All Pairs Shortest Path

Group: 31
Members: Ben Smus, Terence Cheung, Lily Yang
Course: CMPT 431

Introduction

In this project, we will explore the serial, parallel, and distributed implementations of the Floyd-Warshall algorithm. It's a popular and efficient algorithm used to find the shortest distance between any two vertices in a directed weighted graph. We will not be accounting for negative weights in our exploration for simplicity. We expect the serial version to be faster than parallel or distributed implementations when our input graph size is small, but as the input size increases, we should expect the shared memory and distributed memory versions to run faster than the serial version. As for the comparison between runtime for shared memory versus distributed memory, we expect shared memory to be slightly faster.

Background

Finding the shortest path from source to destination is a common problem in many fields such as networking, transportation, flight planning, geographical information systems, and etc. We usually represent such systems using weighted directed graphs. Edge weights can represent distances between cities, costs of financial transactions, available bandwidth, fuel costs between airports, and many other things.

We could have used Dijkstra's algorithm for solving this problem. Since Dijkstra's SSSP algorithm is O(V^2), if we run it from every vertex to get All Pairs Shortest Paths ("APSP"), it will be O(V^3). This is the same runtime as the Floyd-Warshall algorithm.

The Floyd-Warshall algorithm has an advantage over Dijkstra's algorithm when solving the APSP problem in a distributed memory environment. Floyd-Warshall does not require the graph adjacency matrix to be copied in the memory of each process. Dijkstra's algorithm requires that every processor has enough local memory to store the entire adjacency matrix of the graph.

Implementation Details

Our project has three executables:

1. `floyd_serial_threaded`
2. `floyd_distrib`
3. `print_shortest_path`

Both the first and second executables are implementations of the Floyd-Warshall algorithm which means that they take a directed graph in the form of an adjacency matrix as input and produce a distance matrix as output. The adjacency matrix file path is passed with `--inputFile` and the distance matrix file path (to be produced) is passed with `--outputFile`. We chose to store the adjacency matrix in the form of a plain text file for simplicity, this could be improved by using a binary encoding which would reduce file size.

The first executable contains the serial and shared memory implementations of the Floyd-Warshall algorithm. `floyd_serial_threaded --mode=0` is serial while `floyd_serial_threaded --mode=1 --np=<number of threads>` is multithreaded. The second executable is run by `project_slurm.script` or by calling `mpirun` directly from the command line.

The third executable takes an adjacency matrix, its matching distance matrix, start node, and end node. Then it prints the shortest path. E.g.

```
Shortest path (distance 5) from 3 to 1:
3 -> 0 -> 1
```

Using files for the distance matrix allows the user to generate the distance matrix once, and then run shortest path queries on that "graph/distance matrix pair" for any two nodes with very low cost.

Serial

The serial implementation is straightforward. We go through 0 → k -1 nodes, and using this as the intermediary node from row i to k, and from k to j, if the total distance is less than the distance going from i to j, we will update Graph[i][j] with the smaller distance.

Pseudocode:

```
For k in number of nodes {
        For i in number of rows {
                For j in number of columns {
                        Graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j]);
                }
        }
}
```

Notes on parallelizing the the Floyd-Warshall algorithm

In a parallel implementation of the all pairs shortest path algorithm, the program must split the computation evenly among all processes. Our group opted to assign each task with an equal number of columns from the graph, so each task will have n/p columns. A solution that would scale to a larger number of tasks would use 2-D block mapping to create squares the size of n/√p x n/√p where n is the number of nodes and p is the number of tasks.

Multithreaded (Shared Memory)

In optimal conditions, we can expect a speedup of exactly proportional to the number of threads i.e. if there are 4 threads, it will take ¼ the amount of time.

Pseudocode:

```
For each process P in parallel {
        For k in number of nodes {
                For i in number of rows {
                        For j in assigned columns {
                                Graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j]);
                        }
                }
                barrier();
        }
}
```

Distributed Memory

Since we divided up the work for each process into a whole column, it is relatively simple to determine which process requires what pieces of data. Each process will require the whole kth column from the process that is working on that part of the

graph and will be able to use the kth row of its own assigned columns. This simplifies the algorithm when compared to using 2-D block mapping, with 2-D block mapping, each process will have to figure out where to send their data, but by splitting by columns, only one process will have to broadcast 1 column for each iteration. The expected speedup for the distributed algorithm is around the same as the shared memory implementation on a system with low communication overhead.

In the distributed memory implementation, process 0 reads the graph file into a buffer and sends chunks of that buffer to all other processes. Then all the processes work together to iterate over their assigned columns. Then they send data back to process 0 which writes to the distance matrix file.

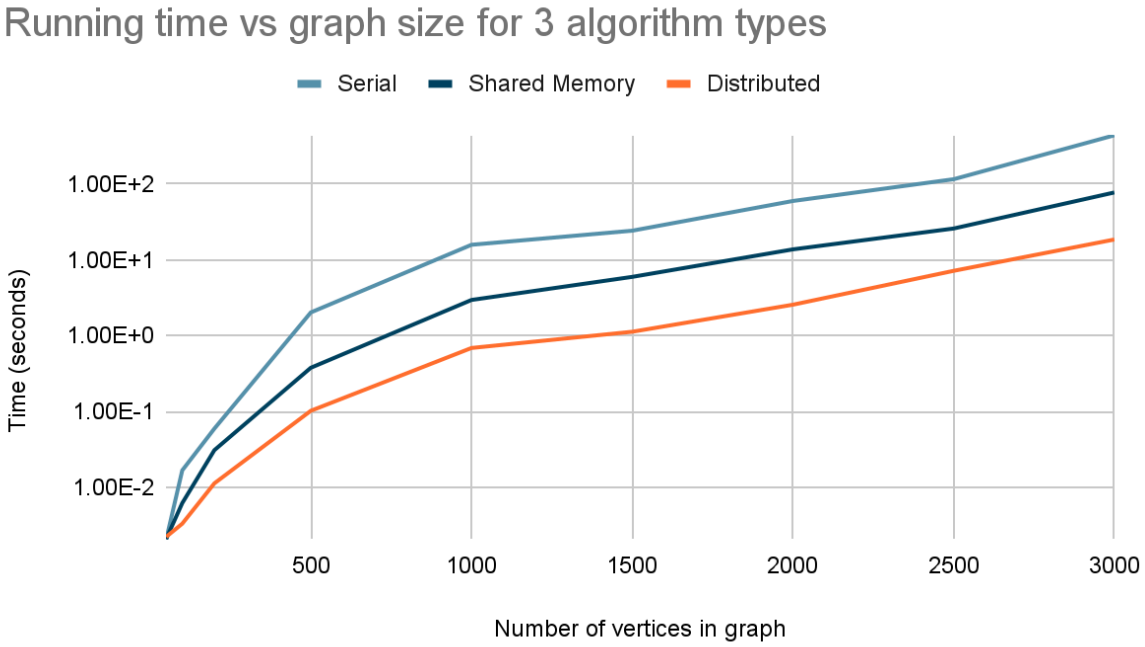
Evaluations

The following performance data was produced by running our program on our local machines (Ubuntu 22.04 x86).

Performance of varying input sizes

Data:

Num verts	Serial	Multithreaded	Distributed
50	0.002073	0.0021169	0.0022517
100	0.016884	0.0062561	0.0033522
200	0.059876	0.031167	0.011396
500	2.0286	0.37894	0.10279
1000	15.776	2.9619	0.69037
1500	24.18	5.9624	1.1291
2000	59.526	13.727	2.5647
2500	115.11	25.836	7.1591
3000	436.14	77.038	18.505

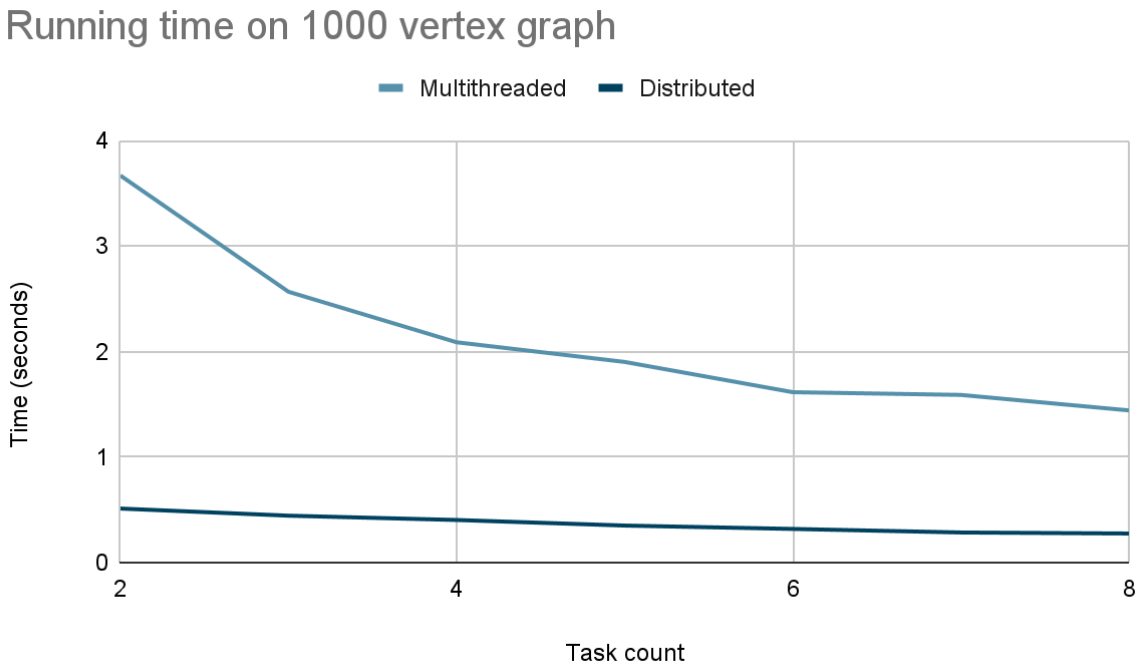


In the graph above: Shared memory used 6 threads, distributed memory used 6 processes.

Performance between multithreaded and distributed

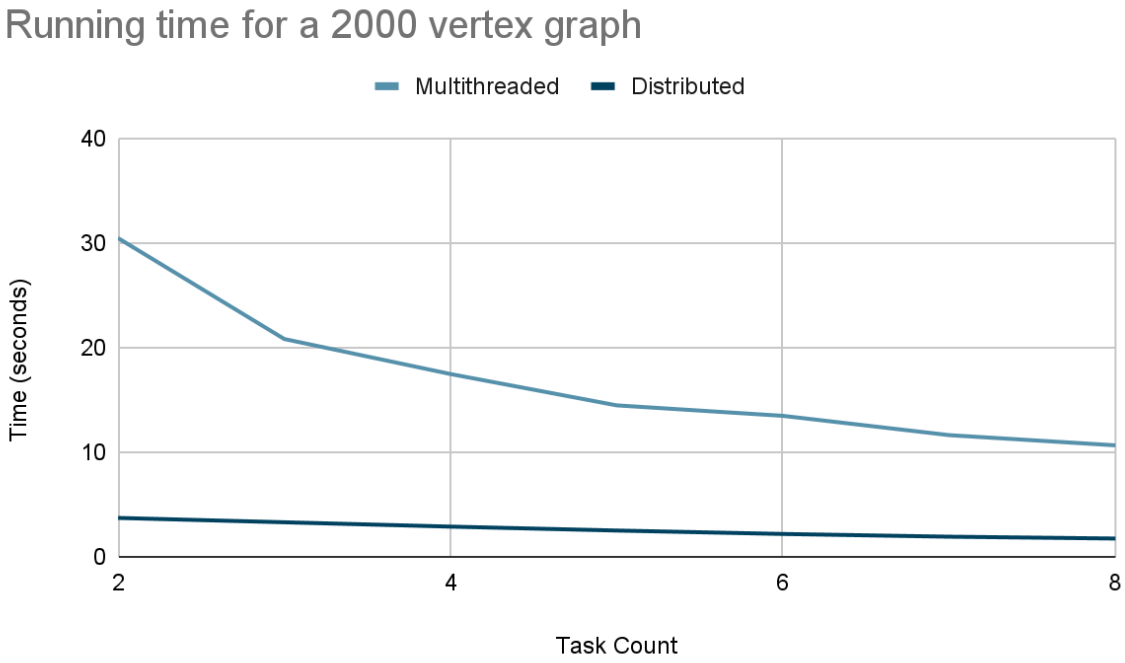
Data for multi-processor runtime on 1000 vertex graph:

Num tasks	Multithreaded	Distributed
2	3.6685	0.51311
3	2.5674	0.44567
4	2.0888	0.40485
5	1.9025	0.35202
6	1.6158	0.32015
7	1.59	0.28647
8	1.4432	0.2767



Data for multi-processor runtime on 2000 vertex graph:

Num tasks	Multithreaded	Distributed
2	30.451	3.7146
3	20.81	3.2989
4	17.464	2.8841
5	14.47	2.5096
6	13.469	2.1819
7	11.625	1.9208
8	10.651	1.7457



From these graphs, we can see large amounts of speedup when going from 2 threads to 3 and 3 threads to 4. When going from 3 threads to 4, there is still speedup, but it is not as large as from 2 to 3.

Conclusions

The problem addressed in this project was all pairs shortest path which finds the shortest distance between any two vertices in a weighted graph. When exploring Dijkstra's algorithm, we found that it is would not work well in a distributed memory environment. From this research we learned that not all algorithms that parallelize well to multiple threads parallelize well to multiple processes/CPUs.

From this project we learned about the ease of a serial implementation and its effectiveness on small input sizes, but as the input size and work increases, there is a need for faster solutions since a serial implementation is no longer able to keep up. From this specific task, we observed large speedups by using both multiple threads and multiple processes. MPI was more performant than C++ threads, which was unexpected. We observed that for small graphs, there is no improvement in performance by using multiple tasks, in fact, the performance was worse. As the graphs increase in size, the relative speedup also increases. In the largest experiment with 3000 vertices, saw a speedup of 24000% by the

distributed implementation compared to the serial implementation. As expected, when doubling the number of processors, the time it takes to process a workload is approximately halved.