CMPT431 Report

All-Pairs Shortest Path

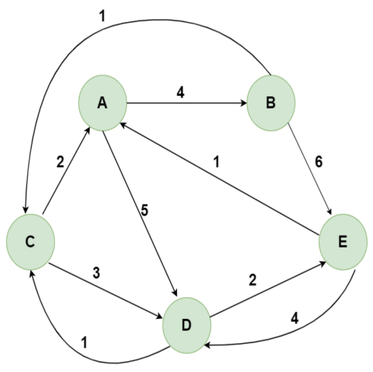
Jan Mazurek
Eric Kempton
Caleb Bradley

1 Introduction

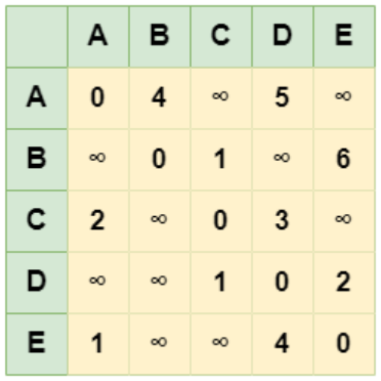
A fundamental problem in graph theory is finding the shortest paths between all pairs of nodes in a graph. This problem, known as the All-Pairs Shortest Paths (APSP) problem, has numerous applications in areas such as transportation, communication networks, and social network analysis. The objective is to determine the shortest path between any two nodes in a graph, taking into account the weights associated with the edges connecting them. To solve this problem, we can deploy the Floyd-Warshall algorithm. Named after its creators, it's used to efficiently find the shortest paths between all pairs of nodes in a weighted graph. It differs from other shortest path algorithms such as Dijkstra and Bellman Ford which are single source shortest path algorithms; the Floyd-Warshall algorithm works on both directed and undirected weighted graphs, so long as there are no negative cycles. In this project, we differentiate three unique deployments of the algorithm as an effort to find performance improvements where computing resources are available. By using parallel and distributed techniques, not only can we find the shortest path for all pairs of a graph, but do so efficiently.

2 Background

Traditionally, the Floyd-Warshall algorithm has been the go-to solution for solving the APSP problem in a serial computing environment. This algorithm, based on dynamic programming principles, computes the shortest paths between all pairs of nodes in a graph. In its serial implementation, the algorithm exhibits a time complexity of $O(n^3)$, where n is the number of nodes in the graph. Therefore it is an excellent choice for optimization using parallel and distributed methods. Parallel and distributed computing architectures offer the potential to accelerate the computation of shortest paths by leveraging multiple processing units concurrently.

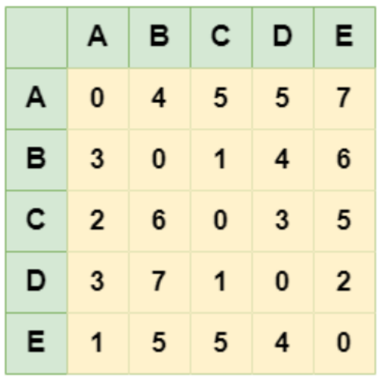


Example Graph



	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

Initial Distance Matrix



	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

After Floyd Warshall

One state of the art-implementation, is the asynchronous distributed version of the algorithm proposed by Toueg. This approach distributes the computational load across multiple nodes in a network, thereby potentially reducing the overall computation time. Each node is responsible for computing a portion of the shortest paths and exchanging information with its neighbors to synchronize the computation.

3 Implementation Overview

In each implementation, we feed in a directed graph and randomly assign edges weights based on seed at runtime. These edges are displayed as a two dimensional array before and after execution to show the execution of the algorithm. The program also can handle weighted and directed graphs, but for the test cases handled in the evaluation portion of this paper, random weights are used.

Our output displays the input graph as it is initialized, performs the algorithm, then outputs both the final distance matrix and the final 'via' matrix. To represent infinity, or unknown (non-neighbor) nodes or unreachable nodes, the value '999' is used.

3.1 Serial Implementation

The serial implementation of the Floyd-Warshall algorithm operates by iteratively updating a matrix to represent the lengths of the shortest paths between all pairs of nodes in the graph. Another matrix is maintained to store the intermediate nodes through which the shortest paths pass, referred to as the "via" matrix.

The algorithm begins by initializing the length matrix with the weights of the edges between adjacent nodes in the graph. It then iterates over all possible pairs of nodes and considers each intermediate node as a potential candidate for improving the shortest path. For each pair of nodes (i, j), the algorithm checks if going through an intermediate node k results in a shorter path than the current shortest path between i and j. If such a shorter path exists, the length matrix is updated accordingly, and the "via" matrix is updated to reflect the new intermediate node.

This process continues for each pair of nodes and intermediate nodes until no further improvements can be made. At the end of the algorithm, the length matrix contains the lengths of the shortest paths between all pairs of nodes, and the "via" matrix provides information about the intermediate nodes along these paths. In terms of time complexity, the serial implementation of the Floyd-Warshall algorithm has a complexity of O(n^3), where n is the number of nodes in the graph. This is because the algorithm iterates over all pairs of nodes and considers each intermediate node. We attempt to use computing resources to parallelize and significantly speed up this implementation.

3.2 Parallel Implementation

In the parallel implementation, the work is divided into independent tasks, each handled by a separate thread. Threads operate on subsets of evenly distributed vertices (total vertices / number of threads). Synchronization is achieved using barriers, ensuring that in each iteration of the algorithm, all threads complete their computation phase before moving on to the communication phase, and all threads complete their communication phase before moving on to the next iteration. By partitioning the vertices rather than the edges, the parallel implementation avoids potential bottlenecks that may arise from uneven edge distributions.

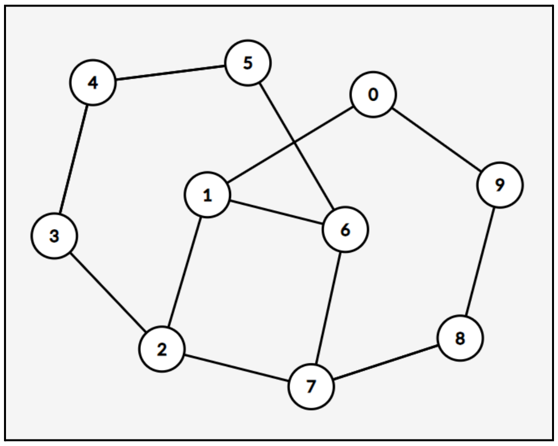
3.3 Distributed Implementation

The distributed implementation leverages MPI to improve the runtime of the All-Pairs Shortest Path (APSP) algorithm, enabling efficient computation of shortest paths in large graphs across multiple computing nodes. The graph's vertices are partitioned among a specified number of MPI processes, with each process responsible for computing shortest paths for a portion of vertices. This partitioning strategy also ensures balanced workload distribution and minimizes data movement across processes. As in assignment 6, communication between processes is facilitated through MPI message passing, with messages exchanged to convey information about the shortest path tree and relevant portions of the length matrix. Synchronization points are implicit through the use of MPI_Bcast, which one process per iteration uses to broadcast its iteration-specific row of the length matrix to every other process. The blocking nature of MPI_Bcast and MPI_Recv ensures that each iteration of the Floyd-Warshall algorithm proceeds only once all processes have completed their communication phase, ensuring coordinated progress across processes at key stages of the algorithm.

During each iteration of the distributed Floyd-Warshall algorithm, processes exchange relevant information as mentioned above and perform local computations to update their portion of the distance matrix. Once the necessary information exchange is completed, processes independently update their local portion of the distance matrix based on the received data.

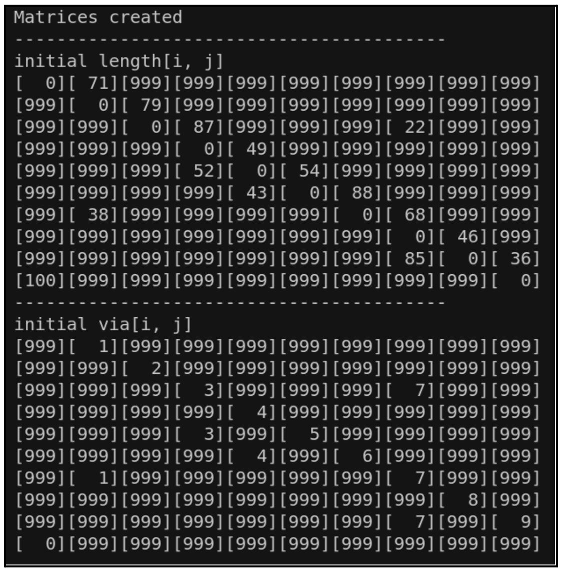
4 Evaluation

4.1 Correctness

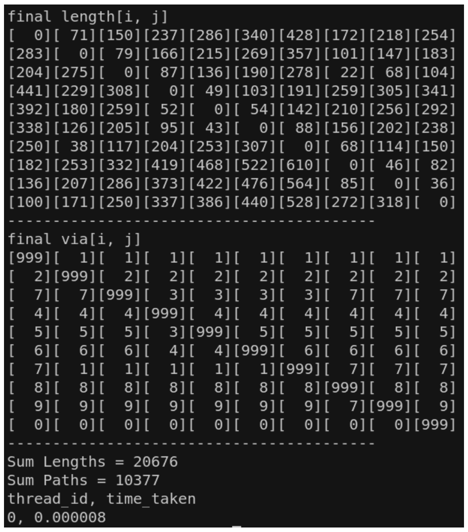
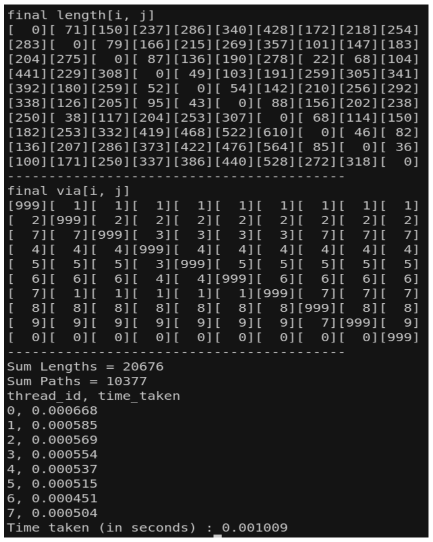
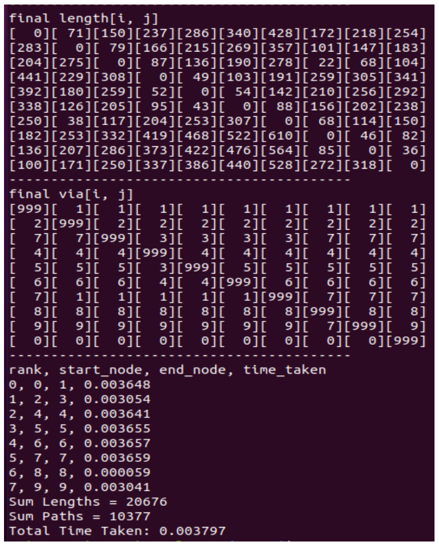


We used a self-created python script to generate graphs that could be used to feed into our implementation. While it is difficult to check over the results of larger graphs, verifying the similarities between the smaller inputs, such as the 10 vertex graph prove helpful. The test graph, as well as the randomly assigned edge weights that are used in all three versions of the code are seen above and to the right. Correctness is verified by confirming that each of the graphs produce the same output arrays and the same sum of lengths and paths.

The serial, parallel and distributed versions are compared below on the same seed (=22), thus calling for the same random edge weights (seen on the right) to be generated in each case. Comparing the results of the arrays for the higher node inputs on 100 and 1000 vertices also reveal a match, namely the Sum of Lengths and Sum of Paths all being equivalent. The final matrixes are commented out in the final submission as the printing matrix to screen did not scale well with large numbers of nodes. Instead, we chose to keep summation as a means of tracking correctness in general use.



```
Matrices created
-----------------------------------------
initial length[i, j]
[  0][ 71][999][999][999][999][999][999][999][999]
[999][  0][ 79][999][999][999][999][999][999][999]
[999][999][  0][ 87][999][999][999][ 22][999][999]
[999][999][999][  0][ 49][999][999][999][999][999]
[999][999][999][ 52][  0][ 54][999][999][999][999]
[999][999][999][999][ 43][  0][ 88][999][999][999]
[999][ 38][999][999][999][999][  0][ 68][999][999]
[999][999][999][999][999][999][999][  0][ 46][999]
[999][999][999][999][999][999][999][ 85][  0][ 36]
[100][999][999][999][999][999][999][999][999][  0]
-----------------------------------------
initial via[i, j]
[999][  1][999][999][999][999][999][999][999][999]
[999][999][  2][999][999][999][999][999][999][999]
[999][999][999][  3][999][999][999][  7][999][999]
[999][999][999][999][  4][999][999][999][999][999]
[999][999][999][  3][999][  5][999][999][999][999]
[999][999][999][999][  4][999][  6][999][999][999]
[999][  1][999][999][999][999][999][  7][999][999]
[999][999][999][999][999][999][999][999][  8][999]
[999][999][999][999][999][999][999][  7][999][  9]
[  0][999][999][999][999][999][999][999][999][999]
```

Results of Serial Execution	Results of Parallel Execution	Results of Distributed Execution
		

```
final length[i, j]
[  0][ 71][150][237][286][340][428][172][218][254]
[283][  0][ 79][166][215][269][357][101][147][183]
[204][275][  0][ 87][136][190][278][ 22][ 68][104]
[441][229][308][  0][ 49][103][191][259][305][341]
[392][180][259][ 52][  0][ 54][142][210][256][292]
[338][126][205][ 95][ 43][  0][ 88][156][202][238]
[250][ 38][117][204][253][307][  0][ 68][114][150]
[182][253][332][419][468][522][610][  0][ 46][ 82]
[136][207][286][373][422][476][564][ 85][  0][ 36]
[100][171][250][337][386][440][528][272][318][  0]
----------------------------------------
final via[i, j]
[999][  1][  1][  1][  1][  1][  1][  1][  1][  1]
[  2][999][  2][  2][  2][  2][  2][  2][  2][  2]
[  7][  7][999][  3][  3][  3][  3][  7][  7][  7]
[  4][  4][  4][999][  4][  4][  4][  4][  4][  4]
[  5][  5][  5][  3][999][  5][  5][  5][  5][  5]
[  6][  6][  6][  4][  4][999][  6][  6][  6][  6]
[  7][  1][  1][  1][  1][  1][999][  7][  7][  7]
[  8][  8][  8][  8][  8][  8][  8][999][  8][  8]
[  9][  9][  9][  9][  9][  9][  9][  7][999][  9]
[  0][  0][  0][  0][  0][  0][  0][  0][  0][999]
----------------------------------------
Sum Lengths = 20676
Sum Paths = 10377
thread_id, time_taken
0, 0.000008
```

```
final length[i, j]
[  0][ 71][150][237][286][340][428][172][218][254]
[283][  0][ 79][166][215][269][357][101][147][183]
[204][275][  0][ 87][136][190][278][ 22][ 68][104]
[441][229][308][  0][ 49][103][191][259][305][341]
[392][180][259][ 52][  0][ 54][142][210][256][292]
[338][126][205][ 95][ 43][  0][ 88][156][202][238]
[250][ 38][117][204][253][307][  0][ 68][114][150]
[182][253][332][419][468][522][610][  0][ 46][ 82]
[136][207][286][373][422][476][564][ 85][  0][ 36]
[100][171][250][337][386][440][528][272][318][  0]
----------------------------------------
final via[i, j]
[999][  1][  1][  1][  1][  1][  1][  1][  1][  1]
[  2][999][  2][  2][  2][  2][  2][  2][  2][  2]
[  7][  7][999][  3][  3][  3][  3][  7][  7][  7]
[  4][  4][  4][999][  4][  4][  4][  4][  4][  4]
[  5][  5][  5][  3][999][  5][  5][  5][  5][  5]
[  6][  6][  6][  4][  4][999][  6][  6][  6][  6]
[  7][  1][  1][  1][  1][  1][999][  7][  7][  7]
[  8][  8][  8][  8][  8][  8][  8][999][  8][  8]
[  9][  9][  9][  9][  9][  9][  9][  7][999][  9]
[  0][  0][  0][  0][  0][  0][  0][  0][  0][999]
----------------------------------------
Sum Lengths = 20676
Sum Paths = 10377
thread_id, time_taken
0, 0.000668
1, 0.000585
2, 0.000569
3, 0.000554
4, 0.000537
5, 0.000515
6, 0.000451
7, 0.000504
Time taken (in seconds) : 0.001009
```

```
final length[i, j]
[  0][ 71][150][237][286][340][428][172][218][254]
[283][  0][ 79][166][215][269][357][101][147][183]
[204][275][  0][ 87][136][190][278][ 22][ 68][104]
[441][229][308][  0][ 49][103][191][259][305][341]
[392][180][259][ 52][  0][ 54][142][210][256][292]
[338][126][205][ 95][ 43][  0][ 88][156][202][238]
[250][ 38][117][204][253][307][  0][ 68][114][150]
[182][253][332][419][468][522][610][  0][ 46][ 82]
[136][207][286][373][422][476][564][ 85][  0][ 36]
[100][171][250][337][386][440][528][272][318][  0]
----------------------------------------
final via[i, j]
[999][  1][  1][  1][  1][  1][  1][  1][  1][  1]
[  2][999][  2][  2][  2][  2][  2][  2][  2][  2]
[  7][  7][999][  3][  3][  3][  3][  7][  7][  7]
[  4][  4][  4][999][  4][  4][  4][  4][  4][  4]
[  5][  5][  5][  3][999][  5][  5][  5][  5][  5]
[  6][  6][  6][  4][  4][999][  6][  6][  6][  6]
[  7][  1][  1][  1][  1][  1][999][  7][  7][  7]
[  8][  8][  8][  8][  8][  8][  8][999][  8][  8]
[  9][  9][  9][  9][  9][  9][  9][  7][999][  9]
[  0][  0][  0][  0][  0][  0][  0][  0][  0][999]
----------------------------------------
rank, start_node, end_node, time_taken
0, 0, 1, 0.003648
1, 2, 3, 0.003054
2, 4, 4, 0.003641
3, 5, 5, 0.003655
4, 6, 6, 0.003657
5, 7, 7, 0.003659
6, 8, 8, 0.000059
7, 9, 9, 0.003041
Sum Lengths = 20676
Sum Paths = 10377
Total Time Taken: 0.003797
```

4.2 Speedup

To understand the effects of parallelizing and distributing this algorithm, we conducted tests over a combination of two parameters, namely with different input graphs and over different combinations of compute power. The serial implementation serves as our baseline for comparison. In total, 28 test combinations were performed manually, where each total is recorded as the average of 5 runs. Testing multiple speedup scenarios was especially intriguing due to the results noticed in the correctness evaluation above, as it is seen that the total time taken was higher in the parallel case versus the serial case.

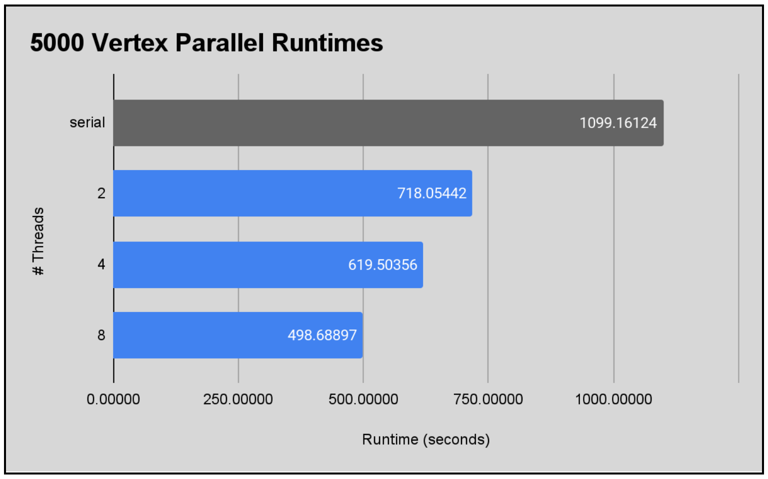
Since the serial implementation runs in $O(n^3)$ time, it is immediately evident that for small inputs (vertices, edges), the runtime is acceptable. For graphs with under 100 vertices, the serial implementation is faster because it takes longer to set up threads. Once the graph inputs rise, however, the serial implementation quickly becomes noticeably and significantly slower. Therefore the improvement in speedup is especially important for larger graphs. This is still logical, as the real world implementation for this algorithm is generally in large datasets, including big data. It is less likely that a parallelized algorithm be deployed on a simple small-scale graph.

For the parallel implementation with threads, there is an immediate impact on the speedup for larger datasets. Even by just using two threads, we see substantial speedup compared to the serial case:

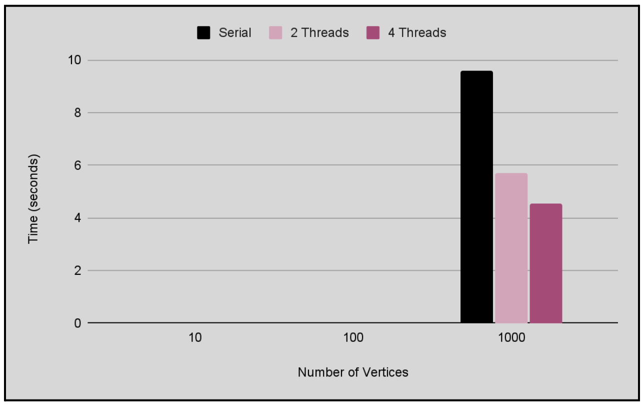
$$= \frac{Ex.Time\ Serial\ System}{Ex.Time\ on\ p - Processor\ System\ (Tp)}$$
$$= \frac{1099.16124}{718.05442}$$
$$= \mathbf{1.5\ x}$$

And for the 8 thread case,
$$= \frac{1099.16124}{498.68897}$$
$$= \mathbf{2.2\ x}$$

While the algorithm still runs in O(n^3) time, the execution time is reduced rather noticeably as processing power is increased.



Note that the statistics presented above graphically are for the more real world 5000 graph case. Speedup is still noteworthy for the 1000 case, however there appears to be negligible or indeterminate effects on parallel code for the smaller graphs. We cannot conclude that the code has positive or negative effects for graphs of under 100 vertices. A summary of the statistics for the remaining experiments on the parallel code are seen below.

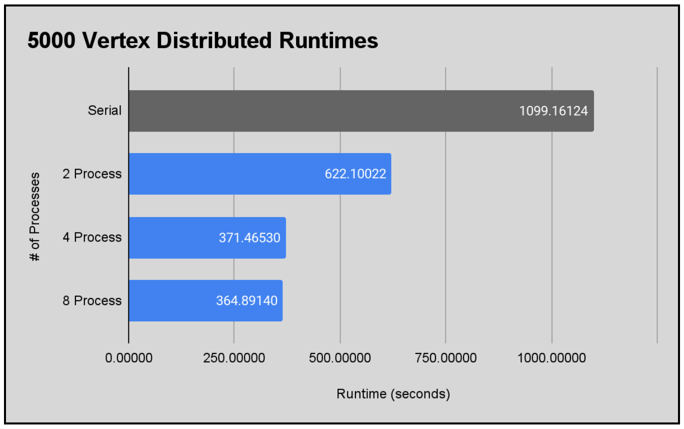


The test cases using apsp with distributed techniques and MPI reveal even more emphasized and pronounced results. At lower inputs, MPI is even slower than threads, but for larger input graphs it sees a sharp rise towards even better results. The speedup for the 1000 and 5000 vertex input graphs is noteworthy compared to the parallel execution, and quite significant compared to the serial implementation. Interestingly, there is also a noteworthy jump in speedup between 2 and 4 process executions, but barely any improvement in doubling the number of processes further. This is likely due to the fact that past a certain threshold *n*, the speedup provided by parallelization is negated by the communication overhead of MPI, as each iteration (*n* iterations) of the algorithm requires (n - 1) messages to be passed and received between processes, meaning the communication overhead grows at a rate of O(n^2).

For the 8 thread case, we have: $\frac{1099.16124}{364.8914}$ = **3.0 x** speedup

Comparing 8 threads to 8 processes, we have $\frac{498.6889}{364.8914}$ = **1.4 x** speedup

The remaining runtimes in each of the distributed test cases, over an average of 5 test runs, are expressed below (chart links to full dataset).



4.3 Test Suite

Test Suite (times, expressed in Seconds)				
Vertexes	**10**	**100**	**1000**	**5000**
Serial	0.000015	0.00613	9.57130	1099.16124
2 Threads	0.00073	0.00895	5.72326	718.05442
4 Threads	0.00117	0.00836	4.55659	619.50356
8 Threads	0.00271	0.01548	3.04915	498.68897
Ver	**10**	**100**	**1000**	**5000**
Serial	0.000015	0.00613	9.57130	1099.16124
2 Process	0.0012168	0.00538	5.34055	622.10022
4 Process	0.0232268	0.01379	3.12295	371.46530
8 Process	0.120818	0.13473	2.21960	364.89140

The test suite above is the summation of all of the collected data in the evaluation section and was used to form educated conclusions about the parallelization efforts. Each cell is composed of the average of 5 runs on the same input. Below are a subset of the sample executions.

Serial: `./apsp_serial  --inputFile input_graphs/10Nodes --rSeed 22`

Parallel: `./apsp_parallel --inputFile input_graphs/10Nodes --rSeed 22 --nThreads 8`

Distributed: `mpirun -n 4 ./apsp_distributed --inputFile input_graphs/10Nodes --rSeed 22`

5 Conclusion

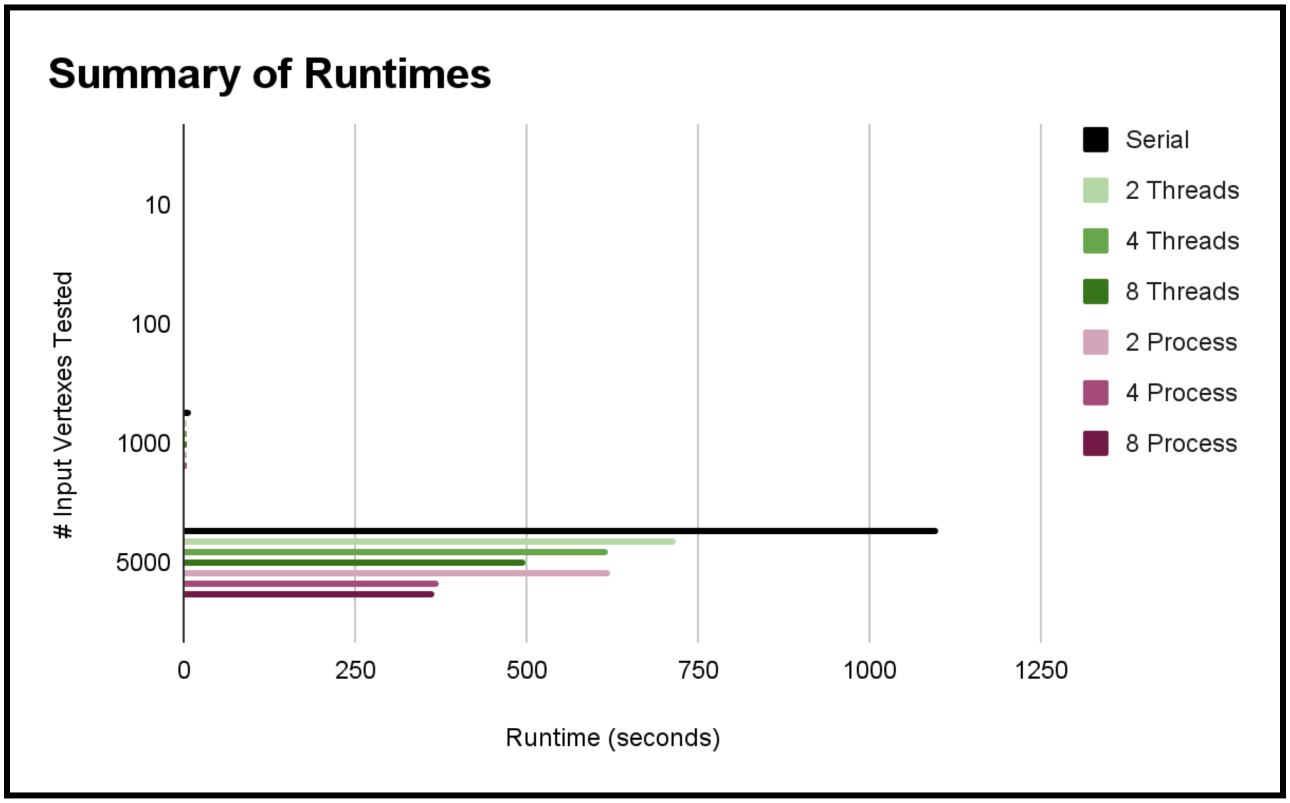
Our project achieved evident speedup and improvement among large test cases of the All-Pairs Shortest Paths (APSP) problem through parallel and distributed implementations, building upon the foundation of the serial Floyd-Warshall algorithm.

The parallel implementation, utilizing multiple threads to divide the computational workload, showcased promising speedup results, particularly for larger graphs. By parallelizing the algorithm, we observed significant reductions in execution time, with speedups reaching up to 2.2 times faster compared to the serial implementation with 8 threads. These results show promise when working with larger data sets as would be seen in real world scenarios, such as managing transportation networks or communication channels.

In the distributed implementation, we optimize using communication channels to find different patterns to maximize the performance of the algorithm. While this method performs negligibly slower on small inputs than both the parallel and serial versions of the algorithm, on large inputs we saw speedups reaching up to 3 times faster than the serial version, and 1.4 times faster than parallel on the same input. This type of performance is enough to conclude that the increased complexity of a distributed implementation is worth the effort as it warrants minutes worth of time savings.

Although, we can also reasonably conclude that parallelizing this algorithm using either technique is not necessarily worthwhile when working with small data sets, as the performance appears to be inconclusive and even potentially slower than serial. This indicates that the overhead of thread creation and synchronization might offset the benefits of parallel execution when there is not much work to partition. Therefore it is worth considering what the intended usage of the algorithm will be, and understand the effects of working with large ‘n’ on an $O(n^3)$ algorithm.

Ultimately the impacts of parallelizing this algorithm depends on the use case, inputs and intended results. While we have learned that serial code is not always the worst case, for higher inputs we have achieved our goal of getting noteworthy speedup that makes the efforts of parallelizing both by using threads and MPI worthwhile.



References

Aini, A., & Salehipour, A. (2012). Speeding up the Floyd–Warshall algorithm for the cycled shortest path problem. *Applied Mathematics Letters*, *25*(1), 1–5. https://doi.org/10.1016/j.aml.2011.06.008

GfG. (2024, April 4). *Floyd Warshall algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

Mouatadid, L. (2016). *DP: All pairs shortest paths, the Floyd-Warshall algorithm*. DP: All Pairs Shortest Paths, The Floyd-Warshall Algorithm. http://www.cs.toronto.edu/~lalla/373s16/notes/APSP.pdf

Wikimedia Foundation. (2024, March 12). *Floyd–Warshall algorithm*. Wikipedia. https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm