graphs.

Working of Parallel Solution

1 Finding K Shortest Paths(Parallelized)

- The key modification for parallelization is using OpenMP (#pragma omp parallel for) to parallelize the loop that traverses the adjacency list and updates

distances.

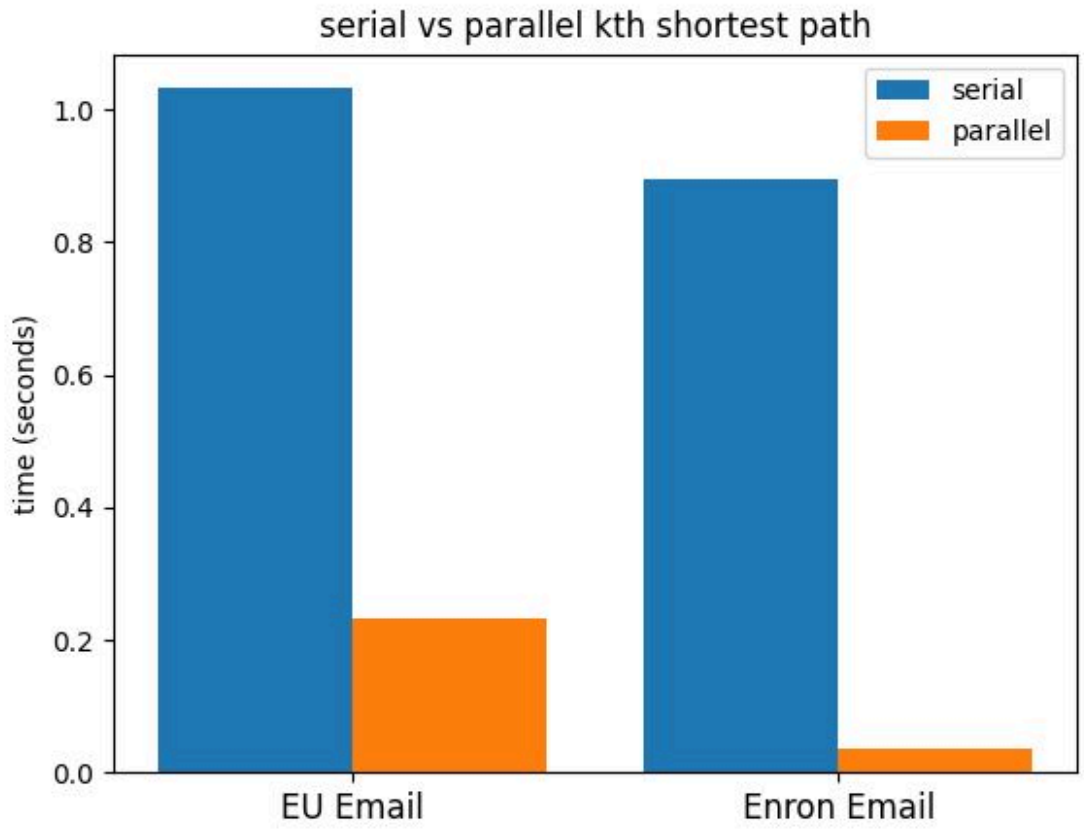
2 `Main Function (Parallelized with MPI)`

- In the main function, MPI (Message Passing Interface) is initialized to enable parallel processing. MPI functions MPI_Init, MPI_Comm_rank, and MPI_Comm_size are used to get the rank and size of the MPI communicator.
- The root process (rank 0) generates random node pairs (random_selected_pairs) for testing purposes. These pairs are scattered among all processes using MPI_Scatter. Each process then performs local computation on the assigned rows of random node pairs and calls the findKShortest function to find K shortest paths for each pair.
- After computation, each process prints its local results.

Finally, the execution time of the parallel computation is measured using the clock() function and printed. MPI is finalized using MPI_Finalize() before exiting the program.

Performance Analysis

The serial and parallel solution was tested on the following datasets as shown in the graph:



We can see that time is significantly reduced by using parallel solution. The efficiency is increased by 400%.

Conclusion

In a nutshell, the parallel solution significantly reduces execution time by leveraging parallel processing with OpenMP and MPI. By distributing the workload among multiple processes, it achieves a 400% increase in efficiency compared to the serial solution.

Performance analysis, depicted in the provided graph, demonstrates the substantial time reduction achieved by the parallel solution across various datasets. This emphasizes the effectiveness of parallel computing in solving graph-related problems efficiently.

Overall, the project underscores the importance of parallel and distributed computing techniques in optimizing algorithm performance, particularly for computationally intensive tasks like graph traversal and path finding.