A Quantitative Analysis of Path Planning Algorithms using Parallel Computing

Aaron John Sabu
Dept. of Electrical Engg. (UG Student)
Indian Inst. of Technology Bombay
Mumbai, India
aaronjs@iitb.ac.in

Athul CD
Dept. of Mechanical Engg. (UG Student)
Indian Inst. of Technology Bombay
Mumbai, India
athulcd@iitb.ac.in

Ayan Sharma
Dept. of Electrical Engg. (UG Student)
Indian Inst. of Technology Bombay
Mumbai, India
170020023@iitb.ac.in

Vaibhav Malviya
Dept. of Electrical Engg. (UG Student)
Indian Inst. of Technology Bombay
Mumbai, India
vaibhav.m@iitb.ac.in

Shivasubramanian Gopalakrishnan
Dept. of Mechanical Engg. (Asst. Professor)
Indian Inst. of Technology Bombay
Mumbai, India
sgopalak@iitb.ac.in

***Abstract*—The demand for Door-to-Door (D2D) delivery services has seen a huge surge in the wake of the pandemic and lockdowns. Path planning algorithms will play a huge role in the future when delivery is performed autonomously using intelligent UGVs and electric cars. We investigate how path planning algorithms can be made faster using parallel computing techniques. We perform serial implementations of Dijkstra's algorithm, the Bellman-Ford algorithm, and the Floyd-Warshall algorithm following which we implement the same with the augmentation of parallel computing platforms such as OpenMP and CUDA. The algorithms are run on two types of databases: a very large graph based on the roadmap of New York state and smaller randomly-generated graphs of prescribed size. We present the simulation results and provide conclusions based on our inferences from the process of parallelization.**

***Index Terms*—path planning, parallel computing, OpenMP, CUDA, Dijkstra's algorithm, Bellman-Ford algorithm, Floyd-Warshall algorithm**

The code for this project has been provided in https://github.com/aaronjohnsabu1999/Path-Planner-ME766

I. Introduction

A large fraction of the developed and developing world has adopted the use of web mapping applications such as Google Maps and Bing Maps for choosing a route from one place to another. We take motivation from this problem and implement a number of commonly used roadmap-based path planning algorithms. We have implemented point-to-point path planning techniques based on Dijkstra's algorithm, the Bellman-Ford algorithm, and the Floyd-Warshall algorithm.

These implementations are further extended to include concepts that we have learned from the course on high performance scientific computing (ME 766, Spring 2021, IIT Bombay) and we demonstrate the effectiveness of parallel computing techniques in drastically increasing the speed of operations. Here, we incorporate features of the OpenMP parallel computing platform to increase the speed of our algorithms by multiple magnitudes. We also implement the algorithms using CUDA on graphical processing units (GPU) manufactured by NVIDIA. Finally, we compare and analyze the results of the implementations in OpenMP and CUDA with respect to the serial implementations.

We have used the roadmap of New York state [1] in the form of an undirected graph to test the implementations. While a real-world dataset helps visualize the working of the algorithms in real-life situations, this choice also draws motivation from the extensive information available on this roadmap. Furthermore, the database is extremely huge with more than 200,000 nodes and more than 700,000 edges, making it suitable for testing the effectiveness of parallelization. The database contains information pertaining to each start node and end node identifier, and the distance between the two points in the following format:

```
a <start> <end> <distance>
```

For algorithms which did not run on the New York state roadmap graph due to algorithm-inherent reasons and for the purpose of testing, we have run the algorithms on smaller randomly-generated graphs. In these cases, the number of edges has been arbitrarily chosen as thrice the number of vertices.

II. Implementation

A. Dijkstra's Algorithm

Dijkstra's algorithm is one of the most efficient algorithm that solves the single-source shortest path (SSSP) problem. It is a greedy algorithm that works on weighted, directed graphs with non-negative edge weights. After the initialization of the graph, the algorithm maintains a priority queue `Q` (keyed with distance from source) initially consisting of all vertices and a set `S`, initially empty. In each iteration until the queue is empty, the algorithm removes the vertex with minimum distance from the queue `Q` and adds it to `S` followed by relaxing all the connected vertices originating from the removed vertex [5]. The algorithm has a time complexity of $O((\mathrm{V}+\mathrm{E})\log(\mathrm{V}))$.

Its efficiency is highly dependent on the efficiency of the implemented data structures. For example, a Fibonacci-heap-based minimum priority queue was specifically developed to increase the speed of Dijkstra's algorithm. We have implemented the best possible serial implementation using C++'s Standard Template Library (STL) data structures. However, we realize that many of these STL data structures in the serial implementation are not compatible with OpenMP due to which we try to design customized data structures. However, these implementations are not efficient enough and the parallelizable components of the serial algorithm become negligible.

As a result, motivated by the concept of graph partitioning

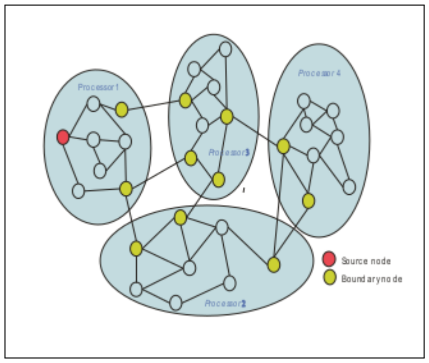


Fig. 1: Graph Partitioning (Dijkstra's algorithm) [2]

[2] as depicted in Fig. 1, we approach the problem by dividing vertices for different threads although not considering the topology of the graph. This idea is also motivated by a different OpenMP implementation of Dijkstra's algorithm [7] on which we have improved by using an adjacency list for graph representation. Although this method increases execution time, we obtain significant relative increase in speed with increasing number of threads. This parallel implementation predominantly parallelizes the process of finding the minimum node over different threads. We have also parallelized the edge relaxation step.

B. Bellman-Ford Algorithm

The Bellman-Ford algorithm is an algorithm with a time complexity of $O(\mathrm{VE})$ and is hence slower than Dijkstra's algorithm but faster than Floyd-Warshall algorithm. However, in constrast to the Dijkstra's algorithm, the Bellman-Ford algorithm works smoothly with negative weights [3]. It can also find and report negative weight cycles. Although distances are always positive, negative edge weights are useful in specific applications such as computer games where the health or experience points can increase or decrease depending on the player's path. Detecting negative weight cycles also helps prevent the algorithm from being stuck in a loop. These properties help the Bellman-Ford algorithm remain relevant for small- to medium-sized graphs despite being slower than Dijkstra's algorithm.

The algorithm is simpler and more naïve compared to Dijkstra's Algorithm since it relaxes all edges and the sequence of relaxation within a particular step does not matter. This provides the opportunity for parallelization [4]. The key part of the algorithm is to perform the relaxation of all edges $\mathrm{V}-1$ times and this step cannot be parallelized. All possible combinations of edge relaxations will converge to the same solution after $\mathrm{V}-1$ iterations.

Special care needs to be taken to ensure that all the threads from a particular loop are complete before moving on to the next step. The update statement should be `atomic` or within a `critical` section to ensure that the values are read/modified simultaneously. These are essential for the correctness of the parallel implementation.

C. Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is an example of dynamic programming that finds the shortest path lengths between all vertex pairs in a directed weighted graph with positive or negative edge weights but with no negative cycles [5]. The entire operation is performed in a single execution. However, it performs with the worst time complexity of the three algorithms at an order of V^3. Moreover, it has a space complexity of V^2. Hence, it is useful only for small dense graphs in which most or all pairs of vertices are connected by edges. It is predominantly used to calculate the minimum distances between all pairs at once in applications such as inversion of real matrices and computing the similarity between graphs.

The algorithm is usually formulated as three nested for-loops [6], the first running over a middle point `mid`, the second over a start point `start`, and the last over an end point `end`. The value of the distance between `start` and `end` is modified based on the triangular relation between the three points:

$$\begin{aligned}\mathrm{dist}(\mathrm{start},\mathrm{end}) = \min(&\mathrm{dist}(\mathrm{start},\mathrm{end}),\\ &\mathrm{dist}(\mathrm{start},\mathrm{mid}) + \mathrm{dist}(\mathrm{mid},\mathrm{end}))\end{aligned} \quad (1)$$

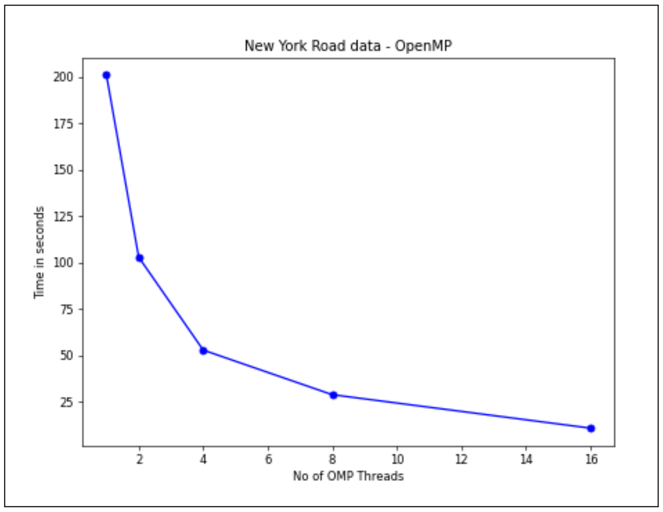
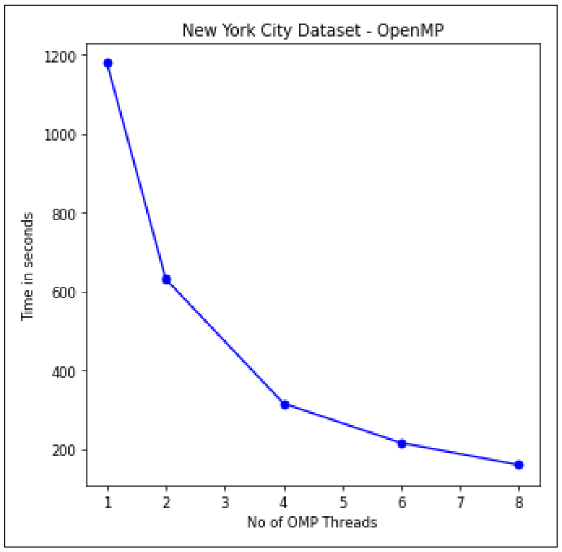
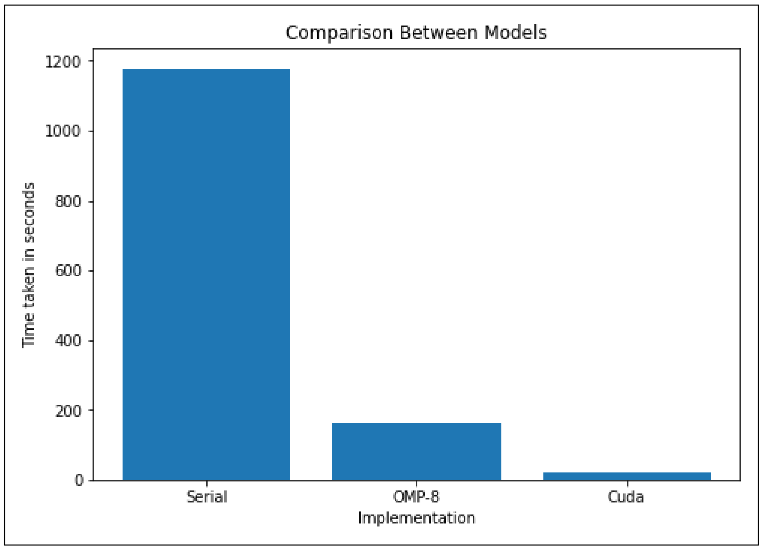


Fig. 2: OpenMP Timing Study for NY dataset (Dijkstra)



(a) OpenMP Timing Study



(b) Comparison between Models

Fig. 3: Bellman-Ford Algorithm: Results

III. Main Results

A. Dijkstra's Algorithm

The serial implementation of Dijkstra's algorithm using STL is very efficient, taking only 3 to 4 seconds (including read and write time) to run on the entire New York state roadmap graph. We develop a parallelizable implementation of the algorithm by distributing vertices to different threads of OpenMP. This increases the speed of the algorithm by 1.95, 3.79, 6.93, and 18.27 time for 2, 4, 8, and 16 threads respectively. The algorithm takes 201, 103, 53, 29, and 11 seconds when run on the New York state roadmap dataset using 1, 2, 4, 8, and 16 OpenMP threads respectively (Fig. 2).

B. Bellman-Ford Algorithm

As observed in Fig. 3a (8 CPU threads), using OpenMP provides a considerable speed-up in comparison to the serial version. The increase in speed is 1.86, 3.72, 5.44 and 7.33 times for 2, 4, 6 and 8 threads respectively. CUDA further provides a significant increase in speed by about 63 times with the edge relaxation step being the major contributor. This is depicted in Fig. 3b. It is, however, possible to create an adversarial test cases that can drastically impact the parallelization of the algorithm. Hence, though it performs well on real-world data, it may not be indicative of consistent performance across all types of graphs.

C. Floyd-Warshall Algorithm

Due to its time and space complexities and the subsequent overflow in data, running the Floyd-Warshall algorithm on the New York dataset is impossible for a normal personal computer. Hence, we have run the algorithm on randomly-generated graphs with sizes ranging up to 5120 vertices for OpenMP and 20480 vertices for CUDA. Running the OpenMP implementation over 16 CPU threads increased the speed of the algorithm about 15.88 times, and running the CUDA implementation made it faster by about 2100 times.

Fig. 5a demonstrates the timing study for the algorithm

Regression	**CUDA**		**OpenMP (8 threads)**	
	R^2	**Adj.** R^2	R^2	**Adj.** R^2
Linear	0.918447	0.908253	0.847152	0.828046
Quadratic	0.999986	0.999982	0.997333	0.996571
Cubic	0.999988	0.999983	0.999999	0.999999

when implemented using the CUDA platform. The linear, quadratic, and cubic trendlines of the data have been included from which it is visible that the implementation tends to be quadratic. This is further clear from the R^2 value of the CUDA implementation. On the other hand, the OpenMP implementation, represented in Fig. 5b, tends to be more cubic since its behavior is similar to the serial algorithm. The R^2 value for quadratic regression is much farther from 1 than in the case of the CUDA implementation whereas the cubic

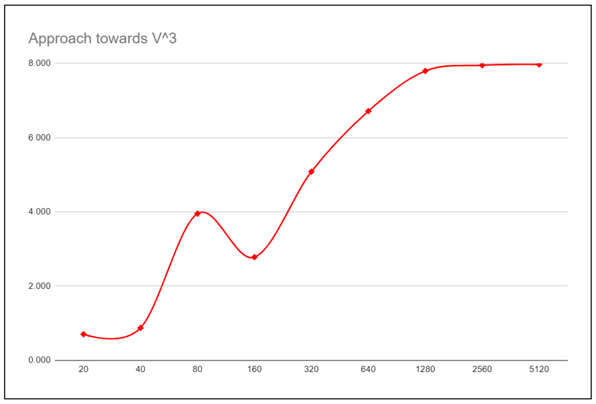
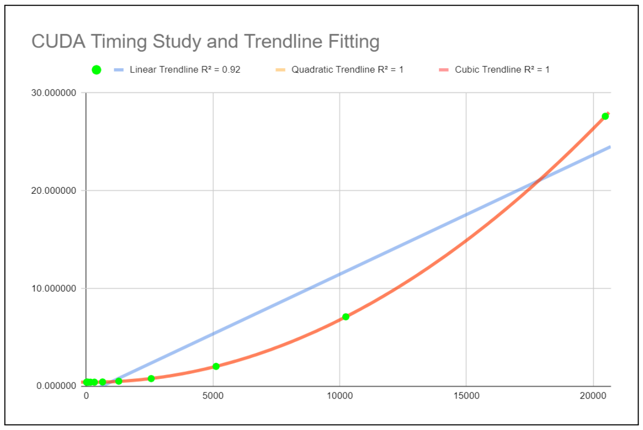
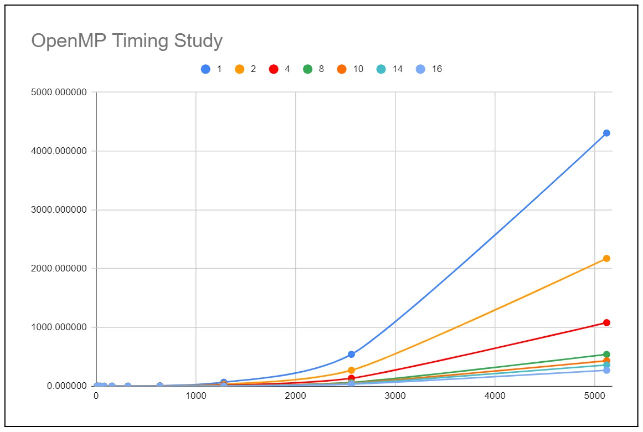


Fig. 4: Approach towards V^3 (Floyd-Warshall Algorithm)



(a) CUDA Timing Study and Trendline Fitting



(b) OpenMP Timing Study

Fig. 5: Floyd-Warshall Algorithm: Results

regression is even better than that in CUDA.
The algorithm, though run on randomly-generated graphs, can be tested for correctness by plotting the increase in time for different numbers of vertices as shown in Fig. 4. The increase in time is not proportional to V^3 for smaller number of vertices. However, it tends to be more and more cubic in time complexity as we move to larger graphs and time taken increases 8 times for a doubling in number of vertices.

IV. A Comparison of the Algorithms

	Dijkstra's	**Bellman-Ford**	**Floyd-Warshall**
Time Complexity	$O((V+E)\log(V))$	$O(VE)$	$O(V^3)$
Compute Neg. Values?	No	Yes	Yes
Detect Neg. Cycles?	No	Yes	No
Suitability	Large/Medium	Medium/Small	Small
Running Time (NY)	3 s	1178 s	∞
Speed-up (8 threads)	6.93	7.33	7.965

On comparing the three algorithms and their implementations, we learn that the most efficient and usable algorithm for the purpose of planning directions based on shortest distance is Dijkstra's algorithm due to its small time complexity and the subsequent decrease in total time taken for the implementations. However, it cannot be used in the presence of negative edge weights or cycles in the graph, hence making it irrelevant for other purposes that include such situations.
The Bellman-Ford algorithm runs reasonably fast for large datasets. However, it is not suitable for real-time computations and is only advantageous with negative weights and/or cycles.
The Floyd-Warshall algorithm is impractical for large graphs but works well for small dense graph where we require minimum distances between all vertex-pairs.
The fastest algorithm to run the New York state roadmap dataset [1] is Dijkstra's algorithm at 3 seconds (serial STL implementation). On the other hand, the Floyd-Warshall algorithm failed to run on the dataset due to memory overload. However, the latter algorithm showed a very high speed-up on parallelization while running on the randomly-generated graphs while Dijkstra's algorithm has the least parallelization capability.

V. Conclusion

We have developed the serial implementations of three path planning algorithms following which we have improved their speed by incorporating elements from shared memory parallel computing such as OpenMP (host) and CUDA (GPU). We have also presented the results for these implementations where we have demonstrated the optimal behavior of Dijkstra's algorithm despite the higher levels of parallelization in the other algorithms.
Our work may be expanded to the implementation of more diverse algorithms such as the A-star algorithm, rapidly-exploring random trees (RRT), and the D-star algorithm using parallel computing. Moreover, as per requirement, these algorithms may be implemented in more versatile parallel computing platforms such as OpenCL. This could be very useful for path planning problems for multiple agents.

Acknowledgment

The support and the resources provided by PARAM Sanganak under the National Supercomputing Mission, Government of India at the Indian Institute of Technology Kanpur are gratefully acknowledged.

References

[1] "9th DIMACS Implementation Challenge - Shortest Paths," url: http://users.diag.uniroma1.it/challenge9/download.shtml
[2] Y. Tang, Y. Zhang and H. Chen, "A Parallel Shortest Path Algorithm Based on Graph-Partitioning and Iterative Correcting," 2008 10th IEEE International Conference on High Performance Computing and Communications, 2008, pp. 155-161, doi: 10.1109/HPCC.2008.113.
[3] J. Bang-Jensen, G. Gutin, "Digraphs: Theory, Algorithms and Applications (1st ed.)," 2000, ISBN 978-1-84800-997-4.
[4] P. Agarwal and M. Dutta, "New Approach of Bellman-Ford Algorithm on GPU using Compute Unified Design Architecture (CUDA)," International Journal of Computer Applications, Volume 110 – No. 13, January 2015, doi: 10.5120/19375-1027
[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, "Introduction to Algorithms (1st ed.)," 1990 MIT Press and McGraw-Hill, ISBN 0-262-03141-8
[6] P. Z. Ingerman. 1962, "Algorithm 141: Path matrix," Commun. ACM 5, 11 (Nov. 1962), 556. doi: 10.1145/368996.369016
[7] J. Burkardt, "DIJKSTRA_OPENMP: Dijkstra Graph Distance Algorithm using OpenMP," https://people.sc.fsu.edu/~jburkardt/c_src/dijkstra_openmp/dijkstra_openmp.html