Parallel Distributed Computing



Group Members:

Moeed Asif (i21-0483)

Muhammad Ahmad (i21-0790)

Section: Y Date: 27/04/2024

# Introduction:

**K-Shortest Path Search**

The project aims to implement an efficient algorithm to find the K-shortest paths in a given graph. The project is implemented in C++ and utilizes concepts of parallel computing and graph theory.

# Code Structure:

textfilegraphs.cpp:

* This file contains the implementation of the functions declared in the header file.
* Functions for reading graph data from files, initializing adjacency matrices, finding the number of neighbors for a given node, and finding K-shortest paths are defined here.
* The main function in this file demonstrates the usage of the implemented functionalities by reading graph data from files, computing K-shortest paths, and measuring execution time.

textfilegraphs\_parallel.cpp:

* This file extends the functionalities of the original implementation to support parallel computing using MPI (Message Passing Interface) and OpenMP.
* It includes similar functionalities as the sequential implementation but parallelizes certain operations to improve performance.
* MPI functions are used for inter-process communication, allowing for parallel execution across multiple processes.
* OpenMP directives are used for parallelizing loops to leverage multi- core processors effectively.

.

csvfilegraphs.cpp:

* This source file contains the implementation of the classes declared in graphs2.h. It defines methods for adding edges between nodes, reading graph data from a file, initializing the adjacency matrix, printing edges, and finding the k shortest paths in a serial manner.
* Reading from File: The readFromFile() function reads graph data from a CSV file, where each line represents an edge between two nodes along with its weight.
* Adjacency Matrix: The initialize\_AdjacencyMatrix() function initializes the adjacency matrix based on the graph data. It iterates over the nodes and their edges to populate the matrix with edge weights.
* K Shortest Paths: The findKShortestSerial() function computes the k shortest paths using a modified Dijkstra's algorithm. It maintains a priority queue of nodes based on their distances from the source node.

csvfilegraphs\_parallel.cpp:

* This source file extends the functionality of graphs2.cpp by introducing parallelization using MPI (Message Passing Interface) and OpenMP (Open Multi-Processing).
* MPI and OpenMP: MPI is used for distributed memory parallelism, allowing multiple processes to communicate and collaborate in finding the k shortest paths. OpenMP is used for shared memory parallelism within each process, leveraging multiple threads for parallel computation.
* Parallel K Shortest Paths: The findKShortestParallel() function parallelizes the computation of k shortest paths using MPI and OpenMP. It distributes the workload among multiple processes and utilizes parallel loops to explore different paths concurrently.

# Mapping Strategy:

Our mapping strategy involved determining the number of neighbors for each thread by initially mapping each neighbor of the start node. By assessing the number of neighbors associated with each, we allocated a corresponding num- ber of processes to efficiently explore these paths. However, given the require- ment for random start and end nodes, we erred on the side of caution by allo- cating 1-3 processes, as some start nodes might have limited neighboring nodes.

To prevent potential issues stemming from an excessive number of processes compared to available neighbors, we implemented a check to terminate the program early and display an error message if the processor count exceeded the number of neighbors.

Subsequently, after the threads had explored their respective paths, we ag- gregated their results and applied the Quick Sort algorithm, renowned for its efficiency, to sort the paths. This approach was chosen to mitigate com- munication and synchronization challenges inherent in inter-thread com- munication, thus streamlining the process.

Ultimately, our methodology aimed to efficiently derive the k shortest paths while also incorporating the distance from the start vertex to the neighboring vertex, optimizing the computational process.

Below are the running outputs of all the files read.

Each file is run 10 times and finally the Average is calculated. Visual graphs are also illustrated for better view.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size of data | Serial Time (s) | Parallel Time (s) | Speedup | Speedup (Amdahl's Law) | Efficiency | Processors in Parallel |
| 265214 | 5.33253 | 0.79543 | 6.703958865 | 0.257178274 | 0.085726091 | 3 |
| 265215 | 5.32642 | 0.73892 | 7.208385211 | 0.17944537 | 0.059815123 | 3 |
| 265216 | 5.36967 | 0.719199 | 7.466181127 | 0.178272091 | 0.05942403 | 3 |
| 265217 | 5.37443 | 0.701341 | 7.663076877 | 0.178310003 | 0.059436668 | 3 |
| 265218 | 5.28681 | 0.781309 | 6.766605786 | 0.180269622 | 0.060089874 | 3 |
| 265219 | 5.61038 | 0.733508 | 7.648696401 | 0.170797616 | 0.056932539 | 3 |
| 265220 | 5.37791 | 0.737562 | 7.291468378 | 0.17781685 | 0.059272283 | 3 |
| 265221 | 5.3861 | 0.816711 | 6.594866483 | 0.176730371 | 0.058910124 | 3 |
| 265222 | 5.41609 | 0.815701 | 6.639798161 | 0.175809019 | 0.058603006 | 3 |
| AVERAGE | 5.386704444 | 0.759964556 | 7.109226365 | 0.186069913 | 0.062023304 | 3 |
|  |  |  |  |  |  |  |
| 36692 | 0.380267 | 0.341717 | 1.112812649 | 1 | 1 | 1 |
| 36692 | 0.374805 | 0.386713 | 0.969207138 | 1 | 1 | 1 |
| 36692 | 0.387539 | 0.354243 | 1.093991977 | 1 | 1 | 1 |
| 36692 | 0.351068 | 0.377773 | 0.9293094 | 1 | 1 | 1 |
| 36692 | 0.373273 | 0.412067 | 0.905855116 | 1 | 1 | 1 |
| 36692 | 0.420931 | 0.378135 | 1.113176511 | 1 | 1 | 1 |
| 36692 | 0.374986 | 0.347174 | 1.080109686 | 1 | 1 | 1 |
| 36692 | 0.34675 | 0.370234 | 0.936569845 | 1 | 1 | 1 |
| 36692 | 0.362716 | 0.379055 | 0.956895437 | 1 | 1 | 1 |
| 36692 | 0.354113 | 0.33563 | 1.055069571 | 1 | 1 | 1 |
| AVERAGE | 0.3726448 | 0.3682741 | 1.015299733 | 1 | 1 | 1 |
|  |  |  |  |  |  |  |
| 377 | 0.177231 | 0.22123 | 0.801116485 | 2.214901562 | 0.738300521 | 3 |
| 377 | 0.162653 | 0.214444 | 0.758487064 | 2.263628173 | 0.754542724 | 3 |
| 377 | 0.171677 | 0.262231 | 0.654678509 | 2.233216263 | 0.744405421 | 3 |
| 377 | 0.190039 | 0.22311 | 0.851772668 | 2.173790177 | 0.724596726 | 3 |
| 377 | 0.223127 | 0.224239 | 0.995041005 | 2.074324427 | 0.691441476 | 3 |
| 377 | 0.198425 | 0.195195 | 1.016547555 | 2.147689444 | 0.715896481 | 3 |
| 377 | 0.183105 | 0.214926 | 0.85194439 | 2.195855688 | 0.731951896 | 3 |
| 377 | 0.189167 | 0.20667 | 0.91530943 | 2.176540664 | 0.725513555 | 3 |
| 377 | 0.189176 | 0.191282 | 0.988990077 | 2.176512241 | 0.72550408 | 3 |
| 377 | 0.174012 | 0.214408 | 0.811592851 | 2.225479665 | 0.741826555 | 3 |
| AVERAGE | 0.1858612 | 0.2167735 | 0.864548004 | 2.188193831 | 0.729397944 | 3 |

Data size = 265214

8

7

6

5

4

3

2

1

0

Serial Time (s)

Parallel Time (s)

Speedup

Speedup (Amdahl's

Law)

Efficiency

Series 1

Data size = 36692

1.2

1

0.8

0.6

0.4

0.2

0

Serial Time (s)

Parallel Time (s)

Speedup

Speedup (Amdahl's

Law)

Efficiency

Series 1 Series 2 Series 3 Series 4 Series 5

Data size = 377

2.5

2

1.5

1

0.5

0

Serial Time (s)

Parallel Time (s)

Speedup

Speedup (Amdahl's

Law)

Efficiency

Series 1 Series 2 Series 3 Series 4 Series 5

# Conclusion:

Based on the results provided, it is evident that parallelization significantly improves performance for larger file sizes. However, the impact of parallelization diminishes for smaller files, as indicated by lower speedup values. Amdahl's Law highlights the importance of optimizing parallelizable sections of the program to achieve higher speedup. Additionally, the efficiency of parallelization depends on factors such as file size and the number of processors utilized. Overall, parallelization demonstrates its effectiveness in accelerating computation, particularly for large-scale data processing tasks.