Implementing Parallel AStar Search Algorithom On A Low Power Embedded System

List of Authors (Abdulraheem Aljarrah, Ala'aldin Hijaz, Daniel Llamocca Obregon)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: aaljarrah@oakland.edu, ahijaz@oakland.edu, llamocca@oakland.edu

A star is a search algorithm that search through a space of possibilities for an optimal solution out of all possible solutions. A star is typically applied to a path finding type of search problems.

In this paper we will implement the A star algorithm using sequential approach and parallel approach. We will examine different parallel approaches on the algorithm.

A time comparison between the different the sequential and parallel approaches were done, as a conclusion was that based on the grid size and dimensions one approach may be better than another approach.

Based on that conclusion, we recommend using a hybrid approach where switching between the sequential and parallel approach based on the grid parameters.

I. Introduction

A* is a search algorithm that search through a space of possibilities for an optimal solution out of all possible solutions. It is typically applied to a path finding type of search problems. Starting from a specific node of a graph, the algorithm aims to find a path to a given end node with the smallest cost. The algorithm does that by maintaining a tree of paths originating at the start node and extending those paths one node at a time until it reaches the end node [1].

The algorithm extends one path at a time, and to determine which path to extend it uses a cost function.

$$f(n) = g(n) + h(n)$$

The cost function f(n) is formatted by adding the cost from the start node to the current node (g(n)) and an estimation of the cheapest cost path from the current node to the end node (h(n)). h(n) is called the heuristic function.

The heuristic function depends on the problem itself, one key point when selecting the heuristic function is to not overestimate the cost to get to the end node. In this paper we will use the Manhattan distance [2] as our heuristic function.

In these days, many microcontrollers are multicores, and to cope with the advancement in technologies and algorithms many industries are switching to multicore embedded systems. To really be able to get benefits of these powerful microcontrollers you would need to make sure that your algorithms can run on a parallel system which may be a challenge for many different algorithms. The A* search algorithm is widely used for many different applications [3]-[6]. Therefore, in this paper, we are focusing on how to implement a parallel A* search algorithm.

In this paper, we will implement the A* algorithm using a sequential approach first, then we will implement different parallel approaches and examine the improvement on performance on different graph size. We will be using TBB.

In the ECE5900 class we have learnt how to parallelize a sequential algorithm and how to examine when it's going to shine. We had to learn how to use the OpenGL library [7] to be able do the graphical view. Also, this was our first time using the A* algorithm.

II. Methodology

A. Serial Algorithm

First, we created a Sequential approach of the Algorithm, which was our starting point. The Serial approach use this idea of open and closed sets. The open set contain possible nodes that are still candidates and not checked yet. The closed set contain nodes that we have already examined. Every loop, we examine a new node from the open set, this node shall have the lowest f(n), we look to it's neighbors and update their g(n), h(n) and f(n) and add them to the open set. The next loop, one of these neighbors will be our next candidate node.

Pseudocode:
```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node
n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a
hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the
cheapest path from start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n
currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0
    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our
current best guess as to
    // how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(1) time if openSet is a min-heap or a
priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to
neighbor
            // tentative_gScore is the distance from start to the neighbor
through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record
it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

Figure1, shows the A* algorithm in action, the orange circle is the starting point, the purple is the target, the green is a node in the open set, the red is a ndoe in the closed set and the black is a wall that we can't go through.

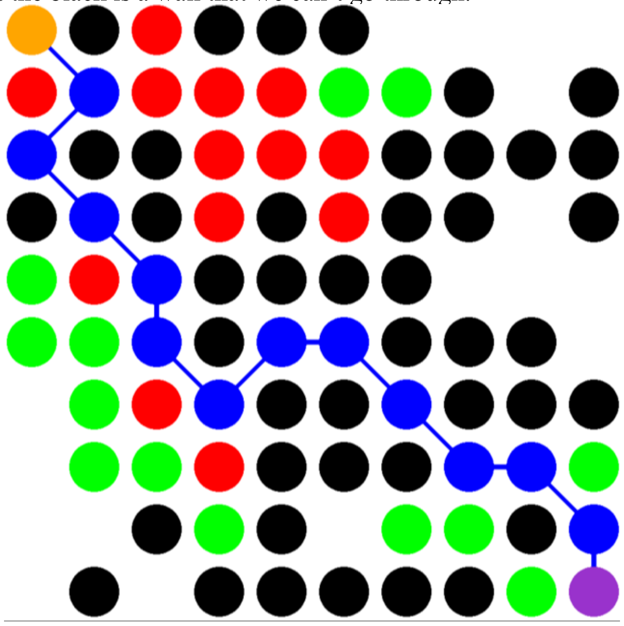
Figure 1: A*

B. *Parallel approach*

We have tried many different approaches for parallelization, however, at the end, a simple parallel for lop showed to be the most efficient and it showed a promising result compared with the sequential approach.

Pseudocode:

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node
n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a
hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the
cheapest path from start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n
currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our
current best guess as to
    // how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(1) time if openSet is a min-heap or a
priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to
neighbor
            // tentative_gScore is the distance from start to the neighbor
through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record
it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

As can be noticed from the pseudo code, we have used the same code we used for the sequential approach, but we added a parallel for loop (Grayed out area), and to make sure we don't access the same memory at the same time, we added a mutex (Red Area). This approach resulted in the most efficient parallel approach. In the next sections we will be discussing the experimental setup and covering the results.

III. Experimental Setup

As an experiment we have collected data using the Atom board, it has two cores, it was provided by the class. The other setup was a Dell XPS 15 9570 (Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz), it had 6 cores and 12 logical processors and a 32Gbytes of RAM.

On the Atom we had Ubuntu 12, on the Dell we had Ubuntu 20.

We expect the speed up on the Dell to be much higher compared with the Atom board. Also, we expect that the parallel approach will shine once we start examining large grids.

IV. Results

We run both approaches (the sequential and parallel) on both setups, captured result for different grids sizes. The sequential approach showed better performance when the grid size was smaller, on the other hand, the parallel approach showed improved performance when the grid became bigger and bigger.

During test and data collection, we found that using a rectangular shape grid showed an improved performance compared with using a square grid. The main reason of that is the fact that with a rectangular shape, we get to examine more data (wider path).

A. Atom board

The Atom board has two cores, it operates with a 600Mhz speed, Table 1 shows the result that we collected when running the algorithm with different grid sizes.

Table 1: Atom board result

nxm	TBB (Sec)	Sequential (Sec)
100 x 100	0.005	0.001
200 x 100	0.026	0.015
300 x 100	0.059	0.045
400 x 100	0.107	0.075
500 x 100	0.220	0.188
600 x 100	0.248	0.236
700 x 100	1.639	2.145
800 x 100	0.321	0.302
900 x 100	1.475	1.816
1000 x 100	0.677	0.743
1100 x 100	1.630	2.057
1200 x 100	0.862	0.969
1300 x 100	13.440	21.667
1400 x 100	2.280	3.094
1500 x 100	6.965	10.523
1600 x 100	7.072	10.308
1700 x 100	1.857	2.375
1800 x 100	5.298	7.343
2100 x 100	3.079	4.058
2300 x 100	8.433	11.324
2500 x 100	11.183	15.073
2700 x 100	152.816	235.984

Computing the speed up, shows that we achieved a max of 1.6 which does makes sense, looking the to the speed up in Figure 2 shows that the speed up is fluctuating a little bit, the reason behind that is the randomness of the added walls.

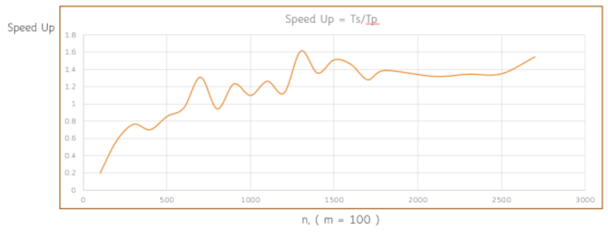

Figure 2: Atom SpeedUp

B. Dell XPS

We run the same experiments on the Dell XPS, and the results are shown in Table 2.

nxm	TBB (Sec)	Sequential (Sec)
300 x 300	0.0022	0.0007
600 x 300	0.0019	0.0012
900 x 300	0.048	0.0368
1200 x 300	0.185	0.178
1500 x 300	0.077	0.055
1800 x 300	0.269	0.252
2100 x 300	0.298	0.275
2400 x 300	1.377	1.728
2700 x 300	5.565	8.973
3000 x 300	2.141	2.750
3300 x 300	2.618	3.730
3600 x 300	2.513	3.195
3900 x 300	42.143	82.920
4200 x 300	12.839	20.831
4500 x 300	3.219	3.903
4800 x 300	29.040	49.776
5100 x 300	2.849	2.890
5400 x 300	6.182	8.429
5700 x 300	7.238	9.606
6000 x 300	10.455	13.155
6300 x 300	37.323	60.292
6600 x 300	108.858	202.161

Computing the speed up, shows that we achieved a max of 2 which is small knowing that we are running on 6 cores computer, we believe that the bigger the data set the more the speed up will improve. We also had a mutex inside the parallel part, so this will certainly limit our speed up.

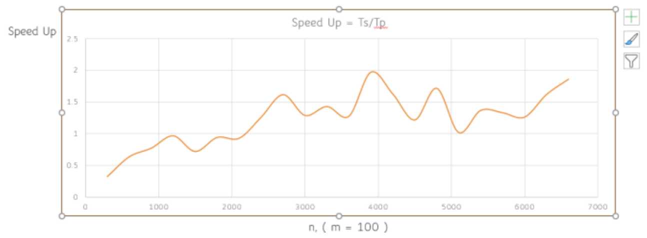

Figure 3: XPS SpeedUp

CONCLUSIONS

The A-Star algorithm is an implementation of a depth-first search algorithm. It can be applied to numerous path-finding algorithms and has a wide range of implementation variants.

Implementing the algorithm with a parallel approach allows for better execution times than serial implementation when inputting a large set of data. The maximum speedup on the XPS processor was ~2, while on the Atom processor it was ~1.6. The higher speedup by the XPS processor is mainly because the processor has more cores than the Atom processor. However, our parallel implementation was not the most optimized solution, further investigation in parallelizable portions of the algorithm can result in higher speedup rates.

Future work includes investigating more ways of parallel execution optimization and finding more parallelizable areas of the algorithm which can result in even higher speedup rates.

References

[1] J. Yao, C. Lin, X. Xie, A. J. Wang and C. -C. Hung, "Path planning for virtual human motion using improved a* star algorithm," *2010 Seventh International Conference on Information Technology: New Generations*, 2010, pp. 1154-1158, doi: 10.1109/ITNG.2010.53.

[2] M. D. Malkauthekar, "Analysis of euclidean distance and Manhattan Distance measure in face recognition," *Third International Conference on Computational Intelligence and Information Technology (CIIT 2013)*, 2013, pp. 503-507, doi: 10.1049/cp.2013.2636.

[3] I. S. AlShawi, L. Yan, W. Pan and B. Luo, "lifetime enhancement in wireless sensor networks using fuzzy approach and a-star algorithm," in *IEEE Sensors Journal*, vol. 12, no. 10, pp. 3010-3018, Oct. 2012, doi: 10.1109/JSEN.2012.2207950.

[4] A. Ghaffari, "An energy efficient routing protocol for wireless sensor networks using A-star algorithm." *Journal of applied research and technology*, vol. 12, no. 4, pp. 815-822, Aug. 2014.

[5] T. Chen, G. Zhang, X. Hu and J. Xiao, "Unmanned aerial vehicle route planning method based on a star algorithm," *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 2018, pp. 1510-1514, doi: 10.1109/ICIEA.2018.8397948.

[6] Z. Boroujeni, D. Goehring, F. Ulbrich, D. Neumann and R. Rojas, "Flexible unit A-star trajectory planning for autonomous vehicles on structured road maps," *2017 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, 2017, pp. 7-12, doi: 10.1109/ICVES.2017.7991893.

[7] J. Rodriguez and S. Ajmal, “OpenGL API Documentation,” Online API documentation. Available: https://docs.gl/