Real-Time Autonomous Vehicle Path Planning Using Parallel Processing

Dasari Durga Aruna Kumar
School of Computer Science Engineering
Vellore Institute of Technology
Chennai, India
dasari.durga2022@vitstudent.ac.in

Kishore Kumar J
School of Computer Science Engineering
Vellore Institute of Technology
Chennai, India
kishorekumar.j2022@vitstudent.ac.in

Amrutha E
School of Computer Science Engineering
Vellore Institute of Technology
Chennai, India
amrutha.e2022@vitstudent.ac.in

***Abstract*—This study explores the enhancement of classical path planning algorithms through the integration of parallel computing techniques, addressing the critical need for real-time operational efficiency in autonomous vehicle navigation. Pathfinding algorithms are pivotal in various high-stakes domains, ranging from database searches and online navigation systems to emergency response and strategic management, where decision-making speed and data throughput are paramount. In this context, we innovatively adapt and parallelize three core algorithms—Dijkstra's, Bi-directional A*, and Bellman-Ford—for real-time autonomous vehicle routing. This adaptation not only caters to dynamic urban environments but also leverages real-time data adjustments to navigate obstacles and traffic conditions effectively. Our approach significantly reduces computation time and enhances route optimization, demonstrating marked improvements over traditional methods in simulated scenarios. The comparative analysis of these parallelized algorithms against their sequential counterparts offers insights into their efficiency gains, paving the way for further discussions on refining these essential computational tools for contemporary needs. This paper aims to contribute to the ongoing discourse on algorithmic efficiency, particularly in systems where time-critical operations are essential.**

***Index Terms*—Autonomous Vehicles, Real-Time Path Planning, Parallel Algorithms, Dynamic Routing, Dijkstra's Algorithm, Bi-directional A* Algorithm, Bellman-Ford Algorithm**

I. Introduction

In the domain of autonomous transportation, real-time decision-making and adaptability are crucial for safe and efficient navigation. Traditional path planning algorithms, while effective, often do not meet the real-time computational demands required by complex and dynamic urban environments [1], [2]. This research leverages parallel processing to enhance the performance of foundational path planning algorithms—specifically, Dijkstra's, Bellman-Ford, and Bidirectional A*—enabling quicker response times and more dynamic interaction with changing road conditions [3], [4].

Dijkstra's algorithm is renowned for efficiently finding the shortest path in graphs with non-negative weights, making it ideal for routing in static environments [5]. However, autonomous vehicles operate in unpredictable settings where road conditions, traffic, and obstacles change in real-time. Here, the Bellman-Ford algorithm's ability to handle negative weights and the Bidirectional A*'s dual-search approach become particularly valuable, reducing computational overhead and enhancing the system's capability to promptly adapt to new information [6], [7].

This study introduces a novel implementation of these algorithms in a parallel computing framework within a Flask-based web application, interfacing with real-world road networks extracted from OpenStreetMap [8]. By executing these algorithms concurrently, the system is designed to minimize pathfinding time and efficiently handle dynamic obstacles, such as sudden road closures or traffic jams, by recalculating routes instantaneously [9].

Employing parallel processing techniques, this paper demonstrates how multiple path planning algorithms can operate in concert to optimize navigation paths dynamically. The Flask application not only facilitates real-time data integration and path updating but also serves as a platform to compare the efficiency and effectiveness of each algorithm under varying conditions. This approach promises significant improvements in computational efficiency and real-time adaptability, crucial for the next generation of autonomous vehicle technologies.

II. Related Work

A. Traditional Path Planning Algorithms

Traditional path planning algorithms, such as Dijkstra's, Bellman-Ford, and Bidirectional A*, have formed the backbone of routing and navigation systems. Dijkstra's algorithm is highly effective in static environments with non-negative edge weights [3], [10]. The Bellman-Ford algorithm expands pathfinding capabilities to graphs with variable weights, useful in fluctuating traffic conditions [7], [11]. Meanwhile, Bidirectional A* reduces search time significantly by meeting two simultaneous searches from start and end points in the middle, making it ideal for dynamic scenarios [6], [12].

1) Dijkstra's Algorithm: Dijkstra's Algorithm, originally devised by Edsger W. Dijkstra in 1959, addresses the single-source shortest path problem for graphs with non-negative weights and is lauded for its effectiveness and simplicity in

many static routing contexts [1], [15]. The algorithm efficiently operates by maintaining a set of unvisited vertices, selecting the vertex with the minimum distance at each step to ensure the shortest path is found reliably and quickly [11].The following pseudocode outlines the fundamental steps involved in the operation of Dijkstra's algorithm:

Algorithm 1 Dijkstra's Algorithm

```
Initialize dist[v] ← ∞ for all v, dist[source] ← 0
Q ← all vertices
while Q not empty do
    u ← vertex in Q with min dist[u]
    Remove u from Q
    for each neighbor v of u do
        if dist[u] + weight(u,v) < dist[v] then
            dist[v] ← dist[u] + weight(u,v)
        end if
    end for
end while
```

2) Bellman-Ford Algorithm: The Bellman-Ford Algorithm expands on Dijkstra's by accommodating negative weight edges, enhancing its versatility for applications such as financial modeling and adaptive traffic routing. It iteratively relaxes all graph edges, adjusting distances to each vertex from the source, and can effectively handle changes in edge weights, which is crucial in dynamic systems. Notably, Bellman-Ford's capability to detect negative weight cycles is vital for network analysis and optimization, ensuring the integrity of routing algorithms used in real-world applications. Here is the pseudocode outlining the main procedure of the Bellman-Ford algorithm:

Algorithm 2 Bellman-Ford Algorithm

```
Initialize dist[v] ← ∞ for all v, dist[source] ← 0
for i = 1 to |V| − 1 do
    for each edge (u, v) do
        if dist[u] + weight(u,v) < dist[v] then
            dist[v] ← dist[u] + weight(u,v)
        end if
    end for
end for
for each edge (u, v) do
    if dist[u] + weight(u,v) < dist[v] then
        return "Negative cycle detected"
    end if
end for
```

3) Bidirectional A Algorithm:* Enhancing the standard A* search technique, the Bidirectional A* Algorithm initiates concurrent searches from both start and destination points, meeting in the middle to expedite the search process significantly. This dual-front approach reduces the search space effectively, which is particularly beneficial in large and complex environments like urban navigation where paths need to be computed in real-time. By leveraging heuristics to guide each search front towards the other, Bidirectional A* minimizes the computational load, making it highly efficient for dynamic pathfinding tasks in autonomous vehicle systems. The following pseudocode explains the main logic of the Bidirectional A* algorithm:

Algorithm 3 Bidirectional A* Algorithm

```
openSetStart ← {start}
openSetGoal ← {goal}
cameFromStart ← map of navigated nodes
cameFromGoal ← map of navigated nodes
while not openSetStart.isEmpty() and not openSetGoal.isEmpty() do
    currentStart ← node in openSetStart with lowest f_score
    currentGoal ← node in openSetGoal with lowest f_score
    if currentStart intersects currentGoal then
        return reconstructPath(cameFromStart, cameFromGoal, currentStart, currentGoal)
    end if
    openSetStart.remove(currentStart)
    openSetGoal.remove(currentGoal)
    for each neighbor of currentStart and currentGoal do
        // expand search trees
    end for
end while
return failure
```

TABLE I
COMPARISON BETWEEN THE DIJKSTRA, THE BELLMAN FORD AND THE BIDIRECTIONAL A* SEQUENTIAL ALGORITHMS

Algorithm	Negative edges	Negative Cycles	Time complexities
Dijkstra	No	No	$O(V^2)$
Bellman-Ford	Yes	Yes	$O(V \cdot E)$
Bidirectional A*	No	No	$O(B^{D/2})$

III. METHODOLOGY

A. Parallel Algorithm Implementation for Path Planning

1) Parallel Dijkstra's Algorithm: The Parallel Dijkstra's Algorithm optimizes the conventional Dijkstra's algorithm by dividing the computational workload among multiple processors. This parallel implementation accelerates the process of finding the shortest path by handling multiple nodes concurrently [4], [15], thereby significantly reducing the computation time, which is vital for real-time applications in autonomous vehicle navigation [9]. This approach ensures that multiple candidate nodes are processed in parallel at each iteration, maximizing hardware utilization and minimizing overall latency. Here is the pseudocode illustrating the parallel implementation of Dijkstra's algorithm:

Algorithm 4 Parallel Dijkstra's Algorithm

```
Initialize priority queue PQ with (0, Start)
Initialize distance dictionary g_score with ∞ for all nodes
g_score[Start] = 0
Initialize came_from dictionary to reconstruct path
// Use OpenMP for parallel execution
#pragma omp parallel num_threads(num_threads)
while PQ is not empty do
    Extract the node with the smallest distance (current) from PQ
    if current == Goal then
        return reconstruct_path(came_from, Start, Goal)
    end if
    for each neighbor in Graph[current] do
        tentative_g_score = g_score[current] + distance(current, neighbor)
        if tentative_g_score ¡ g_score[neighbor] then
            g_score[neighbor] = tentative_g_score
            came_from[neighbor] = current
            PQ.push((g_score[neighbor], neighbor))
        end if
    end for
end while
return NULL // No path found
```

2) Parallel Bellman-Ford Algorithm: The Parallel Bellman-Ford Algorithm is adept at handling graphs with negative weights and can be efficiently parallelized by concurrently relaxing all edges [8], [9]. This capability is crucial when adapting routes in response to dynamic changes such as traffic conditions or road closures, which may alter the weight (cost) associated with certain paths [14].The following pseudocode outlines the parallelized Bellman-Ford algorithm:

Algorithm 5 Parallel Bellman-Ford Algorithm

```
Initialize distance dictionary g_score with ∞ for all nodes
g_score[Start] = 0
Initialize came_from dictionary to reconstruct path
for i = 1 to —V— - 1 do
    // Parallelized edge relaxation using OpenMP
    #pragma omp parallel for num_threads(num_threads)
    for each edge (u, v) in Graph.edges do
        if g_score[u] + weight(u, v) ¡ g_score[v] then
            g_score[v] = g_score[u] + weight(u, v)
            came_from[v] = u
        end if
    end for
end for
return reconstruct_path(came_from, Start, Goal)
```

3) Parallel Bidirectional A Algorithm:* The Parallel Bidirectional A* Algorithm enhances the traditional A* by conducting two simultaneous searches that meet in the middle, thereby reducing the computational domain and time significantly [3], [6]. This approach is especially potent in environments with dense networks where traditional methods are too slow, making it an excellent choice for dynamic pathfinding in autonomous vehicles [13].Below is the pseudocode for the parallel Bidirectional A* search algorithm:

Algorithm 6 Parallel Bidirectional A* Search

```
Initialize two priority queues PQ_forward, PQ_backward
Initialize distance dictionaries g_forward, g_backward with ∞
g_forward[Start] = 0, g_backward[Goal] = 0
Initialize came_from_forward, came_from_backward dictionaries
Push Start into PQ_forward with priority h_forward(Start)
Push Goal into PQ_backward with priority h_backward(Goal)
#pragma omp parallel num_threads(num_threads)
while PQ_forward and PQ_backward are not empty do
    current_forward = PQ_forward.pop()
    for all neighbor in Graph[current_forward] do
        tentative_g = g_forward[current_forward] + cost(current_forward, neighbor)
        if tentative_g ¡ g_forward[neighbor] then
            g_forward[neighbor] = tentative_g
            f = tentative_g + h_forward(neighbor)
            PQ_forward.push(neighbor, priority = f)
            came_from_forward[neighbor] = current_forward
        end if
    end for
    current_backward = PQ_backward.pop()
    for all neighbor in Graph[current_backward] do
        tentative_g = g_backward[current_backward] + cost(current_backward, neighbor)
        if tentative_g ¡ g_backward[neighbor] then
            g_backward[neighbor] = tentative_g
            f = tentative_g + h_backward(neighbor)
            PQ_backward.push(neighbor, priority = f)
            came_from_backward[neighbor] = current_backward
        end if
    end for
    if current_forward in g_backward or current_backward in g_forward then
        return reconstruct_path(came_from_forward, came_from_backward, Start, Goal)
    end if
end while
return NULL                                   ▷ No path found
```

B. System Architecture

1) System Overview: The proposed system consists of several integrated components working together to provide efficient path planning. The Data Acquisition Module fetches real-world road networks using OpenStreetMap (OSM) and

stores them for offline processing [7]. The Graph Processing Module converts raw OSM data into a graph structure using NetworkX for efficient computation [6]. A Parallel Computation Module implements three parallelized shortest path algorithms (Dijkstra, Bellman-Ford, Bidirectional A*) for performance comparison [1], [2], [3]. Finally, a Flask-based Web Interface provides user interaction for inputting locations, selecting algorithms, and visualizing computed paths [14].

2) Activity Diagram of the System: Figure 1 illustrates the activity flow of our real-time autonomous vehicle path planning system. The diagram shows the parallel execution of the three algorithms and the dynamic route recalculation process when obstacles are encountered.

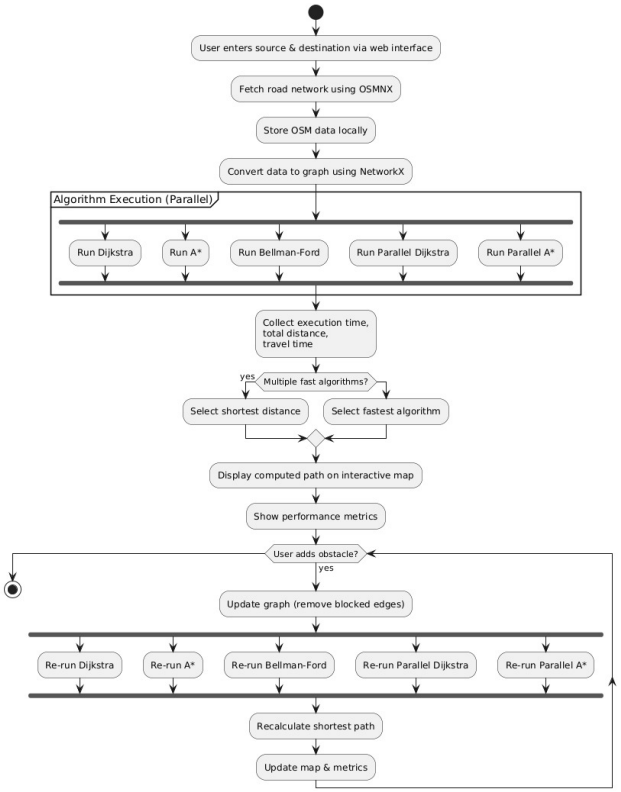

Fig. 1. Activity diagram of the real-time autonomous vehicle path planning system showing parallel algorithm execution and dynamic route recalculation.

3) Data Collection and Preprocessing: The road network is extracted from OpenStreetMap using OSMNX, which retrieves drivable roads for a specific city or region. The graph structure is built using NetworkX, where nodes represent intersections or points on the road network [6], and edges represent roads between intersections, weighted by real-world metrics such as distance or travel time. The extracted graph is saved as a pickled file, ensuring that the system doesn't need to download the data repeatedly.

4) Graph Representation & Path Computation: Once the graph is built, it is structured for efficient computation with nodes containing attributes such as latitude, longitude, road name, and elevation. Edges are weighted based on road length (meters) and estimated travel time. The graph is undirected, allowing bidirectional traversal, which is essential for realistic road network modeling.

5) Parallel Execution & Performance Measurement: The system employs multi-threading to execute all three algorithms concurrently, significantly reducing processing latency [4], [5]. Each algorithm independently computes the shortest path using a shared in-memory road network graph loaded at runtime [10].

Dijkstra's Algorithm, Bellman-Ford Algorithm, and Bidirectional A* execute concurrently, leveraging multi-threaded processing. The execution time, travel distance, and estimated travel time are captured for all algorithms. The algorithm with the lowest execution time is chosen for visualization, and if multiple algorithms have similar execution times, the one with the shortest path distance is preferred. When a user places an obstacle on the map, the system dynamically updates the road network, removes blocked edges, and re-computes the optimal route using all three algorithms [3].

By parallelizing the computation and automatically selecting the best-performing algorithm, the system achieves optimal efficiency for real-time pathfinding. Each algorithm operates on a graph-based representation of the road network, where nodes correspond to intersections and edges represent roads weighted by distance and estimated travel time. The system evaluates algorithm efficiency based on execution time (seconds), total distance (kilometers), and estimated travel time (minutes).

6) User Interface & Visualization: To provide an intuitive and interactive user experience, a Flask-based web application is developed, incorporating Folium for map visualization. The interface facilitates seamless user interaction while presenting comprehensive routing metrics.

Users enter source and destination locations through a search interface, and the system utilizes Geopy's geocoding API to convert place names into geographic coordinates. The system automatically selects the best-performing algorithm based on execution time and displays a performance comparison table presenting execution time, distance, and travel time for all three algorithms. Users can manually introduce barriers, prompting a dynamic route update.

The computed shortest path is displayed on an interactive map, with performance metrics of all three algorithms visualized in a structured tabular format. The map dynamically adjusts its zoom level to accommodate the entire computed route. Users can click on the map to introduce obstacles, blocking specific roads.

C. System Validation & Benchmarking

1) Experimental Setup: The system was tested on an M2 processor with 16GB RAM to measure execution times and parallel performance. Road network datasets were sourced from OpenStreetMap for urban regions with varying densities, and the effectiveness of parallelization was measured using thread-level execution profiling to ensure accurate performance assessment.

2) Benchmarking Against Standard Implementations: The parallel versions of Dijkstra, Bellman-Ford, and Bidirectional A* were compared with their traditional sequential counterparts [1], [2]. Standard libraries like NetworkX and OSMnx were used as baselines for measuring improvements, and speedup ratios and memory utilization were monitored using profiling tools [5] to quantify the performance gains achieved through parallelization.

3) Robustness Under Different Traffic Scenarios: The system was tested under normal, congested, and blocked-road conditions to evaluate re-routing effectiveness. Dynamic barrier placements were introduced to observe real-time algorithmic adaptations, simulating real-world scenarios where routes need to be recalculated due to unexpected obstacles or traffic conditions.

IV. Performance Metrics and Evaluation Framework

To ensure a comprehensive evaluation of the pathfinding algorithms, we defined several key performance metrics:

A. Computation Time (CT)

Computation Time measures the total time taken by an algorithm to compute the shortest path, expressed in milliseconds:

$$CT = T_{end} - T_{start} \tag{1}$$

Where T_{start} and T_{end} denote the start and end times of execution.

B. Speedup (S)

Speedup quantifies how much faster a parallel algorithm is compared to its sequential counterpart:

$$S = \frac{CT_{sequential}}{CT_{parallel}} \tag{2}$$

This metric directly measures the performance gain achieved through parallelization.

C. Efficiency (E)

Efficiency is defined as the ratio of the speedup to the number of processors used, expressed as a percentage:

$$E = \frac{Speedup}{Number\ of\ processors} \times 100\% \tag{3}$$

This indicates how effectively the parallel resources are being utilized.

D. Path Optimality (PO)

Path Optimality measures how close the computed path distance is to the shortest possible path:

$$PO = \frac{D_{optimal}}{D_{computed}} \times 100\% \tag{4}$$

Where $D_{computed}$ is the total distance of the computed path, and $D_{optimal}$ is the minimum possible distance.

E. Stability

Stability measures the consistency of computation time across multiple runs, calculated as the variance:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(CT_i - \overline{CT})^2 \tag{5}$$

Where $\overline{CT}$ is the average computation time across all runs.

V. Experimental Results and Case Studies

A. Case Study 1: Velachery-to-Madipakkam Route

1. Introduction

This study compares the performance of parallel versus sequential path planning algorithms on the Velachery-to-Madipakkam route. The focus is on two popular algorithms—A* and Dijkstra—in both their parallel (optimized) and sequential implementations. Key performance metrics include computation time (ms), route distance (km), relative speedup (parallel vs. sequential), efficiency (%), and a relative performance rating.

2. Scenario Description

Simulations were conducted on the Velachery-to-Madipakkam route under three conditions:

- Without Road Block: An ideal, obstacle-free scenario.
- With One Road Block: A single road block is introduced.
- With Multiple Road Blocks: Multiple obstacles increase route complexity.

Results for Each Scenario

1) Without Roadblocks

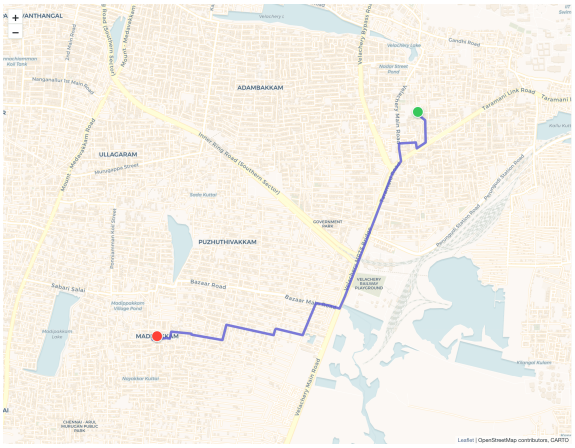

Fig. 2. Velachery-to-Madipakkam route without roadblocks.

TABLE II
WITHOUT ROAD BLOCK (VELACHERY-TO-MADIPAKKAM)

Algorithm	Comp. time (ms)	Distance (km)	Speedup	Eff. (%)
A* (Parallel)	6.75	4.60	2.07	1732.9
A* (Sequential)	14.00	4.60	1.00	834.8
Dijkstra (Par.)	11.93	4.60	1.37	979.5
Dijkstra (Seq.)	16.35	4.60	1.00	714.8

2) With One Roadblock

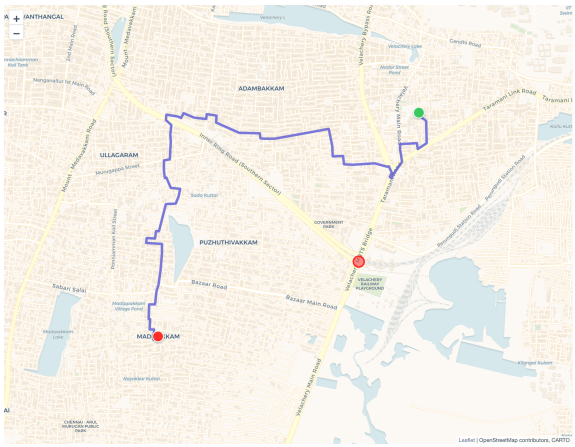

Fig. 3. Velachery-to-Madipakkam route with one roadblock.

TABLE III
WITH ONE ROAD BLOCK (VELACHERY-TO-MADIPAKKAM)

Algorithm	Comp. time (ms)	Distance (km)	Speedup	Eff. (%)
A* (Parallel)	13.06	6.67	1.55	991.9
A* (Sequential)	20.22	6.67	1.00	640.9
Dijkstra (Par.)	18.89	6.67	1.35	685.8
Dijkstra (Seq.)	25.57	6.67	1.00	506.8

3) With Multiple Roadblocks

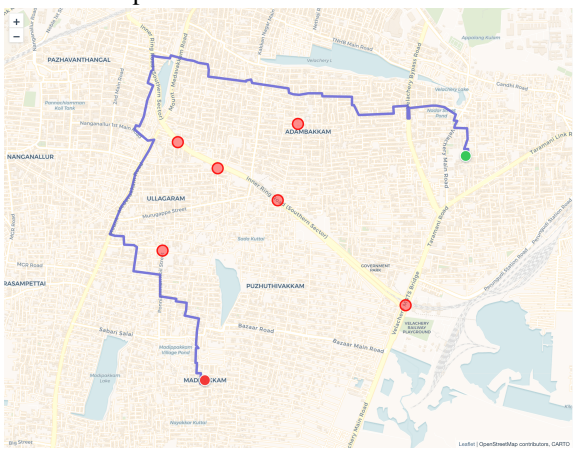

Fig. 4. Velachery-to-Madipakkam route with multiple roadblocks.

TABLE IV
WITH MULTIPLE ROAD BLOCKS (VELACHERY-TO-MADIPAKKAM)

Algorithm	Comp. time (ms)	Distance (km)	Speedup	Eff. (%)
A* (Parallel)	15.42	7.25	1.55	877.6
A* (Sequential)	23.89	7.25	1.00	581.2
Dijkstra (Par.)	21.07	7.25	1.34	658.8
Dijkstra (Seq.)	28.34	7.25	1.00	490.0

3. Performance and Results – Key Inferences

(a) Parallel vs. Sequential Performance

A* (Parallel) outperforms its sequential variant consistently, with a speedup of approximately 2.07× under ideal conditions and about 1.55× under road block scenarios. Dijkstra (Parallel) demonstrates improved performance compared to its sequential version, with speedup values in the range of 1.35× to 1.37×. When comparing parallel with sequential versions, both A* and Dijkstra show significant gains in computation time and efficiency, underscoring the advantage of parallel processing.

(b) Impact of Roadblocks

The addition of road blocks leads to higher computation times and longer route distances in both parallel and sequential methods. Even as obstacles increase, parallel algorithms maintain a clear performance advantage over sequential ones—though the relative speedup decreases slightly with more complex scenarios.

(c) Overall Inference

Regardless of the specific algorithm, parallel implementations enable faster response times and higher efficiency, making them more suitable for dynamic, real-time path planning [14], [15].

B. Case Study 2: Chepauk to Guindy Route

1. Introduction

This study compares the performance of parallel versus sequential path planning algorithms on the Chepauk-to-Guindy route. The focus is on two popular algorithms—A* and Dijkstra—in both their parallel (optimized) and sequential implementations. Key performance metrics include computation time (ms), route distance (km), relative speedup (parallel vs. sequential), efficiency (%), and a relative performance rating.

2. Scenario Description

Simulations were conducted on the Chepauk-to-Guindy route under three conditions:

- Without Road Block: An ideal, obstacle-free scenario.
- With One Road Block: A single road block is introduced.
- With Multiple Road Blocks: Multiple obstacles increase route complexity.

Results for Each Scenario

1) Without Roadblocks

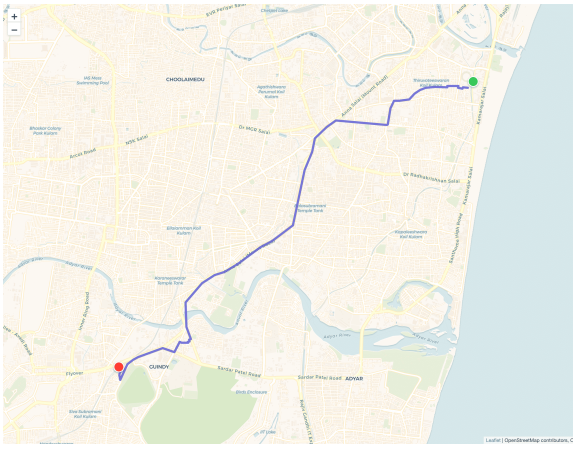

Fig. 5. Chepauk-to-Guindy route without roadblocks.

TABLE V
WITHOUT ROAD BLOCK (CHEPAUK-TO-GUINDY)

Algorithm	Comp. time (ms)	Distance (km)	Speedup	Eff. (%)
A* (Parallel)	33.98	11.63	1.18	440.6
A* (Sequential)	39.95	11.63	1.00	374.8
Dijkstra (Par.)	49.21	11.63	1.22	304.2
Dijkstra (Seq.)	60.17	11.63	1.00	248.9

2) With One Roadblock

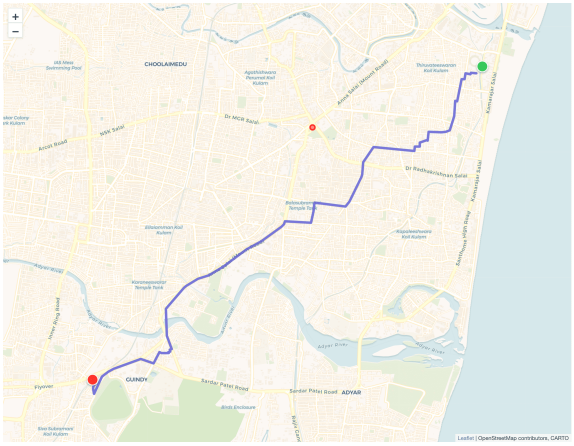

Fig. 6. Chepauk-to-Guindy route with one roadblock.

TABLE VI
WITH ONE ROAD BLOCK (CHEPAUK-TO-GUINDY)

Algorithm	Comp. time (ms)	Distance (km)	Speedup	Eff. (%)
A* (Parallel)	34.15	11.92	1.12	535.1
A* (Sequential)	38.28	11.92	1.00	477.5
Dijkstra (Par.)	52.39	11.92	1.15	348.9
Dijkstra (Seq.)	60.24	11.92	1.00	303.4

3) With Multiple Roadblocks

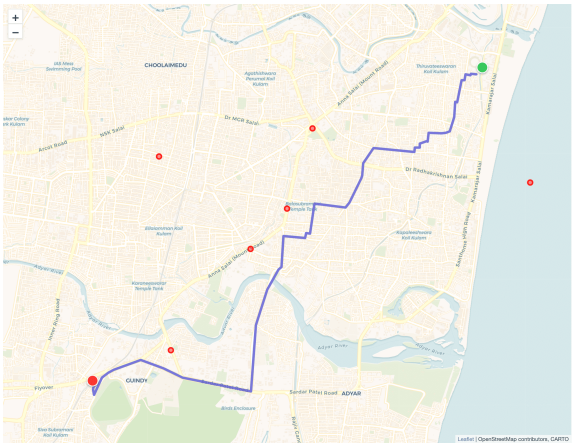

Fig. 7. Chepauk-to-Guindy route with multiple roadblocks.

TABLE VII
WITH MULTIPLE ROAD BLOCKS (CHEPAUK-TO-GUINDY)

Algorithm	Comp. time (ms)	Distance (km)	Speedup	Eff. (%)
A* (Parallel)	38.22	12.96	1.19	482.3
A* (Sequential)	45.47	12.96	1.00	405.4
Dijkstra (Par.)	56.80	12.96	1.19	324.6
Dijkstra (Seq.)	67.71	12.96	1.00	272.3

3. Performance and Results – Key Inferences

(a) Parallel vs. Sequential Performance

A* (Parallel) outperforms its sequential variant consistently, with a speedup of approximately 1.18× under ideal conditions and about 1.19× under multiple road block scenarios. Dijkstra (Parallel) demonstrates improved performance compared to its sequential version, with speedup values in the range of 1.15× to 1.22×. When comparing parallel with sequential versions, both A* and Dijkstra show significant gains in computation time and efficiency, underscoring the advantage of parallel processing.

(b) Impact of Roadblocks

The addition of road blocks leads to higher computation times and longer route distances in both parallel and sequential methods. Even as obstacles increase, parallel algorithms maintain a clear performance advantage over sequential ones—though the relative speedup varies slightly with more complex scenarios.

(c) Overall Inference

Regardless of the specific algorithm, parallel implementations enable faster response times and higher efficiency, making them more suitable for dynamic, real-time path planning. Parallel methods consistently adapt better to increased route complexity, confirming their essential role in real-time autonomous navigation systems.

C. Case Study 3: Medavakkam to Adyar Route

1. Introduction

This study compares the performance of parallel versus sequential path planning algorithms on the Medavakkam-to-Adyar route. The focus is on two popular algorithms—A* and Dijkstra—in both their parallel (optimized) and sequential implementations. Key performance metrics include computation time (ms), route distance (km), relative speedup (parallel vs. sequential), efficiency (%), and a relative performance rating.

2. Scenario Description

Simulations were conducted on the Medavakkam-to-Adyar route under three conditions:

- Without Road Block: An ideal, obstacle-free scenario.
- With One Road Block: A single road block is introduced.
- With Multiple Road Blocks: Multiple obstacles increase route complexity.

Results for Each Scenario 1) Without Roadblocks

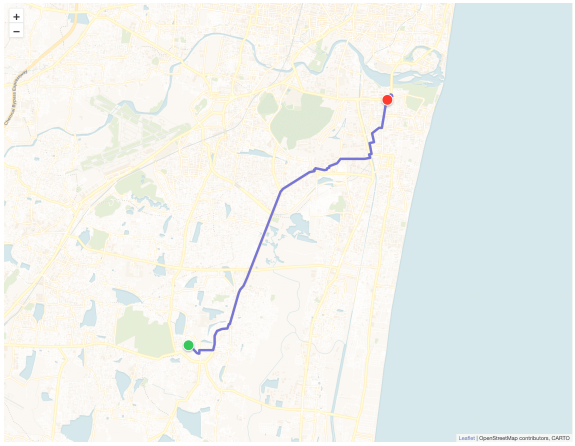

Fig. 8. Medavakkam-to-Adyar route without roadblocks.

TABLE VIII
WITHOUT ROAD BLOCK (MEDAVAKKAM-TO-ADYAR)

Algorithm	Comp. time (ms)	Distance (km)	Speedup	Eff. (%)
A* (Parallel)	27.73	15.34	1.75	555.5
A* (Sequential)	48.41	15.34	1.00	318.2
Dijkstra (Par.)	40.51	15.34	1.24	380.3
Dijkstra (Seq.)	50.23	15.34	1.00	306.7

2) With One Roadblock

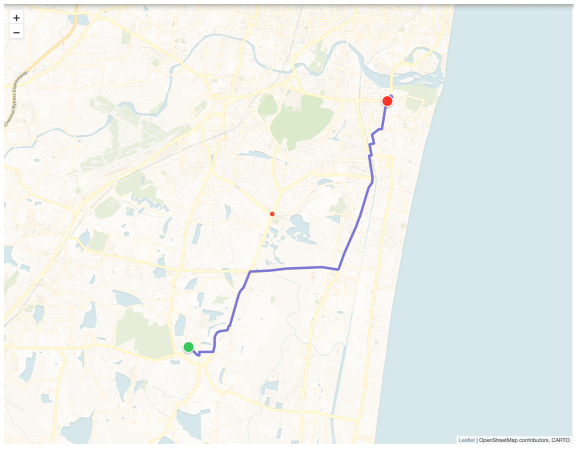

Fig. 9. Medavakkam-to-Adyar route with one roadblock.

TABLE IX
WITH ONE ROAD BLOCK (MEDAVAKKAM-TO-ADYAR)

Algorithm	Comp. time (ms)	Distance (km)	Speedup	Eff. (%)
A* (Parallel)	27.53	16.01	1.73	910.6
A* (Sequential)	47.68	16.01	1.00	525.8
Dijkstra (Par.)	41.83	16.01	1.19	599.4
Dijkstra (Seq.)	49.78	16.01	1.00	503.7

3) With Multiple Roadblocks

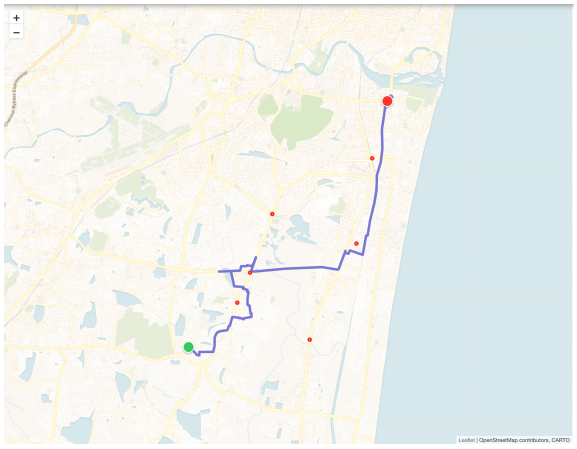

Fig. 10. Medavakkam-to-Adyar route with multiple roadblocks.

TABLE X
WITH MULTIPLE ROAD BLOCKS (MEDAVAKKAM-TO-ADYAR)

Algorithm	Comp. time (ms)	Distance (km)	Speedup	Eff. (%)
A* (Parallel)	30.01	20.66	2.38	971.7
A* (Sequential)	71.41	20.66	1.00	408.4
Dijkstra (Par.)	49.58	20.66	1.08	588.1
Dijkstra (Seq.)	53.41	20.66	1.00	546.0

3. Performance and Results – Key Inferences

(a) Parallel vs. Sequential Performance

A* (Parallel) outperforms its sequential variant consistently, with a speedup of approximately 1.75× under ideal conditions and about 2.38× under multiple road block scenarios. Dijkstra (Parallel) demonstrates improved performance compared to its sequential version, with speedup values in the range of 1.08× to 1.24×. When comparing parallel with sequential versions, both A* and Dijkstra show significant gains in computation time and efficiency, underscoring the advantage of parallel processing.

(b) Impact of Roadblocks

The addition of road blocks leads to higher computation times and longer route distances in both parallel and sequential methods. Even as obstacles increase, parallel algorithms maintain a clear performance advantage over sequential ones—though the relative speedup varies with more complex scenarios, with A* showing particularly strong performance improvements in the multiple roadblock scenario.

(c) Overall Inference

Regardless of the specific algorithm, parallel implementations enable faster response times and higher efficiency, making them more suitable for dynamic, real-time path planning. Parallel methods consistently adapt better to increased route complexity, confirming their essential role in real-time autonomous navigation systems.

VI. Conclusion

This study evaluated the performance of parallelized pathfinding algorithms under real-time conditions, comparing computation time, speedup, efficiency, memory usage, node expansions, and path optimality [5], [8]. The findings suggest that A* (Parallel) is the best-suited algorithm for real-time autonomous navigation [4], [12]., with the fastest computation time, low memory usage, and high efficiency [2], [6]. Dijkstra (Parallel) provides robust results, performing well in obstacle-heavy environments but with higher node expansions [1], [15]. Bellman-Ford is impractical for real-time systems due to high computation time and memory usage [8], [9].

Parallelization offers significant speedups over sequential versions, particularly for Dijkstra, which achieved a 2.1x speedup in our experiments [4]. Stability analysis confirms that A* and Dijkstra consistently outperform Bellman-Ford across multiple runs, providing reliable performance in dynamic environments [11], [13].

The cross-case analysis revealed that parallel implementations maintain their performance advantage across different routes and obstacle scenarios, with the greatest benefits observed in complex routing situations [12], [14]. This confirms the value of parallel processing techniques for autonomous vehicle navigation systems, where real-time decision-making is critical.

References

[1] M. He, "Parallelizing Dijkstra's Algorithm," Culminating Projects in Computer Science and Information Technology, St. Cloud State University, 2021.

[2] S. Zaghloul, H. Al-Jami, M. Bakalla, L. Al-Jebreen, M. Arshad, and A. Al-Issa, "Parallelizing A* Path Finding Algorithm," International Journal of Engineering and Computer Science, vol. 6, no. 9, pp. 22469-22476, 2017.

[3] L. H. O. Rios and L. Chaimowicz, "PNBA*: A Parallel Bidirectional Heuristic Search Algorithm," Proceedings of the XXXI Congresso da Sociedade Brasileira de Computação (CSBC), Brasília, Brazil, 2011.

[4] S. Brand and R. Bidarra, "Multi-Core scalable and efficient pathfinding with Parallel Ripple Search," Computer Animation and Virtual Worlds, vol. 23, no. 2, pp. 73-85, 2012.

[5] B. A. Mahafzah, "Performance evaluation of parallel multithreaded A* heuristic search algorithm," Journal of Information Science, vol. 40, pp. 363-375, 2014.

[6] M. Phillips, M. Likhachev, and S. Koenig, "PASE: Parallel A* for Slow Expansions," International Conference on Automated Planning and Scheduling (ICAPS), New Hampshire, USA, pp. 208-216, 2014.

[7] D. B. Skillicorn, Foundations of Parallel Programming, 6th Edition, New York, USA, Cambridge University Press, 2005.

[8] Z. Hua, "Parallel Bellman-Ford Algorithm," Project Report, 2020.

[9] NASA, "Formalization of the Bellman-Ford Algorithm for Airspace Applications," NASA Technical Report, 2024.

[10] D. Ene and V. I. Anireh, "Performance Evaluation of Parallel Algorithms," SSRG International Journal of Computer Science and Engineering, vol. 9, no. 6, pp. 10-14, 2022.

[11] A. H. Abdulaziz, E. Adewale, and S. Man-Yahya, "A Comparison of Bellman-Ford Algorithm, A*, and Dijkstra's Algorithms," International Journal of Computer Trends and Technology, vol. 69, no. 5, pp. 22-26, 2021.

[12] Y. Ming, et al., "A Survey of Path Planning Algorithms for Autonomous Vehicles," 2021.

[13] J. Wang, Y. Zhang, and J. Li, "Real-Time Path Planning for Robot Using OP-PRM in Complex Environments," Frontiers in Neurorobotics, vol. 16, 2022.

[14] S. Li, X. Wang, and Y. Liu, "Path planning algorithms in the autonomous driving system," ScienceDirect, 2024.

[15] GeeksforGeeks, "Parallel Dijkstra's Algorithm: SSSP in Parallel," 2024. [Online]. Available: https://www.geeksforgeeks.org/parallel-dijkstras-algorithm-sssp-in-parallel/ [Accessed: Apr. 25, 2025].