# Implementation and Analysis of Serial, Parallel, and Distributed A\* Algorithms for Maze Solving

Gourav Bharadwaj

Sukhmanpreet Singh 301575063

CMPT 431

Prof. Alaa Alameldeen

Project

Dec 4, 2024

**Abstract**

This report presents the implementation and evaluation of serial, parallel, and distributed versions of the A\* pathfinding algorithm applied to maze-solving tasks. Our approach investigates the computational efficiency and scalability of each implementation, providing insights into the advantages and limitations of parallel and distributed computing techniques in solving complex pathfinding problems.

**1. Introduction**

Pathfinding in complex environments is a fundamental problem in computer science, with applications ranging from robotics to geographical information systems. The A\* algorithm is widely used for its efficiency and accuracy in finding the shortest path between two points. This project explores the implementation of A\* in three different computing paradigms: serial, parallel using threads, and distributed using MPI, to understand their performance implications on modern computing architectures. The start and end points are hard coded i.e. from one end to another to keep the problem set simple as the freedom of choosing the points could add more complexity to the program itself.

**2. Background**

The A\* algorithm, is a best-first search that uses a heuristic to estimate the cost from the current node to the goal. The state of the art in A\* implementations varies from simple serial approaches to complex parallel and distributed systems. Serial implementations are limited by single-thread performance, while parallel and distributed versions leverage multiple cores and machines to improve computation time. The basic Euclidean distance as the heuristic function as other functions require further dynamic programming skills which we lacked. The use of multiple resources in parallel distribution means a lot of communication overhead, resource management and scalability concerns. This report will evaluate an implementation of the A\* Algorithm.

**3. Implementation Details**

The execution time of the algorithm is carefully measured using a high-precision timer that was used in our in class assignments, allowing for an accurate assessment of the parallel implementation’s performance benefits. The timing starts just before the algorithm begins processing and stops immediately after the path is successfully reconstructed or the algorithm terminates if no path is found. This measurement highlights the time-efficiency gains achieved through parallel processing.

**3.1 Serial Implementation**

This C++ implementation of the A\* pathfinding algorithm is designed to navigate mazes. The core of the implementation is the use of a priority queue to prioritize exploration based on the f-cost, which is a sum of the actual distance from the start (g-cost) and the estimated distance to the goal (h-cost), calculated using Euclidean distance as a heuristic. Each maze cell is encapsulated by the Cell struct that includes the cell’s coordinates and cost metrics. The algorithm commences from a designated start point and iteratively explores adjacent cells, updating the path costs and maintaining a record of visited cells to avoid redundant processing. Upon reaching the goal, the algorithm reconstructs the path from the goal to the start by tracing back through the parent links of each cell. The execution time is measured and displayed, alongside the resultant path if one is found. This implementation focuses on a methodical and efficient exploration of the maze, leveraging data structures that support quick priority-based retrieval of cells and effective tracking of the optimum path through complex mazes. The Cells are marked visited or not and saved in the maze itself as a modification to least significant bit. The Code itself follows the generic A\* algorithm implementation with keeping memory efficiency in mind. The heuristic of the cells are computed on the fly in order to save more space.

**3.2 Parallel Implementation**

The parallel implementation employs a thread-safe priority queue, a modification crucial for supporting concurrent operations by multiple threads. This queue manages the nodes to be explored, ensuring that the node with the lowest f-cost is processed next. The f-cost, as before, combines the g-cost (actual cost from the start node) and the heuristic estimate to the goal, which continues to use the Euclidean distance for its calculations.

To ensure that multiple threads can safely access the priority queue without causing data corruption, the queue is equipped with mutexes at its access points. The head\_lock and tail\_lock protect the front and back of the queue respectively, thus ensuring that no two threads can push or pop nodes simultaneously, maintaining the queue's integrity.

In the parallel version, the exploration of each node’s neighbors is delegated to separate threads. This means that for each node dequeued from the priority queue, multiple threads can simultaneously explore different directions (north, south, east, and west). This parallel exploration allows the algorithm to process large parts of the maze concurrently, drastically reducing the time to find the shortest path by avoiding the linear bottleneck of processing nodes one at a time.

Similar to the serial version, this implementation uses bit manipulation to mark nodes as visited directly within the maze's data structure, using the least significant bit to record this state. This approach minimizes the memory overhead by eliminating the need for additional data structures to track visited nodes. The 16th bit (using a bitwise OR operation with 0b1000000000000000) of the node’s value in the maze is set when a node is visited, ensuring that each node is processed only once.

To manage the shared resources efficiently—specifically the maze grid and the cost maps (g\_cost\_map)—mutexes are used to synchronize access. This is vital when threads update the cost of reaching new nodes or mark nodes as visited. By locking these resources during updates, the implementation prevents race conditions and ensures that all threads have a consistent view of the maze's state.

**3.3 Distributed Implementation**

**4. Evaluation**

Experiments were conducted using mazes of varying sizes and complexities. Each implementation's performance was measured based on the time taken to find a solution:

|  |  |  |  |
| --- | --- | --- | --- |
| Maze Size | Sequential (in seconds) | Parallel (in seconds) | Distributed(in seconds) |
| 20 | |  | | --- | | 0.0011718273 | | |  | | --- | | 0.0355401039 | |  |
| 40 | |  | | --- | | 0.0017049313 | | |  | | --- | | 0.1079778671 | |  |
| 60 | |  | | --- | | 0.0031440258 | | |  | | --- | | 0.1985390186 | |  |
| 80 | |  | | --- | | 0.0053839684 | | |  | | --- | | 0.3693070412 | |  |
| 100 | |  | | --- | | 0.0086519718 | | |  | | --- | | 0.6142678261 | |  |
| 200 | |  | | --- | | 0.0243799686 | | |  | | --- | | 2.4223921299 | |  |

**- Serial Version:**

The serial implementation of the A\* algorithm demonstrates a non-linear increase in computation time as the maze size grows, reflecting a polynomial escalation in complexity. Specifically, the time required to solve the maze increases by a factor of 1.6 for every additional 20 units in maze size, and by a factor of 3 when the maze size is doubled (e.g., from 20 to 40, 40 to 80, or 100 to 200). This pattern suggests that while the serial implementation scales effectively within a certain range, it faces practical limitations as the problem size continues to expand.

* **Detailed Assessment of Scalability:-** The scalability of this serial implementation is notable within a moderate range of maze sizes, from small to large. This adaptability is primarily due to the polynomial rate of time increase, which, though significant, remains manageable within these bounds. The results indicate that the algorithm can accommodate progressively larger mazes, but the growth in execution time could become overly burdensome for extremely large mazes. This limitation is chiefly due to the inherent constraints of single-threaded processing, which precludes the algorithm from benefiting from multi-core systems that could distribute the computational load more efficiently.
* **Influence of the Heuristic:-** The use of the Euclidean distance as a heuristic is pivotal to the efficiency of the algorithm. It directly estimates the distance to the goal, thereby effectively concentrating the search on the most viable paths. This heuristic minimizes unnecessary explorations, reducing the number of nodes that need to be processed and consequently easing the overall computational demand. However, as maze complexity and size increase, the heuristic's ability to limit exploration might be lessened, potentially leading to longer solve times.

**- Parallel Version:**

The observed performance data from the parallel implementation of the A\* algorithm reveals an unexpected increase in execution time, particularly when comparing smaller size increments and larger maze size doublings. For instance, the time increase by a factor of 4 from maze size 20 to 40 and a significant worsening from size 100 to 200 was contrary to expectations, which anticipated better performance scalability with larger mazes. Conversely, the mazes in the intermediate size range from 40 to 100 exhibited a decreasing factor of increased time, from 1.85 to 1.72 and then notably to 0.5, suggesting an optimal performance window within this range.

The pattern of execution time increase suggests a higher-degree polynomial trend in how the parallel algorithm scales with increasing maze size. Such a trend indicates that while the algorithm can handle moderate increases in maze size efficiently, it becomes disproportionately slower as the maze size continues to grow, especially beyond a certain threshold.

**Factors contributing to the slowdown**

* **Thread Overhead and Synchronization**: As the number of thread management increases to handle larger mazes, the cost associated with managing these threads and synchronizing their operation becomes a significant performance bottleneck. This includes the time taken for thread creation, destruction, and the synchronization mechanisms needed to ensure data consistency across threads.
* **Resource Contention**: The parallel algorithm's need to access shared data structures, such as the priority queue for open set management, can lead to contention, particularly as the complexity of the maze (and thus the number of threads interacting with shared structures) increases. This contention can drastically slow down the performance due to waiting times incurred during lock acquisition.
* **Load Imbalance**: The algorithm suffer from load imbalance, where some threads have more work than others, leading to inefficient utilization of computational resources. This is particularly problematic in large mazes where the pathfinding workload might not be evenly divisible by the number of threads.
* **Granularity:** This implementation was highly granular as the number of threads created for each task and the communication taken over them was higher than the actual computation of the task. The parallelized area was exploration but the path creation was done sequentially at the end. Due to this fine Grained granularity, the overhead to the computation work increases for each thread, thus making it more inefficient.

**- Distributed Version**:

**5. Conclusions**