Parallelizing A* algorithm using MPI

Sandya Rani Prasadam
School of Computing, Clemson University
Clemson, South Carolina
Email: sandyap@clemson.edu

Tejaswine Kantu
School of Computing, Clemson University
Clemson, South Carolina
Email: tkantu@clemson.edu

***Abstract*—We have so many search algorithms like BFS(Breadth First Search), DFS(Depth First Search), Branch and Bound, Uniform Cost Search and so on. Above all there is A* search algorithm we normally use in searching problems. A* algorithm is superior to all the others because it takes less time to give or compute the result. When compared to all other algorithms A* normally expands fewer nodes. So time and space complexity will be lower. This is true for almost all the starting states. But when the starting state we give is far from the goal state A* algorithm is taking more time to reach the goal state. So our goal is to parallelize A* algorithm so that the goal state will be reached faster even when the starting state is far from the goal state.**

1. Introduction

Artificial Intelligence came a long way and i taking part of almost every field. AI became a booming field ad so many are looking at it right now. There are so many things we can do make do with AI. AI has so many features, branches, algorithms and many applications. One of the application of AI is searching. Search algorithms help the AI agents to attain the goal state through the assessment of scenarios and alternatives. The algorithms provide search solutions through a sequence of actions that transform the initial state to the goal state. The faster we attain the goal state the more the powerful or optimal the search algorithm is. We have so many search algorithms in AI and A* is one of them and powerful among them.

1.1. A* algorithm

A* is an informed search algorithm also known as best-first search, It is formulated in terms of weighted graphs, it finds the path to the given goal node starting from a specific starting node of a graph with the smallest cost possible. It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration , A* needs to determine which of its paths to extend. It does so based on the f(n) value of each node. A* selects the path that minimizes f(n)=g(n)+h(n) where n is the next node on the path, g(n) is the cost of the path from the start node to n, h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal. A* terminates when the goal is reached or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible, A* is guaranteed to return a least-cost path from start to goal. Admissible means that it never overestimates the actual cost to get to the goal.

A* is implemented using a priority queue also known as the open set or fringe to select the node with minimum (estimated) cost at each level or iteration to expand further. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a removed node is a goal node. The f value of that goal is then the cost of the shortest path, since h at the goal is zero in an admissible heuristic.

There are a number of simple optimizations or implementation details that can significantly affect the performance of an A* implementation. The first detail to note is that the way the priority queue handles ties can have a significant effect on performance in some situations. If ties are broken so the queue behaves in a LIFO manner, A* will behave like depth-first search among equal cost paths (avoiding exploring more than one equally optimal solution).A search algorithm is said to be admissible if it is guaranteed to return an optimal solution. If the heuristic function used by A* is admissible, then A* is admissible.

The time complexity of A* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: $O(b^d)$, where b is the branching factor (the average number of successors per state). This assumes that a goal state exists at all, and is reachable from the start state; if it is not, and the state space is infinite, the algorithm will not terminate. The space complexity of A* is roughly the same as that of all other graph search algorithms, as it keeps all generated nodes in memory.

A* is often used for the common pathfinding problem in applications such as video games, but was originally designed as a general graph traversal algorithm. It finds applications in diverse problems, including the problem of parsing using stochastic grammars in NLP. Other cases

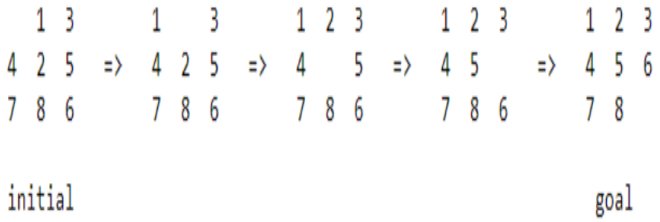

Figure 1. 8 puzzle problem

include an Informational search with online learning

1.1.1. A* Pseudo code.

START ← a node with STATE==problem.INITIAL-STATE, PATH-COST=0, H-VALUE=Heuristic(node.STATE)
OPEN.insert(START) // Open is a priority queue
CLOSED = []
while !OPEN.empty()
node = OPEN.pop()
if node.STATE==GOAL return path(node)
CLOSED.insert(node.STATE)
for each child of node:
if child.STATE not in OPEN and not in CLOSED
OPEN.insert(child)
else if child.STATE in OPEN with higher PATH-COST
update that OPEN node with child

1.1.2. Advantages of A* algorithm.

- It is optimal search algorithm used to solve complex search problems in terms of heuristics.
- A* expands fewer nodes than any other algorithm to find optimal path
- It is one of the best heuristic search techniques
- It will always find the solution in shortest path
- Best of all, it is relatively quick because it assesses the best direction to explore at each stage of the search, rather than blindly searching every available path in a predetermined order.
- There is no other optimal algorithm guaranteed to expand fewer nodes than A*.

.

1.1.3. Applications of A* algorithm.

- It is commonly used in web-based maps and games to find the shortest path at the highest possible efficiency.
- A* is used in many artificial intelligence applications, such as search engines.
- It is used in other algorithms such as the Bellman-Ford algorithm to solve the shortest path problem.
- The A* algorithm is used in network routing protocols, such as RIP, OSPF, and BGP, to calculate the best route between two nodes.
- Used in tower defence game.
- Used in traffic navigation.

.

1.2. 8 puzzle problem

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. The 8-puzzle problem belongs to the category of “sliding block puzzle” type of problem. The 8-puzzle i s a square tray in which eight square tiles are placed. The remaining ninth square is uncovered. Each tile in the tray has a number on it. A tile that is adjacent to blank space can be slide into that space. The game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around. The state of 8-puzzle is the different permutation of tiles within the frame. The operations are the permissible moves up, down, left, right as shown in fig 1.

The 8-puzzle is the largest possible N-puzzle that can be solved entirely. It is simple and yet has a significant problem space. There are larger variants to the same problem type, like the 15-puzzle. But those cannot be solved to completion. This complexity makes the N x N extension of the 8-puzzle an NP-hard problem. 8 puzzle has 9! possible tile permutation states. Out of these, every second permutation state is solvable. Hence, there are a total of 9!/2 = 181,440 solvable problem states.

Alexander Reinefeld from the Paderborn Center for Parallel Computing, Germany, has shown that the average length of all optimal solution paths is about 22 moves for any given random configuration. For the 181440 solvable configurations, there are a total of 500880 optimal solutions. This gives an average solution density of 2.76 per problem, with the minimum and maximum number lying at 1 and 64 solutions.The problem lies in creating the possible search tree and then traversing the most optimal tree branch that will lead from the start state to the end state.

2. How to solve 8 puzzle using A* algorithm

The key feature of the A* algorithm is that it keeps a track of each visited node which helps in ignoring the nodes that are already visited, saving a huge amount of time. It also has a list that holds all the nodes that are left to be explored and it chooses the most optimal node from this list, So we use two lists namely ‘open list‘ and ‘closed list‘ the open list contains all the nodes that are being generated and are not existing in the closed list and each node which is expanded further after discovering it’s neighboring nodes is put in the closed list and the neighbors are put in the open list this is how the nodes expand. Each node has a pointer to its parent so that at any given point it can retrace the path to the parent. Initially, the open list holds the start node. The next node chosen from the open list is based on its f score, the node with the least f score is picked up and expanded further.

f-score=h-score+g-score
h gives how far the goal node is and g the number of nodes traversed from the start node to the current node. For h, we will use the Manhattan distance, and for g, we will use the depth of the current node. The Manhattan distance heuristic is used for its simplicity and ability to estimate the number of moves required to bring a given puzzle state to the solution state. Manhattan distance is computed by the sum of the distances of each tile from where it should belong.For example, the Manhattan distance between ”213540678” and ”123456780” is 9.

We first move the empty space in all the possible directions in the start state and calculate the f-score for each state. This is called expanding the current state. After expanding the current state, it is pushed into the closed list and the newly generated states are pushed into the open list. A state with the least f-score is selected and expanded again. This process continues until the goal state occurs as the current state. Basically, here we are providing the algorithm a measure to choose its actions. The algorithm chooses the best possible action and proceeds in that path.This solves the issue of generating redundant child states, as the algorithm will expand the node with the least f-score. Here is an figure 2.

3. Why should we parallelize A* algoritm

Though A* algorithm is faster than any other algorithm it still takes significant time when we give input that is far from the goal state. For example in the input we took in our program it took nearly 15 minutes to give output and actions it took to reach the output. In this era we need the output to be given faster. When compared to other search algorithms, A* gives output nearly twice as faster but it is still not enough. We need it to be a bit faster so that it can be used efficiently. If we are using,let us say, search engines, which became integral part of our daily life, we want the result to be displayed within milliseconds. if it takes minutes we will not use it. Or if we are using maps while driving and we searched for one destination. If the route for that destination takes so much time we may take some wrong turns in the meantime. So we thought parallelizing it would be better solution. And we are using MPI programming to do that.

4. Background and motivation

When we worked on a work dealing with solving 8 puzzle problem, we used BFS, DFS, uniform cost search and A* algorithm. A* algorithm was faster than the others but when we gave certain inputs the time it took to solve it was significantly high. We came to know that it is because the f value we discussed in 1.1 was higher for these outputs and apparently the distance between the goal and input state is higher. So we wanted to use MPI programming to reduce the time of execution. We also added Wtime so that we will know the time it is taking to execute the program.

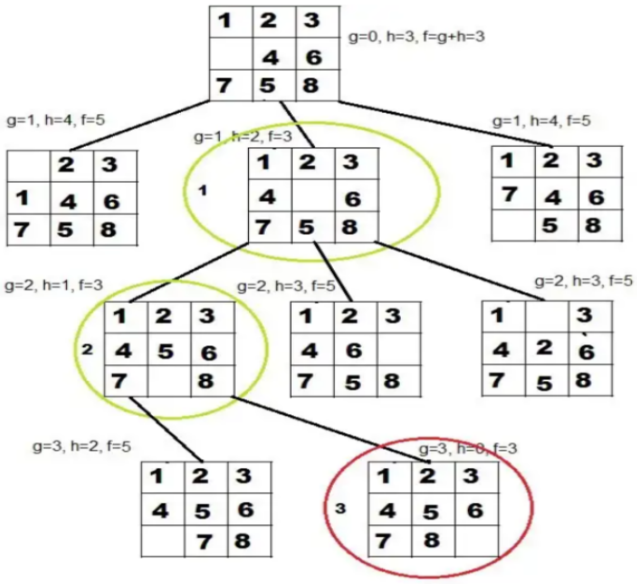

Figure 2. solving 8 puzzle using A* algorithm

5. Plan and design

Our main goal is making the slave processes work as fast as possible So we always focus on how to make the communication time as less as possible First, we make the master process getting the start board and the goal board Then, we let the master process run an modified a * algorithm The modified a * algorithm have the following goals:

- making a new boards to use them later.
- the number of the new boards should be as near as possible from the number of the entered processes.
- ensure that we did not find the goal node while creating those boards. And if so we should terminate the process because no need for the slaves work.

. If the master process create the suitable number of the boards and did not find the goal boards while the previous steps then it should send each slave processes the g “distance” and the board size and the suitable board and the goal board Then all the slave processes will receive that information and run the serial a star code then send the number of the steps and it’s rank to the master process Then the master process will receive that info from the slaves and will examine that info to find the minimum cost node then will print the path. We created our program in such a way that the master program will receive data from all the slave processors and the distance that slave process is processed. You will also see what children state a slave process took and the from that child node to the goal node. the program will print the path it received first from one process and them the next process and so on. And the process are sorted in the order of the distance they covered.

Input	Sequential(8 core)	Parallel 8 cores	12 cores	16 cores
0 5 4 8 7 2 3 1 6	16.939	3.20909	2.15759	3.26329
0 7 4 5 2 6 8 3 1	16.519	3.4861	0.651033	0.77023
5 4 0 8 6 2 7 1 3	21.213	5.29081	6.58833	5.39357
0 8 5 4 1 6 7 2 3	20.399	4.97298	0.7816	1.04187
2 3 0 8 6 4 7 5 1	22.018	6.84657	8.00556	12.8881
8 0 6 5 4 7 2 3 1	1040.183	622.726	213.019	226.388

Figure 3. Outputs

6. Evaluation

We ran our algorithm for various inputs on sequential algorithm and parallel using many cores. We observed that parallel, when used 8 cores, worked nearly 4 times better than sequential. When used more number of cores, like 12 and 16, they are significantly faster than the sequential execution but they are, not always, faster than 8 cores implementation. There is fluctuations of timing differences. Some times the speed increased sometimes decreased as shown in fig 3. This evaluation is done for the output 1 2 3 4 5 6 7 8 0. But we created the program in such a way that the goal states can be changed i.e., we can give different goal states at the time of execution.

7. Challenges

- Limiting the communication between processes was hard. We were not able to stop the program execution when the goal is found by one process.
- Since functions are called by MPI functions we stopping the function call after particular time became difficult.
- Some processes found the goal faster but the program ran till all the processes are done with their execution.
- Even though it is significantly faster than sequential a* algorithm, the program that ran on 8 cores is taking more time for inputs that are too far from the goal node.
- We thought using more nodes will make the goal finding faster. But that was not the case for every input.

8. Conclusion

We described what A* algorithm is along with its advantages and applications. We included what 8 puzzle is and how it is solved. We also described how we can parallelize A* algorithm to decrease its time consumption. We showed couple of examples of 8 puzzle problem.

Solving 8 puzzle problem sequentially or rather conventionally takes O(b^d) time where b is the branching factor and d is the depth. When d increases, time complexity increases exponentially resulting in an increase in time taken to solve the problem. So in order to solve the problem faster we parallelised the creation of children nodes from the parent node. We did that using the master slave process by making the communication time between slaves as less as possible. First, we make the master process of getting the start board and the goal board. Then, we let the master process run the a * algorithm. If the master program is not able to solve the problem, it should send each slave processes the g "distance" and the board size and the suitable board and the goal board Then all the slave processes will receive that information and run the serial a star code then send the number of the steps and it's rank to the master process. Then the master process will receive that info from the slaves and prints the steps and time taken and its rank. Even though we faced some challenges in between, we found that we were able to reduce the time taken by the sequential algorithm to 4 times using parallel algorithm. And the time further decreased when more number of process were involved depending on the input.

References

[1] https://en.wikipedia.org/wiki/A*$_search_algorithm$

[2] https://www.geeksforgeeks.org/a-search-algorithm/

[3] https://www.d.umn.edu/ jrichar4/8puz.html

[4] https://faramira.com/solving-8-puzzle-problem-using-a-star-search/

[5] https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288

[6] Aditya Nongmeikapam, *Parallel Breadth First Search Using MPI* https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Aditya-Nongmeikapam-Spring-2022.pdf

[7] Aydın Buluç, Kamesh Madduri, *Parallel Breadth-First Search on Distributed Memory Systems* https://people.eecs.berkeley.edu/ aydin/sc11$_bfs.pdf$

[8] Charles E. Leiserson, Tao B. Schardl, *A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducer Hyperobjects)*,22nd ACM Symposium on Parallel Algorithms and Architectures 2010 http://web.mit.edu/neboat/www/presentations/spaa2010.pdf

[9] MaximNaumov, AlyssonVrielink, MichaelGarland, *ParallelDepth-FirstSearch forDirectedAcyclicGraphs*, https://research.nvidia.com/sites/default/files/publications/nvr-2017-001.pdf

[10] Jon Freeman, *Parallel Algorithms for Depth-First Search Parallel Algorithms for Depth-First Search*, https://repository.upenn.edu/cgi/viewcontent.cgi?article=1444context=cis$_reports$