Path Planning 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 technique can be made faster using parallel computing techniques. We perform serial implementations on C++ 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 like Google Maps for routing 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 such as Dijkstra's algorithm, the Bellman-Ford algorithm, and the Floyd-Warshall algorithm.

These concepts 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 speed up our algorithms by multiple magnitudes. We also perform the algorithms using CUDA on the graphical processing unit (GPU). Finally, we compare and analyze the results of the implementation of each algorithm in OpenMP and CUDA with respect to the serial implementation.

We have used the roadmap of New York state [2] in the form of an undirected graph to test the algorithm 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 extremely useful 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 with the following format:

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

For the purpose of testing and for algorithms which did not run on the New York state roadmap graph for algorithm-inherent reasons, 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 a very efficient single-source shortest path (SSSP) greedy algorithm that works on graphs with positive edge weights. After initializing the graph, the algorithm selects the node with minimum distance and checks whether the distance between the neighboring connected nodes can be minimized by relaxation. This step is executed until distances to all nodes have been minimized.

The algorithm has a time complexity of $O((\mathrm{V}+\mathrm{E})\log(\mathrm{V}))$ that is achieved when efficient data structures such as minimum priority queues are implemented with a Fibonacci heap. We have implemented the best possible serial implementation using C++'s Standard Template Library (STL) data structures. However, we realize that many of these data structures are not compatible with OpenMP for parallelization due to which we use a customized data structure. However, the parallelizable components of the serial algorithm now become negligible.

Motivated by the concept of graph parallelization [1] as depicted in Fig. 1, we surpass this problem by dividing vertices for different threads although not considering the topology of the graph. Although this method increased execution time, we were able to obtain significant relative increase in speed with increasing number of threads.

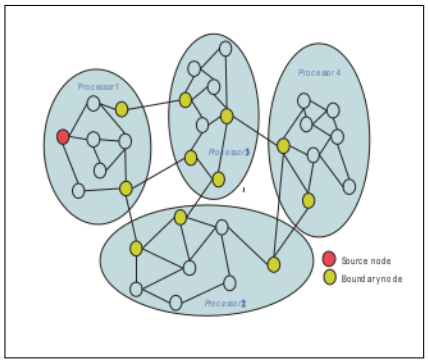


Fig. 1: Graph Partitioning (Dijkstra's algorithm) [1]

B. Bellman-Ford Algorithm

The Bellman-Ford algorithm is an algorithm with a time complexity of $O(\mathrm{VE})$, hence slower than Dijkstra's algorithm but faster than Floyd-Warshall algorithm. However, contrary to the Dijkstra's algorithm, the Bellman-Ford algorithm works smoothly with negative weights. It can also find and report negative 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.

Bellman ford is simpler compared to Dijkstra's Algorithm, which is a greedy algorithm which stores objects in a priority queue. But Bellman ford is more naive because it relaxes all edges and the sequence of relaxation within a particular step does not matter. This provides the opportunity for parallelisation. After $\mathrm{V}-1$ iterations, all possible combinations of edge relaxations will converge to the same solution.

The key part of the algorithm is to perform the relaxation of all edges $\mathrm{V}-1$ times and this step cannot be parallelized. The notion of "relaxation" is that the cost of the shortest path is an overestimate at every step. As shorter paths are found, the estimated cost is lowered. Eventually, the shortest path, if one exists, is found and all the distances are their optimal values. Once the relaxation process completed, a check for negative cycles is performed.

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 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. 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, computing the similarity between graphs, and optimal routing.

The algorithm is usually formulated as three nested for-loops, 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:

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

III. Main Results

A. Dijkstra's Algorithm

The serial implementation of Dijkstra's algorithm using STL is very efficient, taking only 3 seconds 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 1.97, 3.81, 5.68, and 6.89 time for 2, 4, 6, and 8 threads respectively. The algorithm takes 1984, 1086, and 653.8 seconds respectively when run on the New York state roadmap dataset using 4, 8, and 16 threads.

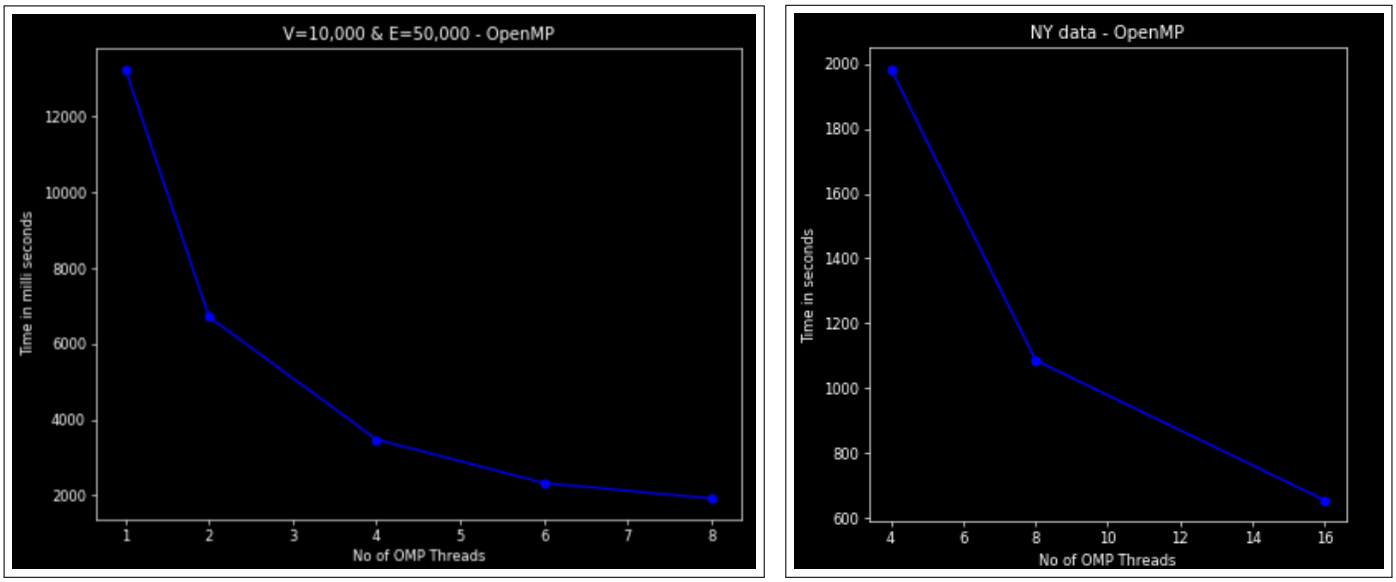
B. Bellman-Ford Algorithm

As observed in Fig. 3a, OpenMP provided good speed up compared to the serial version using 8 threads. OpenMP provided speed up values 1.86, 3.72, 5.44 and 7.33 for 2, 4, 6 and 8 threads respectively. CUDA provided a significant increase in speed by about 63 times in comparison to the serial algorithm with the edge relaxation step being the major contributor. This is depicted in Fig. 3b.

C. Floyd-Warshall Algorithm

Due to its time and space complexities and the subsequent overflow in data, running the algorithm on the New York database 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. 5b demonstrates the timing study for the algorithm when implemented using the CUDA platform. The linear,



(a) OpenMP Timing Study for New York database

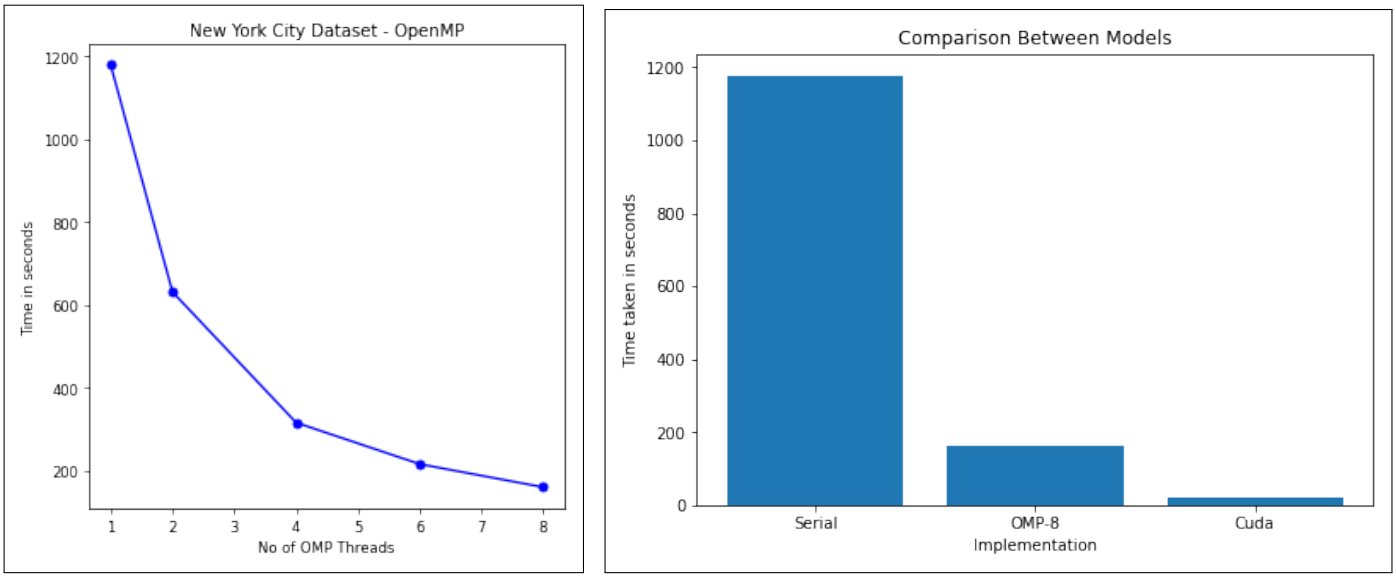
(b) OpenMP Timing Study for random graphs

Fig. 2: Dijkstra's Algorithm: Results

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

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. 5a, 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 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 the graph



(a) OpenMP Timing Study

(b) Comparison between Models

Fig. 3: Bellman-Ford Algorithm: Results

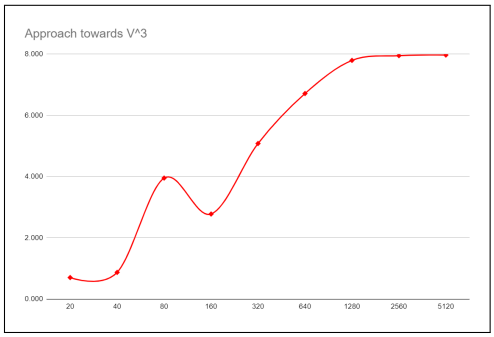


Fig. 4: Approach towards V^3

almost flattens at a factor of 8 on the y-axis for a factor of 2 on the x-axis.

D. A Comparison of the Algorithms

	Dijkstra's	Bellman-Ford	Floyd-Warshall
Time Complexity	$O((V+E)\log(V))$	$O(VE)$	$O(V^3)$
Negative Values Computability	No	Yes	Yes
Negative Cycles Computability	No	Yes	No
Suitability	Large/Medium	Medium/Small	Small
NY Dataset Running Time	3 s	1178 s	∞
Speed Up (8 threads)	6.89	7.33	7.965

On comparing the three algorithms and their implementation, we learn that the most efficient and usable algorithm for the purpose of planning directions based on shortest distance is the Dijkstra's algorithm due to its small time complexity and the subsequent decrease in total time taken for the implementations. It however cannot work if there are negative values 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 [2] was Dijkstra's algorithm at 3 seconds for completion. 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.

IV. Conclusion

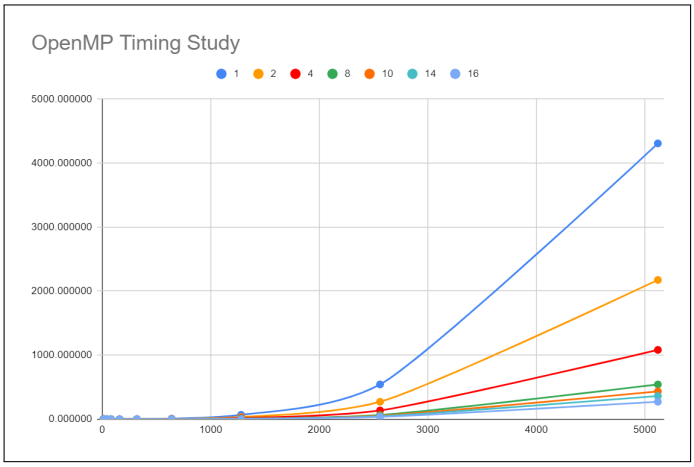
We have developed the serial implementations of a number of path planning algorithms following which we have improved their speed by incorporating elements from shared memory parallel computing such as OpenMP (host) and CUDA (GPU). We have also presented the results for these implementations where we 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 also be really helpful 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] 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.
[2] "9th DIMACS Implementation Challenge - Shortest Paths" URL: http://users.diag.uniroma1.it/challenge9/download.shtml



(a) OpenMP Timing Study

(b) CUDA Timing Study and Trendline Fitting

Fig. 5: Floyd-Warshall Algorithm: Results