Comparison of Parallel and Serial Execution of Shortest Path Algorithms

Ahmed Taha Ameen[1], Ahmed M. Fouad[1], Emil Mourad Matta[1], Rawan Abdelkhalek Elkhishen[1], Shorouk Abdelfattah Alalem[1] , Aya Mohamed Safan[1] , Islam Tharwat Abdel Halim[2,3]
[1]*School of Engineering and Applied Sciences, Nile University, Giza, Egypt*
[2]*School of Information Technology and Computer Science, Nile University, Giza, Egypt*
[3]*Center of Informatics (CIS), Nile University, Giza, Egypt*
{Ah.Ameen, Ahm.Mohamed, E.Mourad, R.Elkhishen, S.Abdelfattah, Asafan, ihalim}@nu.edu.eg

***Abstract*—Shortest Path Algorithms are an important set of algorithms in today's world. It has many applications like Traffic Consultation, Route Finding, and Network Design. It is essential for these applications to be fast and efficient as they mostly require real-time execution. Sequential execution of shortest path algorithms for large graphs with many nodes is time-consuming. On the other hand, parallel execution can make these applications faster. In this paper, three popular shortest path algorithms - Dijkstra, Bellman-Ford, and Floyd Warshall - are both implemented as serial and parallel programs and tested on various problem sizes. The performance of these algorithms is evaluated by comparing their execution times. To achieve parallelization, the OpenMP (Open multiprocessing) framework is employed.**

Index Terms— Bellman-Ford, BFS, Complexity, Dijkstra, Floyd Warshall, MPI, OpenMP, PVM

I. Introduction

In the modern world, the Shortest Path Algorithms set of algorithms is crucial [1]. It has numerous uses, including network design, route finding, and traffic consultation. These applications must be quick and effective because they typically need to be executed in real-time. For huge graphs with plenty of nodes, shortest path techniques must be executed sequentially. On the other hand, these apps may run more quickly when executed in parallel. In order to solve large-scale problems within a reasonable time frame, it is essential to incorporate parallel versions of shortest-path algorithms. In many real-world scenarios, the size of the networks being analyzed can be massive, and the computation required to find the shortest path can be very time-consuming [2].

In general, parallel versions of these algorithms can distribute the computational workload across multiple processors, allowing for significant speedup compared to sequential implementations. This can be especially important in time-critical applications where decisions need to be made quickly, such as in emergency response scenarios or real-time traffic management. Additionally, parallel implementations can provide a more cost-effective solution for large-scale problems, as they can be run on high-performance computing systems, reducing the time and resources required to find solutions [3]. In short, the implementation of parallel versions of shortest-path algorithms is critical for addressing the challenges posed by large-scale problems and improving the efficiency and effectiveness of decision-making processes [4].

There are several approaches for writing parallel versions of shortest-path algorithms, each with its own strengths and weaknesses. One popular approach is to use a divide-and-conquer strategy, where the network is partitioned into smaller subproblems that can be solved independently and then combined to form the final solution [5]. Another approach is to use a parallel implementation of a sequential algorithm, where the algorithm is executed on multiple processors in parallel. This can be done by parallelizing the computations or by parallelizing the data structures used by the algorithm [2]. Another approach is to use a parallel version of a specific shortest path algorithm, such as Bellman-Ford or Dijkstra's algorithms, which have been specifically designed for parallel execution [6]. Additionally, there are hybrid approaches that combine elements of multiple techniques to achieve the best possible performance. Writing parallel versions of shortest path algorithms can be challenging, as it requires a deep understanding of the algorithms themselves, as well as the parallel programming models and hardware architecture. However, the benefits of having parallel implementations of these algorithms make the effort well worth it.

Open Multi-Processing (OpenMP), Parallel Virtual Machine (PVM), Message Passing Interface (MPI), and other parallel programming environments are only a few examples of the many parallel programming models and environments available [7]. In this paper, Dijkstra's, Bellman Ford's, and Floyd Warshall's algorithms are three shortest path algorithms that we propose parallel versions for. The parallel model used for this is the input decomposition model. The algorithms will be implemented using Open Multi-Processing (OpenMP), and their performance will be examined. These systems are scalable, allowing for the effective execution of both demanding sequential and parallel workloads. They may be adjusted to the available budget and computational needs. For the parallelization of the shortest path techniques in this research, we employed OpenMP.

The structure of the paper is as follows. Section 2 gives a summary of the background information and related research in the area. Section 3 details the proposed solution and technical merit, including a thorough explanation of the algorithms and methods used. The performance of the proposed solution is evaluated and analyzed in Section 4, where we present the results and discuss the strengths and weaknesses of the approach. Finally, Section 5 provides a conclusion and future work.

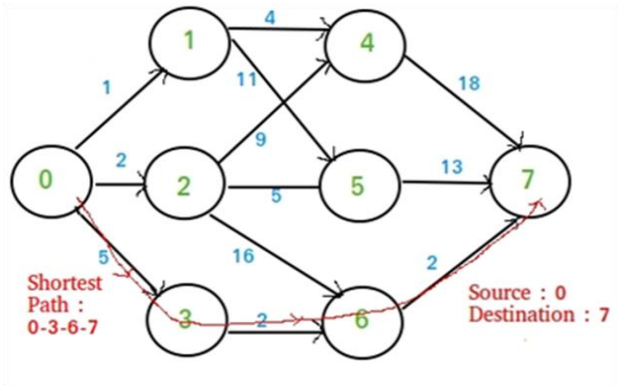

Fig. 1. Path from vertex 0 to vertex 7 of a graph.

II. Background and Related Work

A. Shortest Path Algorithms

One aim of the shortest path problem is to find a path that connects two vertices in a graph while minimizing the total sum of the weights of the edges. This can be defined as undirected, directed and mixed graphs. A path is a sequence of vertices ($P= (V_i, V_{i+1}, V_{i+2}\ldots\ldots, V_n)$) such that Vi isadjacent to V_{i+1} (the two vertices have a common edge). The length of the path P is (n-1). A path from vertex 0 to vertex 7 is marked in red in Figure 1.

There are several types of shortest-path algorithms, and each has its unique advantages and limitations, making it suitable for specific graph types and problem domains. Some of the most used types of shortest path algorithms are [8] [9] [10] [11]:

1) **Dijkstra's Algorithm:** In 1959, Edsger Dijkstra introduced an algorithm for graph searching that resolves thesingle-source shortest path problem in a graph where edge weights are non-negative, resulting in the shortest path tree. The shortest route between a single source vertex and all other vertices in the graph is determined using this algorithm. It operates by keeping track oftwo sets of vertices: one set with finalized shortest paths from the source vertex, and the other set with undetermined shortest paths. The algorithm repeatedly selects the vertex from the latter set with the lowest distance estimate and updates the distance estimates of its neighbors. The algorithm terminates when all vertices have been processed. Dijkstra's algorithm is suitable for graphs with nonnegative edge weights and has a time complexity of $O(n^2. \log n)$.
2) **Floyd-Warshall algorithm:** In 1962, Robert Floyd and Bernard Warshall introduced a graph search algorithm that resolves the all-pairs shortest path problem in a graph with positive or negative edge weights, provided that there are no negative cycles. One can use this algorithm to find the shortest path between all pairs of vertices in a graph. It works by iteratively improving an initial estimate of the shortest path between two vertices, by progressively relaxing the edges of the graph. The algorithm has a time complexity of $O(n^3)$ and is suitable for dense graphs.
3) **Bellman-Ford Algorithm:** In 1958, Richard Bellman and Lester Ford Jr. proposed a graph search algorithm that resolves the single-source problem even when edge weights are negative. This algorithm is utilized to discover the shortest path from a sole source vertex to all other vertices in the graph. Its operation includes iterative relaxation of the graph's edges while detecting negative cycles. The algorithm has a time complexity of O(n*m) and is suitable for sparse graphs
4) **A* Algorithm:** This is a heuristic search algorithm that combines the advantages of Dijkstra's algorithmand the Bellman-Ford algorithm. It uses a priority queueand an additional heuristic function to guide the search towards the target node, making it faster than Dijkstra's algorithm in practice.
5) **Bidirectional Search:** This is a technique that solves the shortest path problem by searching simultaneously from both the start and the target node. It can be faster than a single-directional search for large graphs, especially when the distance between the start and the target node is relatively small.
6) **Yen's Algorithm:** This is a variation of Dijkstra's algorithm that finds the k-shortest paths between two nodes, where k is a user-specified parameter. This algorithm canbe useful in cases where there are multiple solutions to the shortest path problem, and the user wants to explore all possible solutions.

The choice of shortest path algorithm depends on the specific requirements of the problem, including the type of graph, the weight function, and the desired performance characteristics. The challenge of determining the all-pairs shortest path in unweighted directed graphs is mitigated by reducing the task of discovering the shortest distance path. To that end, Running the breadth-first-search algorithm (BFS) from each node in the graph will produce the all-pair shortest path algorithm for unweighted graphs. The Floyd Algorithm and Dijkstra's Algorithm are two different Minimum Spanning Tree algorithms. Whenever a large graph with finite nodes and edges is involved, the issue must be solved sequentially. The authors in [12] found that the time complexity takes $O(n^3)$ and space complexity takes $O(n^2)$. All the paths from one vertex to another vertex of a graph are compared according to their lengths to find the shortest path by using shortest path Algorithms. The time complexity of some common shortest path algorithms is shown in Table [1], here, V refers to the total number of vertices and E denotes the total number of edges in the graph."

TABLE I. Time Complexity of Most Common Shortest Path Algorithms

Algorithm	Runtime
Dikstra's (with list)	$O(\|V\|^2)$
Floys-Warshall	$O(\|V\|^3)$
Bellman-Ford	$O(\|V\| . \|E\|)$
Topological Sort	$O(\|V\| + \|E\|)$
A∗	$O((\|V \| + \|E\|) \log\|V \|)$

B. Parallel Shortest Path Algorithms

The need for parallelization of shortest-path algorithms was raised because of the high scalability of the applications that depend on shortest-path algorithms that need to minimize the delay [13]. Shortest-path algorithms are highly applicable and utilized in various fields. For instance, mapping software such as Google Maps and Apple Maps rely on shortest-path algorithms. In addition, these algorithms play a critical role in road network planning, operations, and logistics research. They are also of great importance to computer networks, including the Internet.

Having a look at how google maps works may show how significant the problem is. Introduced in 2005 as a desktop application designed to assist users in navigating from "point A to point B," Google Maps has garnered over a billion, monthly active users. Over the past 15 years, it has evolved into an essential service that individuals rely on nearly every day. We all have this app pre-installed from the initial desktopsolution on our smartphones. To compute the shortest path from source point A to destination point B, Google Maps employs two graph algorithms, namely Dijkstra's algorithm and A* algorithm [14]. However, this algorithm has a limitation. Due to the nearly infinite or uncountable number of nodes in Google Maps, it may fail to perform efficiently as time and space complexity increase.

Parallel implementation of shortest-path algorithms refers to the implementation of these algorithms on parallel computing platforms, such as multi-core processors, GPUs, and distributed systems [15]. The main goal of parallelizing these algorithms is to achieve faster running times and improved scalability by leveraging the computing power of multiple processors. There are several approaches to parallelizing shortest-path algorithms, including [16] [17] [18]:

1) **Task Parallelism:** This approach involves dividing the computation into smaller tasks and executing each task on a separate processor. For example, in the case of Dijkstra's algorithm, each processor can be responsible for updating the distances of a portion of the nodes.
2) **Data Parallelism:** This approach involves executing the same computation on multiple processors, each operating on a different part of the input data. For example, in the case of the Bellman-Ford algorithm, each processor can be responsible for relaxing a portion of the edges.
3) **Hybrid Parallelism:** This approach combines task parallelism and data parallelism, and can be applied in various ways, depending on the specific requirements of the algorithm and the hardware platform.

The parallel implementation of shortest-path algorithms can be challenging due to several factors, including the need to maintain the consistency and correctness of the results, the need to synchronize the processing of different processors, and the communication overhead between processors. To overcome these challenges, various techniques have been proposed, including load balancing, data partitioning, and communication optimization. Overall, parallel implementation of shortest path algorithms can provide significant speedups for large-scale graphs and can make these algorithms practical for solving real-world problems. However, the design of the parallel algorithms is often a trade-off between parallelism, communication overhead, and algorithmic complexity, and requires careful consideration of the specific requirements of the problem and the hardware platform.

III. Proposed Solution and Technical Merit

In this study, three shortest path algorithms: Dijkstra, Bellman-Ford, and Floyd Warshall are implemented in serial and parallel programs on different problem sizes. The three algorithms are compared according to their execution time. For parallelization of these algorithms, Open multiprocessing (OpenMP) is used. All graphs implemented in this study are undirected graphs. The three algorithms will be explained in both sequential and parallel manner later in this section.

To make the terms more general, we can say that in a weighted graph G = (V, E, W), V is the set of vertices, E is the set of edges, and W represents the weight assigned to the edges between the nodes of the graph. In parallel execution, data parallelism will be applied to the three algorithms. the following directives are used in the parallel execution:

1) **Schedule** determines how the threads divide the iterations of the loop.
2) **Dynamic** denotes the method of dividing loop iterations into blocks of size s and dynamically scheduling them among the threads. Once a thread completes a block, it receives a new block dynamically.
3) **Critical** describes a part of code that must be executed by only one thread at a time.
4) **Atomic** describes that a specific memory location must be updated atomically, to avoid the case in which multiple threads attempt to write to it.
5) **For** indicates that the iterations of the loop following it must be executed in parallel.

A. Dijkstra's Algorithm

The single-source shortest path problem, which is solved by Dijkstra's Algorithm, uses a greedy strategy to determine the shortest path between a node and every other node in the graph. On graphs with non-negative edges, the algorithm can only be used.

1) Sequential Algorithm: Dijkstra's sequential algorithm is implemented through the following steps:

i. Setting a distance for each node, let the distance be zero to the source node and infinity to all other nodes.
ii. Creating a list of the unvisited nodes, adding all nodes to the unvisited list except the initial node.
iii. Setting the initial node as current.
iv. For the current node, calculate the tentative distances. Overwrite the value of the distance with the newly calculated distance if it is smaller than the previous distance.

```
#pragma omp single
1. Read the number of vertices //V
2. Enter the adjacency matrix //graph[][]
3. source = 0
4. for each vertex v in Graph: //Initialization distance graph
        dist[v] ← INF
        sptSet[v] ← 0
5. dist{source]=0
#pragma omp parallel for schedule(dynamic,chunk)
collapse(2)
6. for count from 0 to |V-1|
7.     for v from 0 to |V-1|
         #pragma omp critical
8.       if v equals 0
9.         min ← INF
10.      end if
11.      if sptSet[v] equals 0 and dist[v] <= min
12.        min=dist[v]
13.        u=v
14.      end if
15.  sptSet[u] ← 1
 #pragma omp parallel for schedule(dynamic,chunk)
16.  for v from 0 to |V-1|
17.    if (!sptSet[v] && graph[u][v] && dist[u] != INF
&&  dist[u]+graph[u][v] < dist[v])
           #pragma omp atomic
18.        dist[v] = dist[u] + graph[u][v];
```

Fig. 2. Parallel Dijkstra Algorithm.

```
#pragma omp single
1. Read the number of vertices //V
2. Read the number of edges //E
3. Read the source vertex, destination vertex and      weight
of the edge
4. Initialise the distance array
#pragma omp parallel for schedule(dynamic,chunk)
for i from 0 to |V-1|
   dist[i]=INT_MAX
dist[src]=0
5. Update the distance matrix
#pragma omp parallel for schedule(dynamic,chunk)
collapse(2)
for i from 1 to |V-1|
    for j from 0 to |E-1|
       u ← graph->edge[j].src
       v ← graph->edge[j].dest
       #pragma omp critical
if dist[u]!=INT_MAX and dist[u]+weight<dist[v]
              dist[v]=dist[u]
      end if
6. Detection for negative weight cycle
#pragma omp parallel for schedule(dynamic,chunk)
for i from 0 to E
   u ← graph->edge[j].src
   v ← graph->edge[j].dest
   #pragma omp critical
   if dist[u]!=INT_MAX and dist[u]+weight<dist[v]
      print "Negative weight cyle is there "
7. Print the final distance array
for i from 0 to |V-1|
   print dist[i]
```

Fig. 3. Parallel Bellman-Ford Algorithm.

v. Once all the neighbors of the current node have been inspected, it is designated as visited and excluded from the list of unvisited nodes, meaning that it will not be examined again.
vi. The algorithm should stop if either the destination node has already been marked as "visited," or the smallest distance among the unvisited nodes is "infinity."
vii. Select the unvisited node with the smallest distance and set it as the new current node, then go back to step 4 and repeat the process.

2) *Parallel Algorithm:* In the parallel execution of the Dijkstra algorithm, the algorithm goes through the same steps addressed in the sequential part. However, the update of neigh-boring nodes' distances is parallelized. The data is divided among threads. The pseudocode of the parallel algorithm is shown in Figure 2.

B. Bellman Ford's Algorithm

The Bellman–Ford algorithm also solves the single source shortest path problem. The difference between this algorithm and Dijkstra is that Bellman-Ford can deal with negative-weighted edges as it doesn't use a greedy approach.

1) Sequential Algorithm: Bellman-Ford's sequential algorithm is implemented through the following steps:

i. Assign a distance value to each node, with a distance of zero to the source node and a distance of infinity to all other nodes. The total number of times the edges have to be processed is equal to the number of vertices in the graph.
ii. The distances to the neighboring nodes of the source node are calculated.
iii. The first iteration gives all shortest paths, one edge long from the source node.
iv. The second iteration gives all the shortest paths, two edges long.
v. The algorithm is repeated for n times where n is the total number of edges.

2) Parallel Algorithm: In the parallel execution of Bellman-Ford's algorithm, the algorithm goes through the same steps addressed in the sequential part. However, the update of neighboring nodes' distances is parallelized. The data is divided among threads. The pseudocode of the parallel algorithm is shown in Figure 3.

C. Floyd Warshall's Algorithm

The Floyd-Warshall algorithm is an alternative method for solving the shortest path problem that calculates the shortest paths between every pair of vertices in the graph. Unlike Dijkstra's algorithm, it can accommodate graphs with negative edge weights, as long as there are no negative cycles. The result of the algorithm is a matrix that shows the shortest distances between each pair of vertices in the graph.

1) Sequential Algorithm: Floyd–Warshall's sequential algorithm is implemented through the following steps:

i. Constructing the distance matrix in which: All nodes have zero distances to themselves. Distance is infinity for unconnected nodes. Distance from node *x* to node *y* is equal to the distance of node *y* to node *x*.
ii. The weights matrix is generated according to the k formula $W_{ij} = \min(W_{ij}, W_{ik} + W_{jk})$ in which *i, j* represents the indices of the node in the matrix and *W* refers to the weight.
iii. The construction of matrix D1 involves keeping the first row and column unchanged while computing the other values using the formula mentioned above.
iv. Repeat step 3 for all the rows and columns (n times where n is the number of edges).

2) Parallel Algorithm: In the parallel execution of Floyed's algorithm, the algorithm goes through the same steps ad- dressed in the sequential part. However, the update of the distances of each 2D matrix is parallelized. The data is divided among threads. The pseudocode of the parallel algorithm is shown in Figure 4.

```
#pragma omp single
1. Read the number of vertices //V
2. Read the adjacency matrix for the graph //graph[][]
3. let dist be a |V| × |V| array of minimum distances
initialized to graph[][]
#pragma omp parallel schedule(dynamic,chunk) collapse(2)
for i from 0 to |V-1|
   for j from 0 to |V-1|
      dist[i][j] ← graph[i][j]
4. Update the dist[][] matrix
#pragma omp parallel schedule(dynamic,chunk) collapse(3)
for k from 0 to |V-1|
     for i from 0 to |V-1|
        for j from 0 to |V-1|
          #pragma omp critical
           if dist[i][k] + dist[k][j] < dist[i][j]
              dist[i][j] ← dist[i][k] + dist[k][j]
           end if
5. Print the final matrix
#pragma omp parallel for schedule(dynamic,chunk)
collapse(2)
for i from 0 to |V-1|
     for j from 0 to |V-1|
        #pragma omp critical
        if dist[i][j] = INF
           print INF
        else
            print dist[i][j]
        end if
```

Fig. 4. Parallel Floyd-Warshall Algorithm.

IV. Evaluation and Results

The previously addressed three algorithms were implemented using C language and OpenMP. They were tested on different problem sizes (graphs with a different number of vertices and levels) and their execution time was compared. There are two parameters were changed in each test as follows:

1. The number of vertices.
2. The degree of the nodes: the number of edges that are incident on the node or connected to it.

The parameters of the Dijkstra implementation were changed as shown in Table II and Figures 5 and 6 show the results of Dijkstra (4threads) and Dijkstra (8threads), respectively. The horizontal axis indicates the node's degree, while the vertical axis indicates the time in seconds. Each curve in those figures represents a running program that whether serial or parallel and its number of vertices for instance Parallel-2048 represents that the running program is a parallel algorithm with 2048 vertices. The parameters of the Bellman-Ford implementation were changed as shown in Table III and Figures 7 and 8 show the results of Bellman-Ford (4threads) and Bellman-Ford (8threads), respectively. Each curve in the previous graph represents a running program that whether serial or parallel and its number of vertices for instance. Finally, Table IV and Figure 9 are for the implementation of Floyed Warshel algorithm.

TABLE I. Parameters of Dijkstra Implementation

Number of vertices	Degree
512	32
1024	64
2048	128
4096	256
8192	512
16384	1024
-	2048
-	4096

TABLE III. Parameters ofBellman-Fordd Implementation

Number of vertices	Degree
100	30
200	50
500	100
1000	-

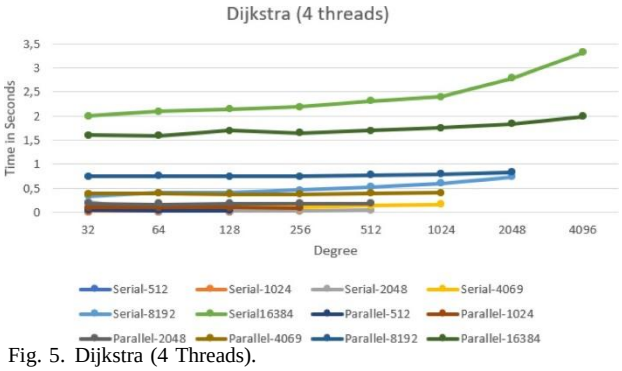

Fig. 5. Dijkstra (4 Threads).

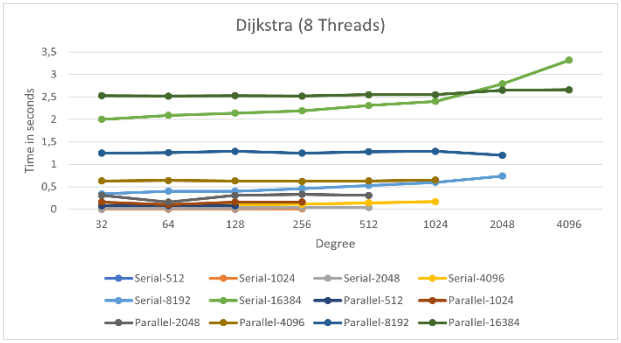

Fig. 6. Dijkstra (8 Threads).

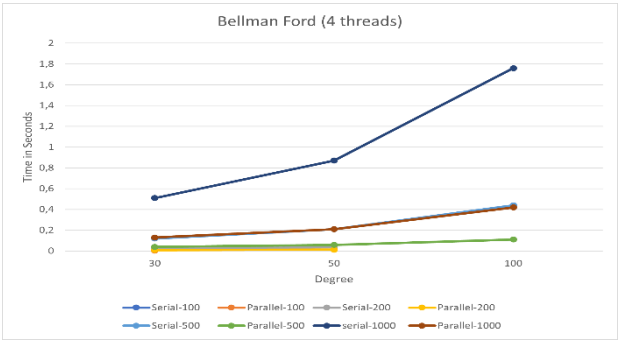

Fig. 7. Bellman Ford (4 Threads).

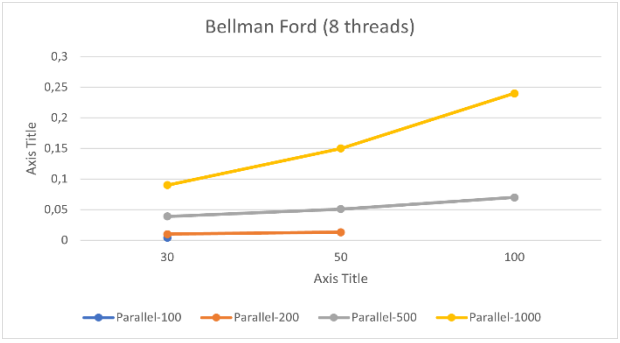

Fig. 8. Bellman-Ford (8 Threads).

TABLE IV. PARAMETERS OF FLOYD WARSHEL IMPLEMENTATION

Number of threads	Number of vertices
2	100
3	200
4	400
5	700
6	1000
7	1200
8	-

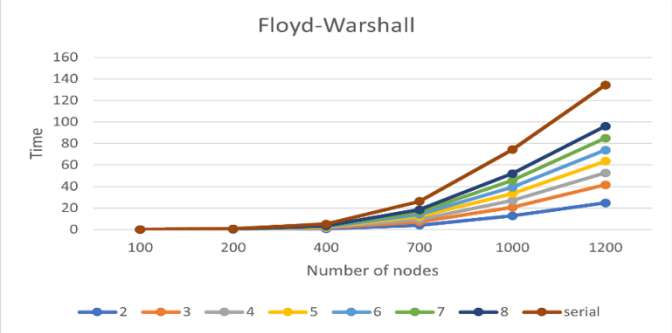

Fig. 9. Floyed Warshel.

v. CONCLUSION AND FUTURE WORK

In the Dijkstra algorithm, the time was relatively close to each other with the smaller graphs, but with extremely large graphs it was significantly faster than the sequential program. In addition to that the performance was better with four threads than with eight threads. In the literature, the largest number of vertices tried was 1000, and it was shown that the sequential program is much better than the parallel program. In Bellman Algorithms, the parallel program was better than the sequential, in addition, the greater number of threads were added the better the performance was. In the literature, it was shown that the best performance is with only 4 threads. In Floyd Warshall's algorithms the time was the same for all below 400 vertices, but more than that the parallel program was better. Running on two threads has given the best performance.This also contradicts the literature that shows that the serial program was much slower in time than the parallel program.

REFERENCES

[1] Karthik Karur, Nitin Sharma, Chinmay Dharmatti, and Joshua E Siegel. A survey of path planning algorithms for mobile robots. *Vehicles*, 3(3):448–468, 2021.

[2] Wei Wu, Takito Hayashi, Kato Haruyasu, and Liang Tang. Exact algorithms based on a constrained shortest path model for robust serial-batch and parallel-batch scheduling problems. *European Journal of Operational Research*, 307(1):82–102, 2023.

[3] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. Raja: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*, pages 71–81. IEEE, 2019.

[4] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)*, 8(1):1–70, 2021.

[5] Poornima Nookala, Zafar Ahmad, Mohammad Mahdi Javanmard, Martin Kong, Rezaul Chowdhury, and Robert Harrison. Understanding recursive divide-and-conquer dynamic programs in fork-join and data-flow execution models. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 407–416. IEEE, 2021.

[6] Alexei Vezolainen, Alexey Salnikov, Artem Klyuchikov, and Sergey Komech. Fast parallel bellman-ford-moore algorithm implementation for small graphs. In *Supercomputing: 8th Russian Supercomputing Days, RuSCDays 2022, Moscow, Russia, September 26–27, 2022, Revised Selected Papers*, pages 436–449. Springer, 2022.

[7] Arwa Alrawais. Parallel programming models and paradigms: Openmp analysis. In *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*, pages 1022–1029. IEEE, 2021.

[8] Madhumita Panda and Abinash Mishra. A survey of shortest-path algorithms. *International Journal of Applied Engineering Research*, 13(9):6817–6820, 2018.

[9] Sunita Kumawat, Chanchal Dudeja, and Pawan Kumar. An extensive review of shortest path problem solving algorithms. In *2021 5th International Conference on Intelligent Computing and Control Systems (ICICCS)*, pages 176–184. IEEE, 2021.

[10] Daniele Ferone, Paola Festa, Antonio Napoletano, and Tommaso Pastore. Shortest paths on dynamic graphs: a survey. *Pesquisa Operacional*, 37:487–508, 2017.

[11] Edsger W Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pages 287–290. 2022.

[12] Danny Z Chen and DT Lee. Solving the all-pair shortest path problem on interval and circular-arc graphs. In *Proceedings of 8th International Parallel Processing Symposium*, pages 224–228. IEEE, 1994.

[13] Kamesh Madduri, David A Bader, Jonathan W Berry, and Joseph R Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 23–35. SIAM, 2007.

[14] Daniel R Lanning, Gregory K Harrell, and Jin Wang. Dijkstra's algorithm and google maps. In *Proceedings of the 2014 ACM Southeast Regional Conference*, pages 1–3, 2014.

[15] Michelle Hribar, Valerie Taylor, and David Boyce. Performance study of parallel shortest path algorithms: Characteristics of good decompositions. In *Proc. 13th Ann. Conf. Intel Supercomputers Users Group (ISUG)*. Citeseer, 1997.

[16] Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara. A task parallel algorithm for finding all-pairs shortest paths using the gpu. *International Journal of High Performance Computing and Networking*, 7(2):87–98, 2012.

[17] Peter Heywood, Steve Maddock, Richard Bradley, David Swain, Ian Wright, Mark Mawson, Graham Fletcher, Roland Guichard, Roger Himlin, and Paul Richmond. A data-parallel many-source shortest-path algorithm to accelerate macroscopic transport network assignment. *Transportation Research Part C: Emerging Technologies*, 104:332–347, 2019.

[18] Yasir Arfat, Rashid Mehmood, and Aiiad Albeshri. Parallel shortest path graph computations of united states road network data on apache spark. In *Smart Societies, Infrastructure, Technologies and Applications: First International Conference, SCITA 2017, Jeddah, Saudi Arabia, November 27–29, 2017, Proceedings 1*, pages 323–336. Springer, 2018.