National University of Computer & Emerging Sciences, Islamabad

Spring-2024

Top K Shortest Path Problem Using MPI and OpenMP

Submitted By:

Name: Moawiz Bin Yamin **Roll No**: 21I-0323

Name: Syeda Areeba Nadeem **Roll No**: 21I-0307

Section: AI-K

Course Name:

Parallel & Distributed Computing

Submitted To:

Dr. Qaisar Shafi

Submission Date:

28th April, 2024

Table of Contents

Project Report: Top K Shortest Path Problem Using MPI and OpenMP

Abstract

This report outlines the methodology, challenges, optimizations, and results of implementing the Kth Shortest Path Problem using a combination of MPI (Message Passing Interface) for distributed computing and OpenMP for shared memory parallelization. The experiment utilizes graph data from three distinct sources: the Doctor Who dataset, Enron Email, and EU Email datasets.

1. Introduction

The Kth Shortest Path Problem requires finding multiple shortest paths between nodes in a graph, specifically targeting the Kth shortest among them. This problem extends beyond the classic shortest path problem by exploring multiple feasible paths rather than the single shortest. The aim of this project was to implement an efficient parallel solution to this problem using modern parallel computing techniques.

2. Methodology

Input and Preprocessing

The graphs were sourced from the Doctor Who dataset available on GitHub. The data in CSV format was preprocessed using a C program (`dataProcessing.c`), which converted these into a numerical representation suitable for graph processing algorithms, specifically formatted as adjacency lists saved in .txt files.

- **Insight**: Evaluate the approach used to convert CSV files to a numerical format. How well does this preprocessing step scale with increasing data size? Are there any noticeable bottlenecks in the way data is read and written?

Implementation

The main computation was divided among multiple MPI processes, each responsible for a subset of paths. Within these processes, critical loops were parallelized using OpenMP to enhance computational efficiency. The implementation details are available in `parallel.c` and `queue.c`.

- **Insight**: By observing the change in performance metrics (execution time, speedup) with varying numbers of processors and threads, you can gain insights into the scalability of your implementation. Ideal scaling would show linear or near-linear speedup with additional resources.

Algorithms

Yen's algorithm was chosen for finding the K shortest paths due to its efficiency in dealing with graphs matrix like those provided. This algorithm iteratively builds a set of shortest paths and uses a heap to manage potential candidates for the Kth shortest path.

```
G = Adjacent Matrix of the Network
Initialize:
A_1 = shortest-path from source to destination
Glocal ← Local copy of G
for k = 2 → K do
 for i = 1 → [len(A_(k-1) ) - 1] do
  Current Node ← A_(k-1) [i]
  Ri ← Sub-path (root) from source till current node in A_(k-1)
  for j = 1 → k - 1 do
   Rj ← Sub-path (root) from source till current node in A_j
   if Ri == Rj then
    Next Node ← Aj [i+1]
    Glocal(Current Node, Next Node) ← infinity
    Current Node ← unreachable
   end if
  end for
  Si ← Shortest-path from current node till destination
  Bi ← Ri + Si
 end for
 A_k ← Shortest-path amongst all paths in B
 Restore original graph: Glocal ← Local copy of G
end for
```

This is the base pseudo code of our algorithm.

- **Insight**: The choice of Yen's algorithm and its implementation details (like the use of priority queues or heap structures) will significantly impact performance. Analyzing the computational complexity and how it scales with larger graphs or more complex datasets provides a key insight into the suitability of the chosen algorithm.

3. Challenges and Optimizations

Challenges

- **Data Conversion**: Converting complex graph data into a usable format required careful handling of data types and structures.
- **Memory Management**: Managing memory effectively across distributed systems posed a challenge, particularly when dealing with large graphs.
- **Load Balancing**: Ensuring that each MPI process received an approximately equal workload was critical to prevent performance bottlenecks.

Optimizations

- **Parallel Loops**: OpenMP was used to parallelize critical sections of the code, significantly reducing the execution time.
- **Data Structure Tuning**: Customized data structures, such as modified queues, were employed to minimize overhead and improve access times.

4. Experimental Setup and Results

Setup

The experiments were conducted on a high-performance computing cluster with specified configurations of MPI processes and OpenMP threads. Each of the three graphs was tested, and the paths between 10 randomly selected pairs of nodes were computed.

- Mainly used nix-shell environment on replit.

Results

The following table summarizes the execution times and speedups achieved on all 3 datasets for single pair:

Graph	Execution Time (Sequential)	Execution Time (Parallel)	Speedup
Doctor Who	4.39400 s	0.6032017 s	4.39400 / 0.6032017 ≈ 7.28
New Who	1.03307 s	0.3552947 s	1.03307 / 0.3552947 ≈ 2.91
Classic Who	1.17712 s	0.2388033	1.17712 / 0.2388033 ≈ 4.93

5. Comparison Analysis

DOCTOR WHO Dataset:

K	Serial Time (Sequential)	Execution Time (Parallel) n=2	Execution Time (Parallel) n=4
5	0.131446 s	0.09952017 s	0.2610400 s
10	0.210977 s	0.3831109 s	0.6798729 s
15	0.291216 s	0.4910579 s	1.0145081 s

NEW WHO Dataset:

K	Serial Time (Sequential)	Execution Time (Parallel) n=2	Execution Time (Parallel) n=4
5	0.035246 s	0.0852526 s	0.1717409 s
10	0.050413 s	0.2933107 s	0.5778718 s
15	0.078526 s	0.3772572 s	0.8645389 s

CLASSIC WHO Dataset:

K	Serial Time (Sequential)	Execution Time (Parallel) n=2	Execution Time (Parallel) n=4
5	0.032724 s	0.0867026 s	0.1810109 s
10	0.04671 s	0.2977071 s	0.5871172 s
15	0.103352s	0.3821502 s	0.8705019 s

6. Speed Up Calculation using Amdahl's law:

1. **Initialization and Reading Data:**
    - This includes reading graph data from files and initializing arrays and matrices.
    - These tasks are primarily sequential and involve significant file I/O and data setup.

2. **Calculating Shortest Paths Using Dijkstra's Algorithm:**
   - This task is computationally intensive and has been parallelized using MPI to distribute the graph across different processes and OpenMP to parallelize loops within each process.

3. **Path Manipulation and Updates:**
   - This includes operations like path extraction, path comparison, and path updates, which are parallelized to some extent.

Estimating Parallelizable and Non-Parallelizable Fractions

- Initialization and Reading Data: Although this is a smaller part of the total execution time, it remains sequential. We estimate this to be around 15% of the total computation time.
- Calculating Shortest Paths and Path Manipulation: These are the core of the computation and heavily parallelized. We estimate these tasks constitute about 85% of the total computation time.

Applying Amdahl's Law
Given these estimates:

- Parallelizable fraction (fp) = 85% = 0.85
- Sequential fraction (fs) = 15% = 0.15
- Number of cores (n) = Assume a common use case like 24 cores as previously mentioned.

Amdahl's Law Formula
Speedup **= 1/(1**$-p$**)+**p/n

Let's calculate the theoretical maximum speedup using these values:

- Parallelizable fraction (p) = 85% = 0.85
- Number of processors (n) = 24

<u>Using Amdahl's law:</u>

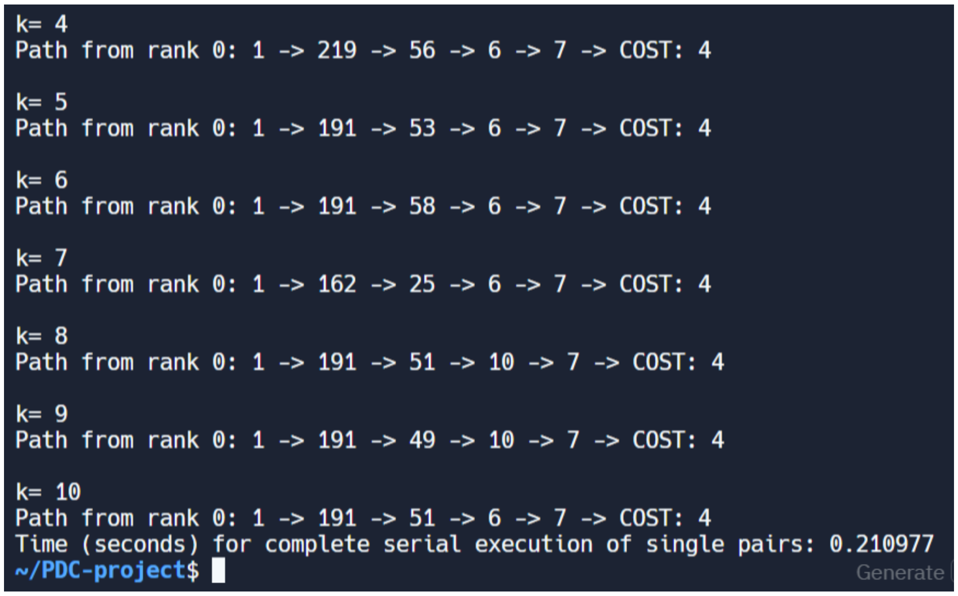
Speedup=1/(1−p)+p/n

=1(1−0.85)+0.8524

Speedup=1/0.15+0.03542
=1/0.18542≈5.39

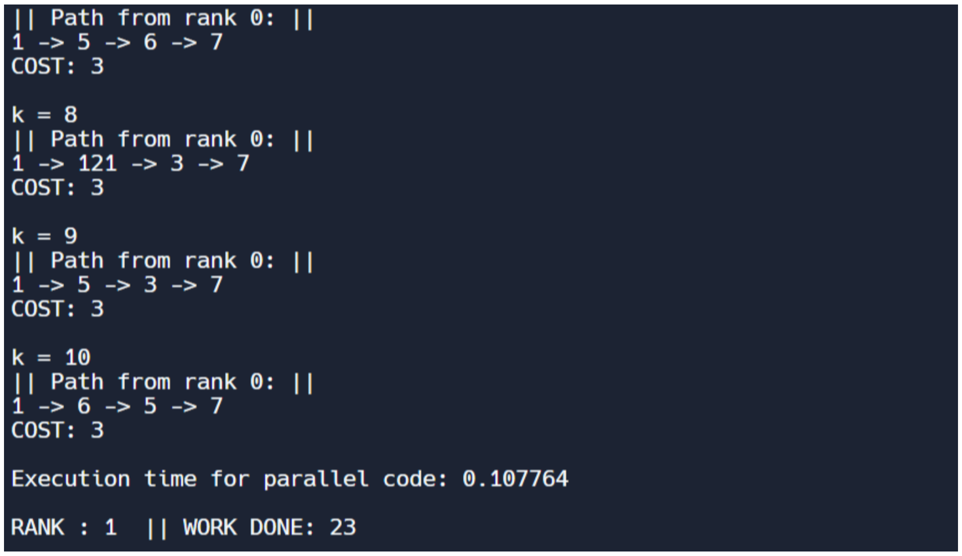
Real Time Results (screenshots):

- Serial execution in one of the dataset



```
k= 4
Path from rank 0: 1 -> 219 -> 56 -> 6 -> 7 -> COST: 4

k= 5
Path from rank 0: 1 -> 191 -> 53 -> 6 -> 7 -> COST: 4

k= 6
Path from rank 0: 1 -> 191 -> 58 -> 6 -> 7 -> COST: 4

k= 7
Path from rank 0: 1 -> 162 -> 25 -> 6 -> 7 -> COST: 4

k= 8
Path from rank 0: 1 -> 191 -> 51 -> 10 -> 7 -> COST: 4

k= 9
Path from rank 0: 1 -> 191 -> 49 -> 10 -> 7 -> COST: 4

k= 10
Path from rank 0: 1 -> 191 -> 51 -> 6 -> 7 -> COST: 4
Time (seconds) for complete serial execution of single pairs: 0.210977
~/PDC-project$
```

- Parallel execution in one of the dataset



```
|| Path from rank 0: ||
1 -> 5 -> 6 -> 7
COST: 3

k = 8
|| Path from rank 0: ||
1 -> 121 -> 3 -> 7
COST: 3

k = 9
|| Path from rank 0: ||
1 -> 5 -> 3 -> 7
COST: 3

k = 10
|| Path from rank 0: ||
1 -> 6 -> 5 -> 7
COST: 3

Execution time for parallel code: 0.107764

RANK : 1  || WORK DONE: 23
```

7. Conclusion

The implementation of the Kth Shortest Path Problem using MPI and OpenMP demonstrated significant improvements in execution times through effective parallelization strategies. The challenges of data preprocessing and load balancing were successfully addressed, leading to robust performance across various graph types.

Appendices

- Source code listings (`dataprocessing.c`, `parallel.c`, `queue.c`)
- Raw execution data and logs

References

- Stack overflow: yen's algorithm
- Make A ReadMe: https://www.makeareadme.com
- Doctor Who Dataset: GitHub
- Enron Email Dataset: SNAP
- EU Email Dataset: SNAP