Project Report

Parallel and Distributed Computing

Ahmad Farhan – i211366
Hasan Kamal – i210694
Section: A

Introduction

Background

Kth Shortest Path Problem:

The Kth Shortest Path Problem is a generalization of the classic Shortest Path Problem in graph theory. Instead of finding the single shortest path between two nodes in a graph, the Kth Shortest Path Problem aims to find the Kth shortest path between the same pair of nodes. This problem was formally introduced by Edger W. Dijkstra in his seminal paper "A note on two connexion with graphs" which also touches upon the concept of finding multiple shortest paths, laying the groundwork for the Kth Shortest Path Problem.

Mathematically, given a weighted graph $G(V,E)$ with nodes V and edges E, and two distinct nodes s (source) and t (destination), the Kth Shortest Path Problem seeks to find the Kth shortest path from s to t in terms of path length.

Top K Shortest Path Problem:

Top K shortest Path Problem is a further generalization of the Kth shortest path problem, wherein we find the top 1^{st} to k^{th} shortest paths between two nodes in a weighted graph.

Objectives

Following are the objectives of this paper:

1. Implement a serial version of the top k shortest path problem.
2. Design and implement a parallelized version of the same algorithm.
3. Perform a comparative analysis of the two versions to identify performance improvements.

The size and complexity of modern-day graph structures has grown exponentially leading to the need for parallel versions of graph algorithms. Parallelization can significantly reduce the execution times by distributing the workload across multiple processing units simultaneously, allowing for faster computation of the Kth shortest path for large graphs and multiple queries. This parallel approach would enable efficient utilization of resources, scalability, and improved performance.

Experimental Setup

Environment Details

The experiments were conducted on a virtual computing cluster on a single machine with the following hardware specifications:

- **Processor**: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.42GHz (4 cores, 8 threads)
- **Memory**: 8.00 GB DDR4 RAM
- **Storage**: 500 GB NVMe SSD

The software environment used for the experiments was as follows:

- **Operating System**: Ubuntu 20.04 LTS
- **Compiler**: GCC 11.2.0
- **Programming Language**: C++ (Version: C++17)

Tools and Libraries

The implementation utilized the following libraries, tools, and frameworks:

- **Standard Libraries**
    - <unordered_map>
    - <unordered_set>
    - <algorithm>
    - <iostream>
    - <sstream>
    - <fstream>
    - <chrono>
    - <vector>
    - <queue>
    - <array>
- **Parallelization Libraries**
    - OpenMP (Version 5.2): For shared memory parallelization
    - MPI (Version 3.3.2): For distributed memory parallelization

Methodology

Preprocessing

Two input file types are accepted: csv and txt. The input format and processing for each is as:

Preprocessing txt Files:

- **Input Format:** Two columns (fromNode, toNode) describing unweighted graph edges.
- **Process Step:** If no weights column, add a weights column with default edge weight of 1.

Preprocessing csv Files:

- **Input Format:** Four Columns as (Source,Target,Weight,Type) describing graph edges.
- **Node Encoding:** Unique String Nodes are encoded into increasing integer numbers.
- **Edge Encoding:** Use node codes to encode edge list into (Source, Target, Weight) format.
- **Edge Sorting:** Sort the encoded edge data based on source node codes for efficiency.
- **Generate Codes File:** Save the String-Integer node encoding into a file with -enc extension.

Output Files:

- **Formatted Edge File:** <original filename> -grf.txt
- **Codes: <**original filename> -enc.txt
- **Source Destination Pairs:** <original filename>-sdn.txt

Serial Implementation Details

Customized Dijkstra's Algorithm:

1. Explore Shortest Paths to Nodes:
   Each iteration focuses on exploring the shortest paths to the nodes connected to the current node under consideration.
2. Update Shortest Paths and Costs:
   For each neighboring node, if a shorter path (including the current node) is found, it updates the shortest path and its cost in the distance vector.
3. Maintain K Shortest Paths:
   After updating the costs for a node's neighbors, it sorts the paths based on their costs to maintain the K shortest paths in the distance vector.

The idea behind these steps is to efficiently find and maintain the K shortest paths from the source to the destination by exploring the graph's nodes in an order that prioritizes shorter paths. The distance vector provides a memoization-like approach. In addition to this, the priority queue ensures that nodes with shorter paths are processed first, leading to more efficient discovery of the K shortest paths.

Other Potential Approaches:

1. **Yen's Algorithm:** A modification of Dijkstra's algorithm to find the Kth shortest path. After finding the shortest path from source to destination, the algorithm finds all possible deviations from the already found paths. This repeated computation of shortest paths and constantly updating graph to remove already found paths reduces the efficiency of the algorithm and introduces increased computational overheads.
2. ***A* Algorithm***: An informed search algorithm that can be adapted to find multiple shortest paths. Considering that the source and destination can be any pair within the graph, we would potentially need n*n heuristic values for the algorithm to work given any node pair. Considering our graphs can range up to 256000 nodes, so just the heuristics file would end up being 50+ GBs. Thus, the algorithm is not feasible for our use case.
3. **Eppstein's Algorithm:** Potential pruning-based Dijkstra's algorithm to find k shortest paths. Pruning in the algorithm helps reduce the number of paths explored, making the algorithm more efficient in finding the k shortest paths. However, pruning may be based on a computationally intensive operation which can introduce overheads, in dense graphs.

Parallelization Implementation Details

OpenMP:

We can parallelize the updating of shortest path information for each neighboring node of the current node if a shorter path is found. The updated path is pushed into a priority queue in a thread-safe manner using "#pragma omp critical". Note that if number of neighboring nodes is less than 3, the overheads of creating and managing threads might outweigh the benefits gained from parallel execution. Thus, in such case we execute the loop sequentially, otherwise we use 4 threads.

MPI:

Cluster can be utilized effectively by parallelizing the computation of k shortest paths for different source and destination pairs. As we are restricted to computing k shortest paths for 10 source-destination vertex pairs, therefore we can provide our cluster nodes with a copy of the graph using broadcasting, followed by assignment of source-destination node pairs.

Testing Strategy

Customizable Parameters:

1. Graph Data File: Graph input file.
2. Source-Destination Nodes: Node Pairs for path finding.
3. Number of Processes: Configures number of MPI processes.
4. Value of k: Number of shortest paths to be computed.

Restricted Parameter:

1. Iterations (Pre-defined as 10 pairs)

For each graph file, we use preprocessing to generate 10 pairs of source-destination nodes that are significantly spaced apart based on node number. A single test case is executed as:

1. Serial: ./serial.exe filename-grf.txt <k-value> <encoded output>
2. Parallel: mpiexec -np num_proc ./parallel.exe filename-grf.txt <k-value> <encoded-output>

Using a bash script, we perform 5 executions per test case, for both serial and parallel versions with consistent parameters for graph file, node pairs, and k-values.

1. Serial: Tested with k-value in range 2 to 5
2. Parallel: Tested with k-value in range 2-5.
3. Setting k = 3, we again test parallel version with number of processes in range 2-5

Challenges

Preprocessing Challenges

1. **File Format Handling:**
   Challenge: Managing different file formats (.txt and .csv) required different parsing and processing methods.
   Solution: Implemented separate preprocessing functions for .txt and .csv files to handle their specific formats.
2. **Metadata Extraction:**
   Challenge: Extracting and handling metadata (number of nodes, edges) from .txt files.
   Solution: Developed custom functions to extract and utilize metadata for initializing graphs. Encoded Graph files also included graph meta data for initializing graphs.
3. **Node Encoding**
   Challenge: Encoding unique node names into integer codes for efficient graph representation and processing.
   Solution: Implemented node encoding algorithms to map node names to unique integer codes, facilitating faster graph operations.
4. **Edge File Formatting**
   Challenge: Standardizing edge data from .txt and .csv files into a consistent format for algorithmic processing. Files had to include meta data for efficient graph initialization.
   Solution: Formulated algorithms to format and structure edge data uniformly, ensuring compatibility with graph algorithms.

Implementation:

1. **Serial K shortest Paths:**
   a. Designing optimized data structures to store and manage multiple shortest paths
   b. Minimizing overheads to ensure maximized efficiency of computation.
2. **Parallel k shortest Paths:**
   a. Assignment of source-destination node pairs.
   b. Managing synchronization between parallel processes to ensure data integrity.
   c. Minimizing load imbalance by assigning each process equal workload.
   d. Designing optimized data structures to store and manage multiple shortest paths.
   e. Minimizing need for communication to reduce communication latencies.
3. **Graph Sharing:**
   a. Efficiently distributing graph data across multiple nodes without overloading the system since. Too many items being sent and received can create a bottleneck.
   b. Ensuring consistency of shared graph data across all nodes.

Testing Challenges:

1. Number of Customizable Parameters
2. Number of Testcases needed for analysis.
3. Single Machine limits accurate benchmarking.
4. Background processes running at the time of testing.

Note that the performance metrics provided are not from executing the code on a Beowulf Cluster. That method introduced significant overheads making the parallel version slower than the serial version. These overheads were a result of setting up the cluster using virtual machines on a single machine. As such, all measurements of parallel version are performed using a single machine with multiple mpi processes.

Optimizations Applied:

1. **Graph initialization**: Adjacency list vector size initialized with number of nodes in graph to eliminate the need for reallocation of memory when appending edges.
2. **Meta Data**: Number of nodes added as meta data to preprocessed graph files.
3. **Graph Sharing**: Sharing done using multiple MPI broadcast operations of 1 variable at a time to not overwhelm MPI system buffers leading to bottlenecks.
4. **Non-blocking Send**: Assignment of source-destination pairs is done using MPI_Isend.
5. **MPI_Reduce**: Used MPI_reduce to combine time metrics calculation for benchmarking.
6. **Memoization**: Distance vector maintains k shortest paths so far, with their costs calculated, to all other nodes at each iteration.
7. **Thread Management Overheads**: Minimized thread creation and management overheads by restricting the number of threads to 4. Threads are only created when the current vertex has more than 3 neighboring vertices, ensuring efficient utilization of parallel processing.

Experimental Results

Performance Metrics

1. Serial Execution Time

The execution times of the serial version increase almost linearly as the value of k increases. As one would expect, the execution time is also large on larger graph. Email-Enron has 36692 vertices whereas Email-EuAll has significantly more vertices i.e., 265214, yet the execution times on Email-Enron dataset are higher than on Email-EuAll dataset.

Note: All time measurements are in milliseconds(ms).

	new-who		classic-who		doctorwho		Email-Enron		Email-EuAll	
K	**total**	**avg**	**total**	**avg**	**total**	**avg**	**total**	**avg**	**total**	**avg**
2	88	8.2	97	9.2	182	17.9	9436	945.9	9742	973.7
3	159	15.2	176	17.1	339	33.6	17074	1707.1	15520	1551.4
4	244	24.1	280	27.5	520	51.6	25855	2585.1	22299	2229.3
5	353	34.7	391	38.4	752	74.6	36589	3658.2	29769	2976.5

Thus, we conclude:

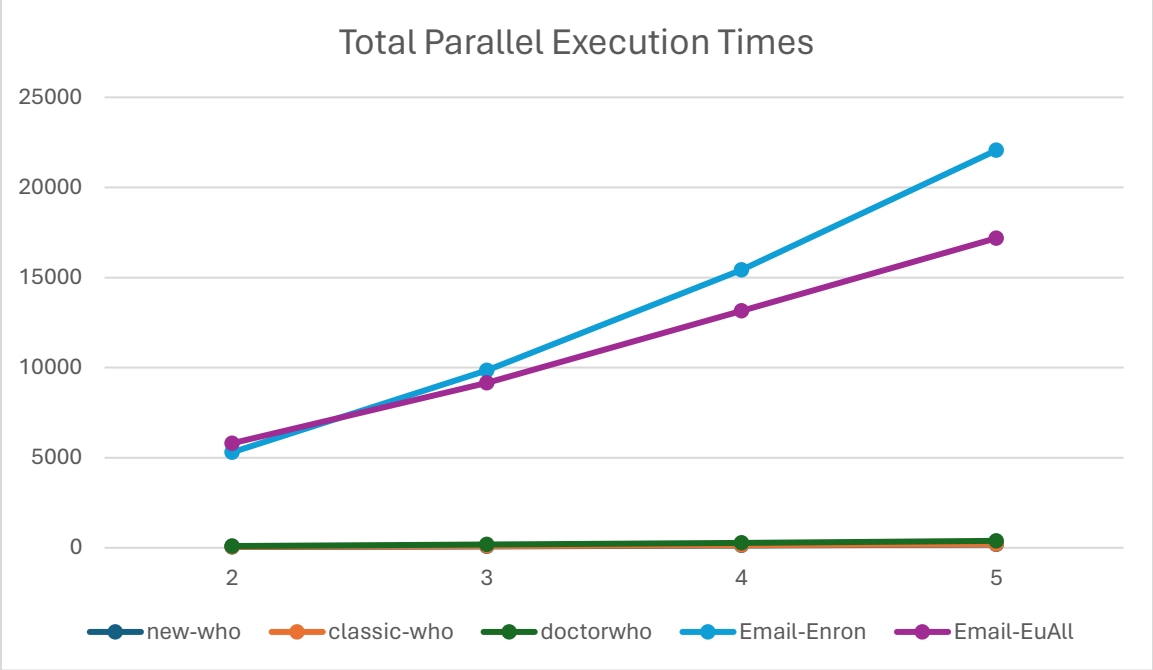
1. Execution times increase with the increase in graph density.
2. Execution times increase with the increase in k-value.

2. Parallel Execution Time (np = 2)

Total execution times for parallel version show a similar performance curve however, Email-EuAll has a more linear line whereas Email-Enron has more of an exponential curve. It may also be noted that the overall execution times of parallel version are lower than those of serial version.

Note: Number of processes of parallel execution is set to 2.

	new-who		classic-who		doctorwho		Email-Enron		Email-EuAll	
K	**total**	**avg**	**total**	**avg**	**total**	**avg**	**total**	**avg**	**total**	**avg**
2	38	3.5	55	4.1	98	7.3	5295	305.1	5798	519.6
3	82	6.5	93	7.4	179	13.7	9837	571.3	9150	861.9
4	125	9.8	140	10.6	274	21.4	15418	885.9	13144	1247
5	180	14.4	200	15.6	384	29.9	22059	1303	17182	1656



Thus, we conclude:

1. Overall pattern of total execution times between serial and parallel version is similar.
2. The parallel version has lower total execution times i.e., is faster.
3. Individual executions of parallel version are also faster than their serial counterparts.

Comparative Analysis:

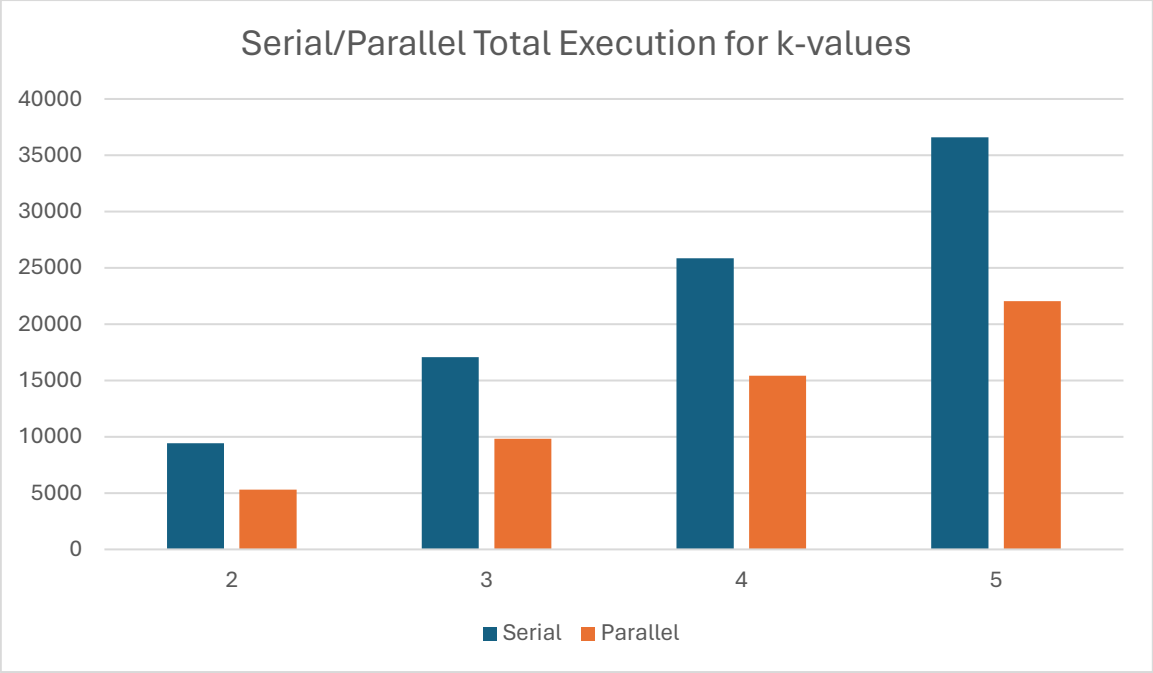
1. Total Execution Time

Total execution times for serial and parallel versions increase almost linearly as the value of k increases. The curve of parallel executions is more linear than that of parallel executions, thereby suggesting better scalability of parallel version based on k-values.

Note: Number of processes of parallel execution is set to 2.

Total Serial-Parallel execution times:

	new-who		classic-who		doctorwho		Email-Enron		Email-EuAll	
K	**serial**	**parallel**	**serial**	**parallel**	**serial**	**parallel**	**serial**	**parallel**	**serial**	**parallel**
2	88	38	97	55	182	98	9436	5295	9742	5798
3	159	82	176	93	339	179	17074	9837	15520	9150
4	244	125	280	140	520	274	25855	15418	22299	13144
5	353	180	391	200	752	384	36589	22059	29769	17182



Comparison of Total execution times of serial/parallel executions for k-values on Email-Enron.

Thus, we conclude that:

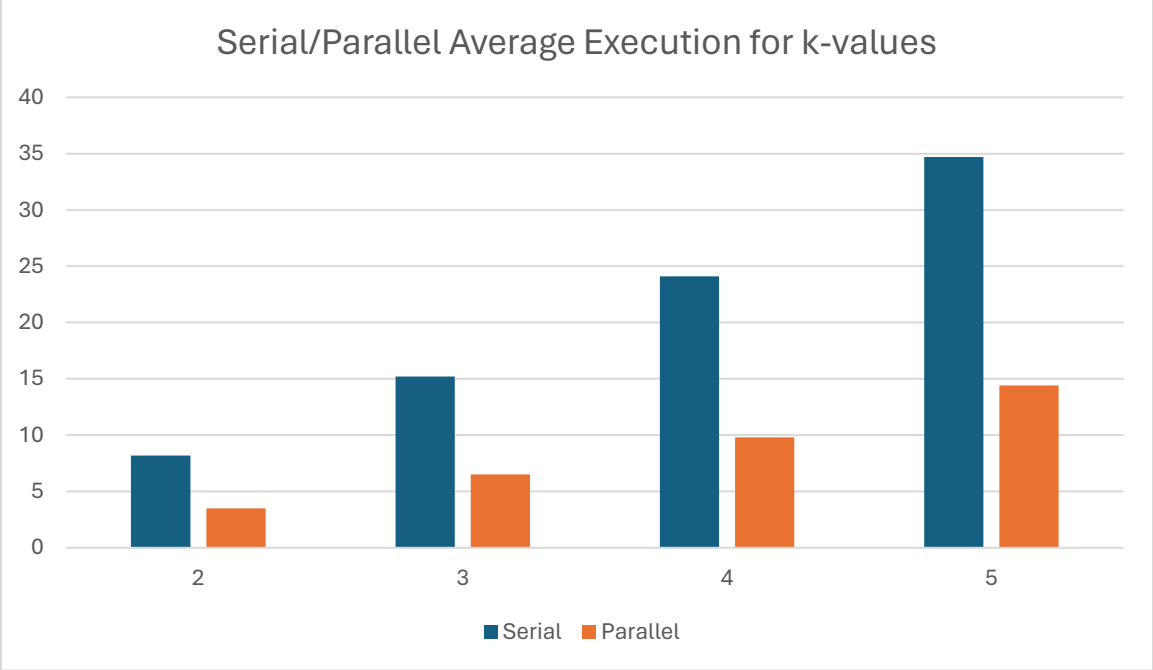
1. The parallel version is more scalable than serial version with respect to k-values.

2. Average Execution Time

Average execution times i.e., execution for a single source-destination pair indicate that the parallel version offers a more significant scalability in terms of single execution as compared to scalability of total execution time over 10 iterations.

Average Serial-Parallel execution times:

	new-who		classic-who		doctorwho		Email-Enron		Email-EuAll	
K	**serial**	**parallel**	**serial**	**parallel**	**serial**	**parallel**	**serial**	**parallel**	**serial**	**parallel**
2	8.2	3.5	9.2	4.1	17.9	7.3	945.9	305.1	973.7	519.6
3	15.2	6.5	17.1	7.4	33.6	13.7	1707.1	571.3	1551.4	861.9
4	24.1	9.8	27.5	10.6	51.6	21.4	2585.1	885.9	2229.3	1247
5	34.7	14.4	38.4	15.6	74.6	29.9	3658.2	1303	2976.5	1656



Comparison of Average execution times of serial/parallel executions for k-values on Email-Enron.

Thus, we conclude that:

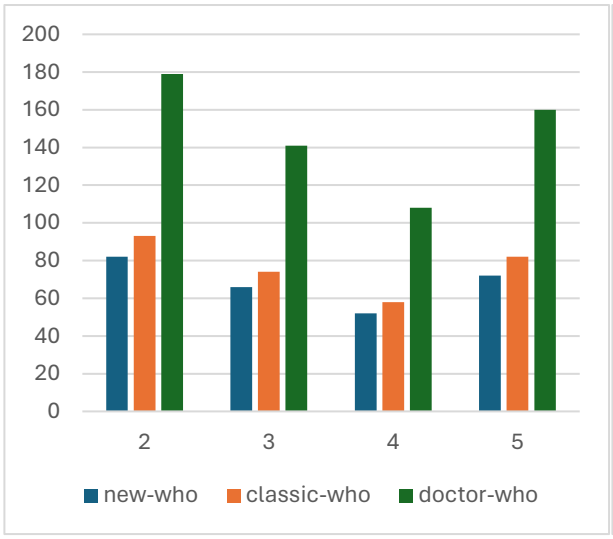
1. Single iteration of parallel version is more scalable than 10 iterations.
   i.e., average execution times are more scalable than total execution times.

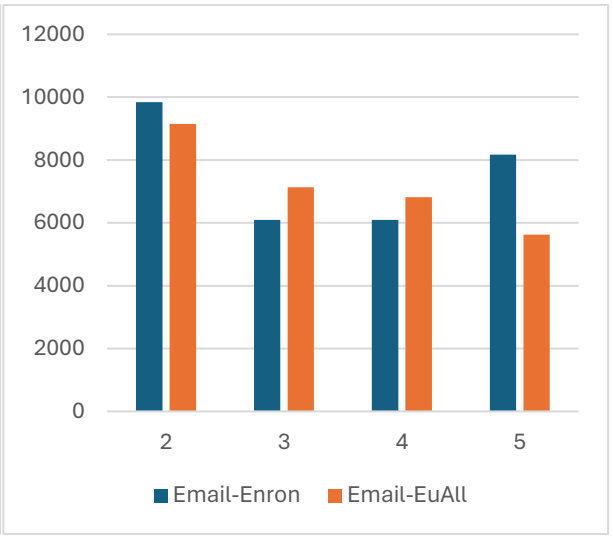
3. Effect of Number of Processes

The number of processes executing the parallel version influences the overall performance of the algorithm. This is because with more processes, communication overheads are increased. Furthermore, the execution of the cluster on a single machine results in increased memory and bus contention, leading to increased execution times. All executions were performed with k = 3.

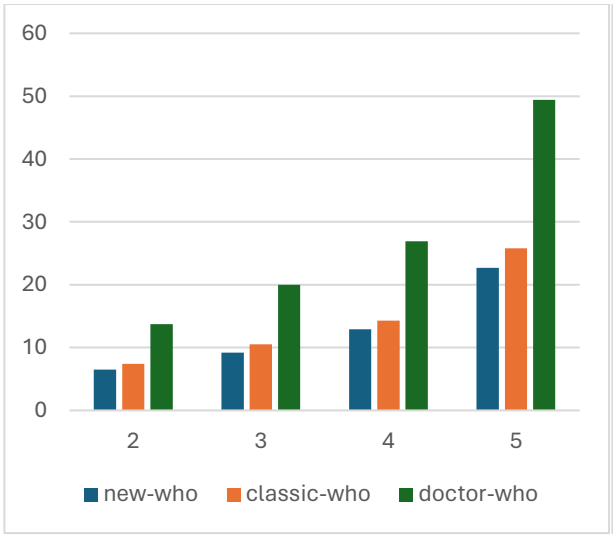
	new-who		classic-who		doctorwho		Email-Enron		Email-EuAll	
np	**total**	**avg**	**total**	**avg**	**total**	**avg**	**total**	**avg**	**total**	**avg**
2	82	6.5	93	7.4	179	13.7	9837	571.3	9150	861.9
3	66	9.2	74	10.5	141	20	6098	949	7131	1080
4	52	12.9	58	14.3	108	26.9	6090	1331	6819	1151
5	72	22.7	82	25.8	160	49.4	8171	2383	5624	2063

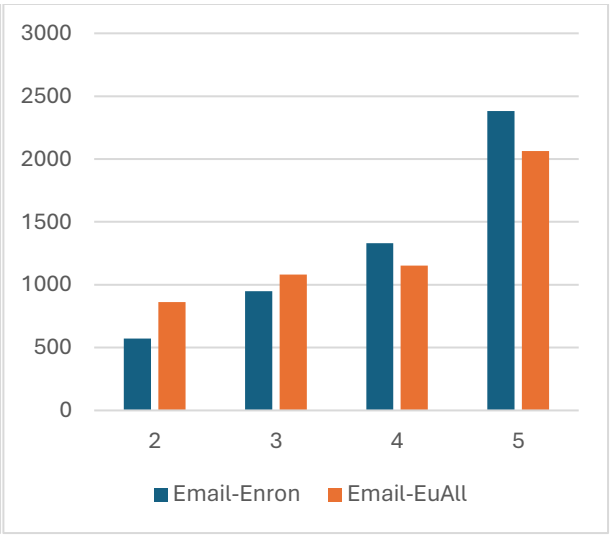
Total Execution Times





Average Execution Times





The total execution time decreases as we increase the number of processes from 2 to 4, and then increases for 5 processes. Under the same conditions we can observe that average execution times continue to increase. This behavior can be justified by considering machine specifications, where the system has 4 processors with 8 simultaneous threads. Thus, we can infer that optimal performance is achieved when each process has a dedicated processor to work on.

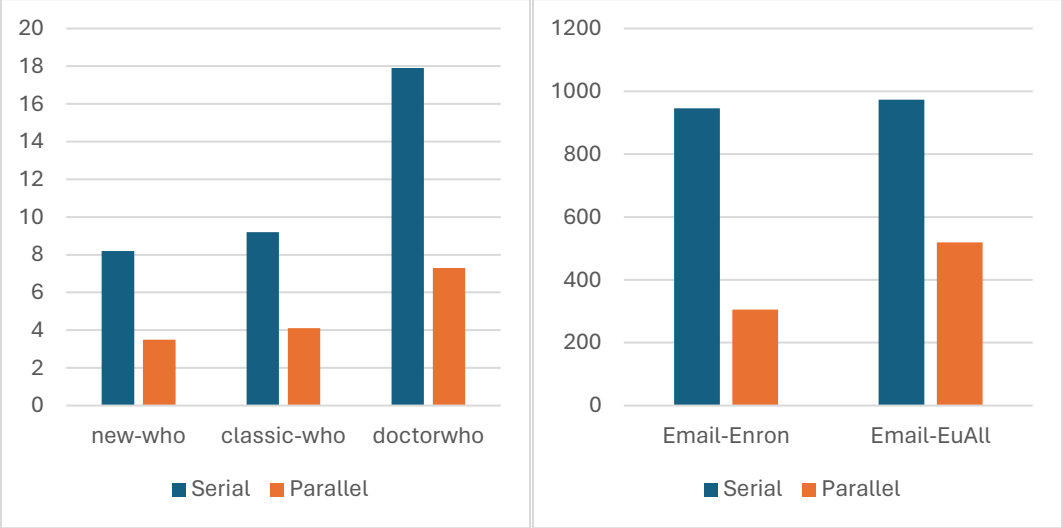
Also note that the execution time of Email-EuAll continues to decrease even with 5 processors, implying that the upper bound of number of processors is higher for sparse graphs.

Thus, we conclude:

1. Total execution time decreases with the increase in number of mpi processes.
2. Average time increases with the increase in number of processes.
3. Machine specific process limit is 4 for total time on dense graph and higher for sparse ones.

4. Graph Density Analysis

Setting k-value to 2, we will now analyze the effect of graph density on the execution times of both serial and parallel versions. For better visualization, the datasets have been grouped into 2 groups based on the number nodes.



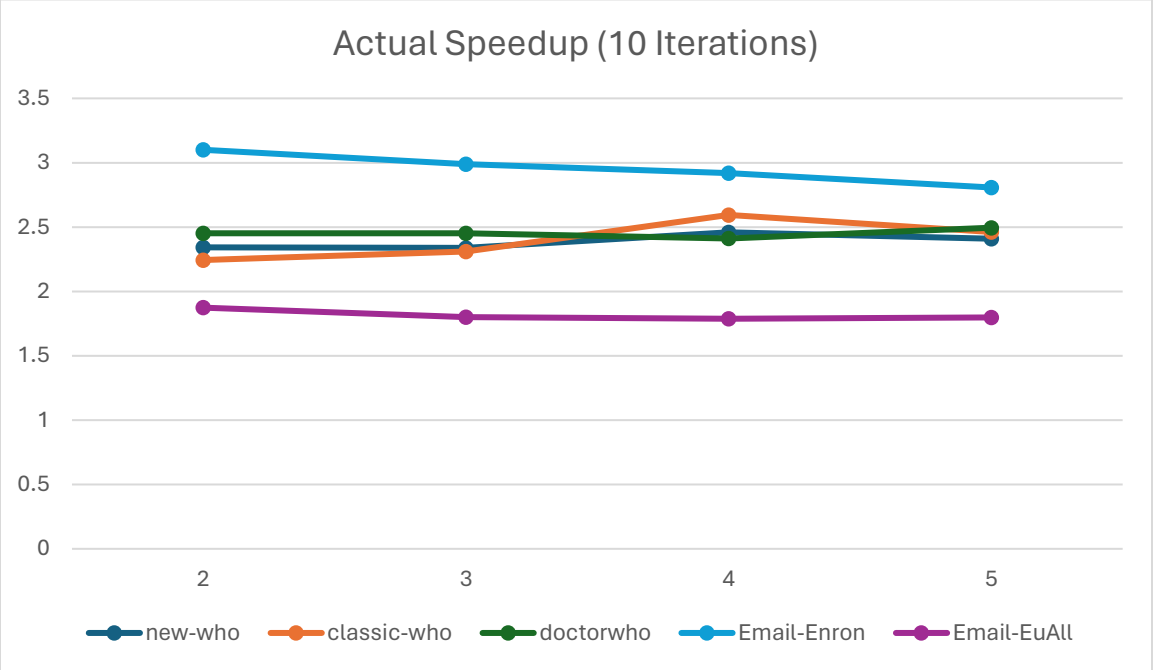
Thus, we conclude that:

1. The performance of parallel versions is better on dense graphs than on sparse ones when the k-values are low.

5. Actual Speedup Metrics:

Actual Speedup on total execution times:

K	new-who	classic-who	doctorwho	Email-Enron	Email-EuAll
2	2.34285	2.24390	2.45205	3.10029	1.87394
3	2.33846	2.31081	2.45255	2.98809	1.79997
4	2.45918	2.59434	2.41121	2.91804	1.78773
5	2.40972	2.46153	2.49498	2.80752	1.79740

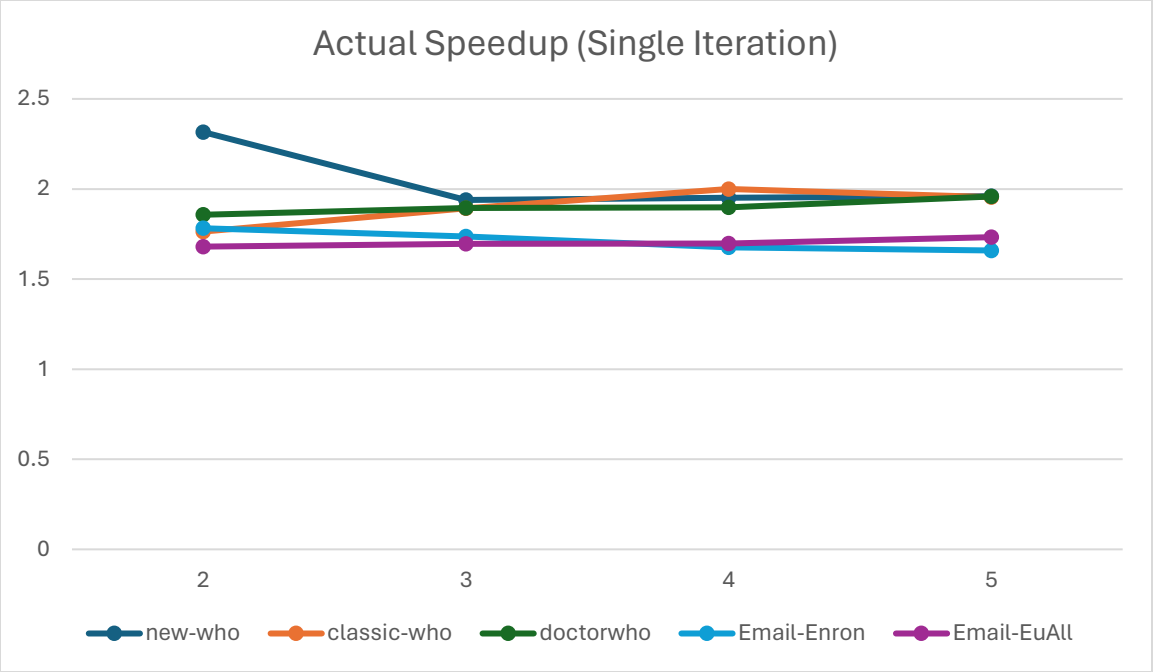


Thus, we conclude that:

1. The parallel version is highly scalable in terms of k-value.
2. Performance improvements are most significant on dense graphs.
3. Improvements are less significant on sparse graphs.
4. Speedup offered by parallelization is not exceptionally fast but still significant.
5. The parallelized version improves execution times by up to 3x.

Actual Speedup on average execution times:

K	new-who	classic-who	doctorwho	Email-Enron	Email-EuAll
2	2.3157	1.7636	1.8571	1.7820	1.6802
3	1.9390	1.8924	1.8938	1.7356	1.6961
4	1.9520	2.0000	1.8978	1.6769	1.6965
5	1.9611	1.9550	1.9583	1.6586	1.7325

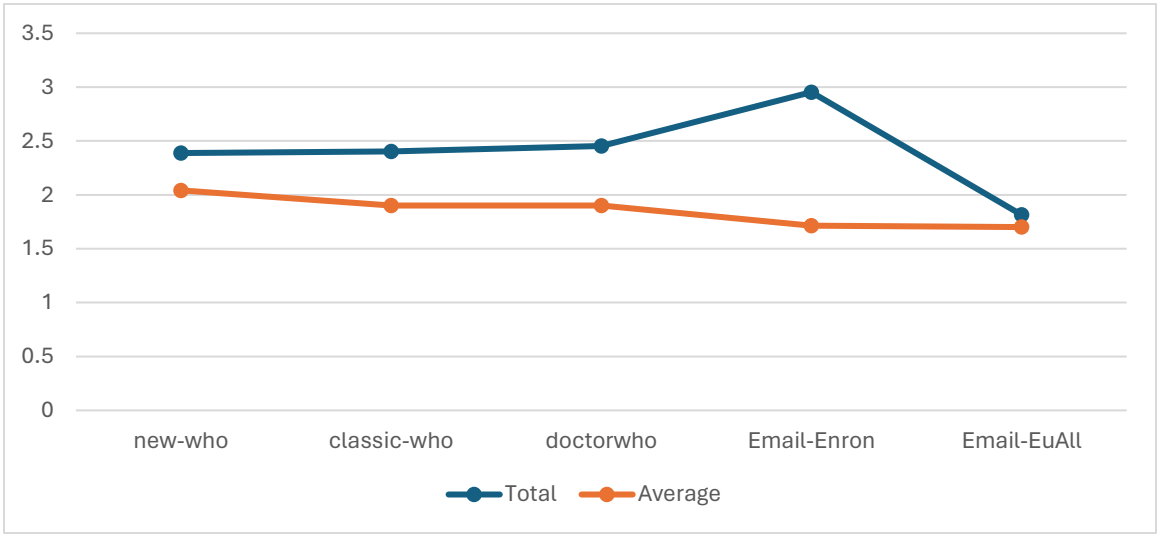


Thus, we conclude:

1. Speedup factor on single iterations is less than that of 10 iterations.
2. Speedup increases with the increase in k-value for specific graphs.
3. The parallel version on single iteration is still fairly scalable.

Average Actual Speedup on Multiple and Single iteration executions

K	new-who	classic-who	doctorwho	Email-Enron	Email-EuAll
Total	2.387556	2.402648	2.452702	2.953491	1.814763
Average	2.041981	1.902777	1.901785	1.713344	1.701373



Conclusion

Summary of Findings

This paper focused on addressing the top Kth shortest paths problem, a vital graph theory problem that has practical applications in various domains such as transportation, logistics, and networking. We implemented and evaluated both serial and parallel versions of the algorithm to explore their efficiency and scalability in real-world scenarios.

1. **Algorithm:** The customized Dijkstra's algorithm efficiently finds and maintains the k shortest paths by exploring nodes in an optimized order. This approach minimizes redundant computations and offers adaptable scalability on dense and sparse graphs alike.
2. **Parallelization:** The parallelized algorithm exploits computational power of multi-core processors and cluster environments to offer significant performance gains:
    a. Up to 3x faster than serial versions
    b. Optimal performance achieved with 4 processors (machine specific).
    c. Better performance improvement on denser graphs compared to sparse ones.
    d. Speedup factor ranges from 1.78x to 3.10x for total execution times.
3. **Scalability:** Parallel version offers better scalability concerning number of processes and the value of k. The scalability was particularly noticeable on denser graphs, making our approach suitable for large-scale graphs.

In conclusion, this paper offers valuable insights into optimizing graph-based computations, demonstrating the potential of parallel computing in accelerating complex graph algorithms.

References

Dijkstra, E. W. (1959, 12 01). A note on two problems in connexion with graphs. *Numerische Mathematik*, 269-271. doi:10.1007/BF01386390

Eppstein, D. (2006, 7 28). Finding the K Shortest Paths. *SIAM Journal on Computing*, 652-673. doi:10.1137/S0097539795290477

Peter Hart, N. N. (1968, 12). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 100-107. doi:10.1109/tssc.1968.300136

Yen, J. (1971, 7). Finding the K shortest loopless paths in a network. *Management Science*, 712-716. doi:10.1090/qam/253822