Analysis of Parallel A* and Other Least-Cost Path Algorithms

Thomas Harmeyer, Ethan Woollet, Robert Bereiter, Tiuna Benito, Kieran Jimenez

***Abstract*—COP4520 is a class that focuses on parallel and distributed paradigms, as well as the tools needed to support them. The purpose of such paradigms is generally to improve the speed of a system, but supporting them requires an additional level of complexity compared to sequential programming. This is caused by the need to communicate between threads and maintain accurate shared memory. Due to the importance of knowing the best tool for a given task, our team has decided to analyze and compare four algorithms with sequential and parallel implementations. For this project our team has decided to focus on *Least-Cost Path Algorithms* due to their broad and adaptable nature, and the versatility of *weighted graphs* to abstractly represent real world problems. We will test single thread implementations and varying levels of multi-thread implementations on multiple graphs of differing sizes. Then, we will perform an *empirical runtime analysis* on the performances of the different implementations to determine the optimal number of threads to manage for each algorithm. It is the hope of our team that such research will help inform future programmers which implementation will ensure maximum efficiency depending on the graph size expected.**

***Index Terms*—Multithreading, Shortest path problem, Graph theory**

I. Introduction

THE GOAL of this project is to provide an analysis of Least-Cost Path Algorithms and how they behave when applied with parallelization. To illustrate this we first need to look at Least-Cost Path Algorithms and understand how they work and what parallelization does in regards to speed.

A cost distance analysis is used to find the shortest route from one point to another point. There are numerous algorithms which have been developed for this such as A* and Dijkstra's algorithm, each with their own method. These algorithms primarily differ in how they decide which point they should next try. These algorithms can be used to not just find the path between two points, but also between two nodes in a graph as well.

When using a graph we can use the nodes and edges between them to represent many common problems. There are a few kinds of graphs that exist, but our project will focus on finding the shortest path in a specific type: the undirected weighted graph. The word ”undirected” is used to refer to the edges between nodes not having a set direction of traversal. This means that if Node A has an edge to Node B, Node B has the same edge to Node A. The word ”weighted” is used to refer to arbitrary weight values placed on each edge. A common example is of a region where cities are the nodes and the weights are the travel times it takes to go from City A to City B.

These algorithms take a source location and a destination and evaluate their possible paths until a best route is found. This best route is evaluated as the series of nodes visited that start from the source node, end at the destination node, and has the lowest possible total weight from each edge traversed with respect to every other path from the source to the destination. The time it takes for an algorithm to find the best route from the source node to the destination node depends on a few factors. These factors include how big the graph is i.e. how many nodes are present in the graph, and whether the algorithm is an efficient one. For example, some algorithms try to predict the best route as it goes while others try every possible combination until they find the shortest route. As should be obvious, larger graphs and algorithms that try more combinations take more time to navigate than their respective alternatives.

Graph searches have a lot of applications, both for math and real world problems. In this project we will be focusing on the Least-Cost Path Algorithms for graphs in a two dimensional space and understanding how their performance and execution time is affected by increasing the numbers of threads and nodes.

UCF
March 28, 2023

II. Problem Statement

We currently know that parallelization generally speeds up a process, but has to spend some resources managing the threads. We want to experimentally discover how different levels of parallelization perform searching different sizes of graphs. As graphs have broad applications, researching the speeds of multi-threaded search algorithms will lead to better resource management and less unnecessary effort in the future.

III. Related Works

J. Ma and X. Guan in [1] proposed the Water Flow Search (WFS) algorithm. WFS is a combination of the traditional Depth First Search (DFS) and Breadth First Search (BFS) algorithms. When WFS reaches a node it spawns threads to perform Depth First Searches along each outgoing edge. Each thread marks nodes as having been reached, and a node's status is visible to all threads. A thread pool is used as an alternative to the costs of creating and destroying multiple short-lived threads. They concluded that while WFS better utilized the CPU compared to unthreaded algorithms, they were unsure if it had the shortest running time.

S. Belhaous et al. in [2] proposed an implementation of the A* search algorithm that is parallelizable. Their implementation took the data-parallelism approach. First they initialize a thread for each of the neighboring nodes to the source node. Each thread has a local priority queue used to determine the next edge to travel and conducts an A* search. Their experimental data using a graph representing city travel times in Morocco supported the conclusion that their parallel implementation of A* had a shorter execution time compared to a sequential implementation, and it satisfied correctness by finding the correct optimal path.

IV. Technique

For this project, the primary goal is to attempt to clearly identify under what circumstances the different algorithms are preferable. To do this, our method will be to vary those circumstances and take measurements of the results for further analysis.

Specifically, the variables of interest we will be modifying are:

- Algorithm used
- Single thread vs multi thread
  - Multi thread options: 1, 2, 4, 8 threads
- Size of graph
  - 10, 50, 100, 1000, 10000 nodes

We will use all of these different points of comparison in our analysis to determine if there is a good rule of thumb, or objective solution to this type of problem. To do this, we will create randomized graphs and measure the time it takes for each search implementation to find a path between the source and destination nodes.

In terms of single thread versus multi thread, it is important to point out that the single threaded algorithms refer to the standard implementation of the algorithms being discussed without any consideration for multi-threading or the concurrent overhead it would require. This is also why we will be testing with a single thread implementation of the basic algorithms and comparing it to the multi-threaded algorithms using a single thread. This will provide a baseline comparison to help determine what the overhead costs for these algorithms are, in terms of extra execution time.

Another important discussion point is how we will generate said graphs for testing. Because these are simply weighted graphs, writing a class to support that is quite simple, and it is our intention to treat the graphs as being read-only, which means that no changes need to be made in regard to the parallelized algorithms.

To generate the graphs tested, we first initialize an empty graph with n nodes. Then we randomly select two distinct nodes; one for the source, and one for the destination. Next we select the source node and a random node. We then add an edge between these two nodes with a random weight. Then we repeat, selecting the second node and picking another random node. This process creates a chain of connected nodes with the source node at one end. We continue chaining nodes together until the last node added is the desired destination node. Before finishing we add a few random edges to create some branching and variability in the graph.

For generating these graphs and testing the algorithms, we are organizing our solutions into classes using a common interface and calling those solutions from a runner file, *main.cpp*. Said runner file will also be responsible for generating the graphs randomly and performing the measurements on the solutions' run-times. Generating a graph along with its source and destination nodes is handled by *generateNewProblem()*. Testing is handled by the *testUnthreaded(algorithm, graphs)* and *testThreaded(algorithm, threadCount, graphs)* functions. They simply store the difference in milliseconds between the start and end times for each algorithm and graph combination.

One last topic of note is the CPU(s) used in testing. We will of course indicate what CPU is used to perform the tests, as well as their specifications. We will be running the test on multiple computers and averaging the results for the final analysis. If the differences between the results on each CPU proves significant or relevant to our Problem Statement we may include them as another axis of comparison and a discussion of their meaning in the appendix. At this point we do not predict the differences as a result of the CPU size to have any major bearing on our research. This is because we are focusing on the speedup between different algorithmic implementations. We expect the ratio between execution times for differing levels of threads to not be statistically different across researcher computers. If there is a significant difference, an examination of the effects of computer hardware and Operating System would make a great subject for an appendix entry or even a follow-up paper.

A. The Algorithms

This study focuses on analyzing four Least-Cost Path Algorithms,; A*, Breadth-First Search (BFS), Depth-First Search (DFS), and Dijkstra's Algorithm, in both single-threaded and multi-threaded implementations on varying graph sizes. The aim is to determine the optimal number of threads for each algorithm and to understand their behavior with parallelization. In an effort to mitigate the unknown factors in library implementations of these algorithms, the research team will be writing each algorithm personally.

- A* is a popular algorithm used in graph theory to find the shortest path between two nodes in a graph. It combines the strengths of Dijkstra's algorithm and breadth-first search to find the optimal solution efficiently. A* uses a heuristic function to estimate the cost of reaching the goal from a given node, allowing it to prioritize exploring promising paths over others.
- BFS, or breadth-first search, is a graph traversal algorithm that visits all the vertices of a graph in breadth-first order. This means that it visits all the vertices at a given depth level before moving on to the next level. BFS is used to find the shortest path between two nodes in an unweighted graph, as well as to solve many other graph-related problems.
- DFS, or depth-first search, is another graph traversal algorithm that visits vertices in a depth-first manner. This

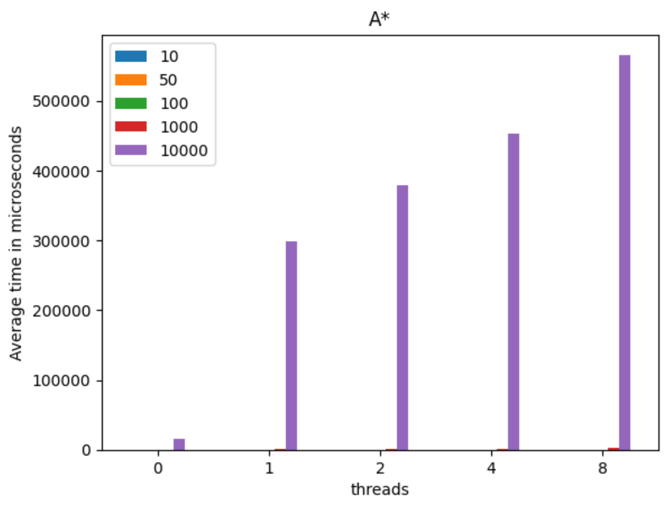

Fig. 1. A* Performance Chart

means that it visits a vertex and then explores all of its neighbors before moving on to the next vertex. DFS is used to find cycles and paths in a graph, and to solve many other graph-related problems.

- Dijkstra's algorithm is a popular algorithm used to find the shortest path between two nodes in a graph with non-negative edge weights. It works by maintaining a priority queue of vertices and visiting them in order of increasing distance from the source node. Dijkstra's algorithm is guaranteed to find the optimal solution and is widely used in many applications.

Our language of choice for this project is C++. We chose C++ for its efficient implementation of thread objects. A link to our public project page will be provided in the appendix.

V. Evaluation

The data we will generate from our tests will be formatted into a CSV string. Then it will be used by an analysis script, *analysis_script.ipynb*, written in python. The script utilizes the Pandas, Numpy, and MatPlotLib packages to create tables and plotted visual representations of the data for ease of analysis.

The data will also include bar charts for the algorithms' performances. The results of the algorithms will be measured over ten graphs per 'version' to account for natural variance of performance and averaged. Overall, it will be relatively simple, but with many possible comparisons to be made within and across the different algorithms.

The performance of the A* algorithm hinges on the priority queue that organizes and calculates the optimal path through the graph of nodes. Larger and more densely-packed graphs would complicate the program and increase calculations, though the histrionic would help in prioritizing steps closer to the end node. Parallelizing the A* algorithm provided complications. While the code for utilizing a priority queue in threads is there, examples of the code were found in research papers, and in forms we found difficult to translate to our own program. In theory, a fully parallelized priority queue would

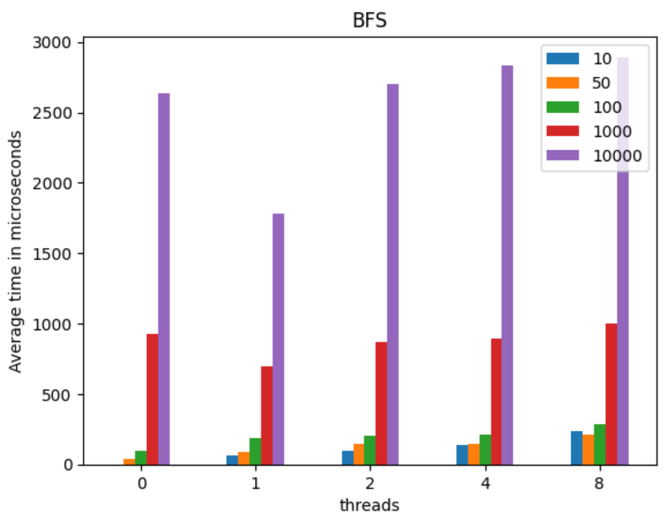

Fig. 2. BFS Performance Chart

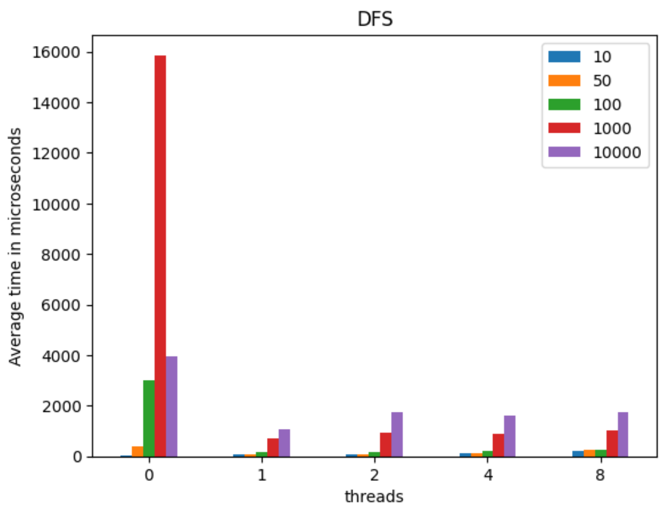

Fig. 3. DFS Performance Chart

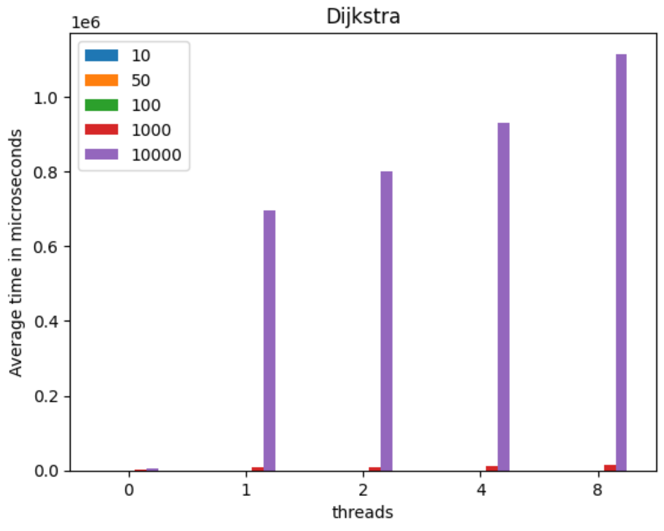

Fig. 4. Dijkstra's Algorithm Performance Chart

greatly reduce the time A* would take, but without it, it is left running longer than BFS and DFS.

The performance of BFS(Breadth-first search) depends mostly on the number of vertices in the graph. The tame it takes will always be proportional to the number of nodes it has to go through. It has to store the queue used in the algorithm which contains the nodes that have yet to be visited. This can be bad for graphs that are on the larger sides as the stored data will increase significantly. This algorithm checks all the nodes on the current depth nodes that are distance one, then it increases the distance and checks those again. As such, it will tend to find the shortest path granted that the graph structure allows it. The neighbors are added to the back of the queue so it knows which ones have been visited and which ones it has yet to visit.

As for DFS (Depth-first Search) it behaves similar to BFS but it goes as deep as it can go through one branch in the graph. It then comes back to the start and tries the other neighbors it did not go through, this makes it a very brute force kind of algorithm that benefits greatly from concurrency as you can assign each of these paths to a thread and have them all happen at the same time, it just needs one variable that gets changed at the start before each thread goes into their respective path. It uses less memory than BFS as it only stores the path from the root node to the current node, however it is also possible for this one to get stuck in a loop if not implemented properly. It should be handled that if the same nodes are visited enough times it should break away from that loop.

Dijkstra's performance is entirely dependant on the kind of data structure used for implementation, the common ways are through a priority queue and a heap data structure. These both end up behaving in similar ways but end up having different run times. We used the priority queue approach as it was easier to implement. Dijkstra's algorithm performs well on graphs with few edges but tends to slow down on denser graphs as it has to visit each node separately. If done with a heap data structure, the worst case scenario would be slightly better as it can take the minimum element for each calculation rather than having to compare it through each iteration. When it comes to parallelizing the algorithm, the challenges become finding an efficient way to use the threads to do different work independently, as it stands the algorithm uses information from each neighbor node in calculating the next path it should take. This leads to the threads needing to wait for the others to finish before they can decide on which path to go through, if we were to let them do each path on their own this would not be the shortest path found to node B from node A but rather the shortest path from node A to every other node in the graph. Ways of parallelizing the algorithm are found in research papers and they do offer a significant boost over the normal approach, but their implementation tends to be complicated.

Overall, it is hard to say how a single-thread program vs a multi-thread program with 1 active thread performs. In Dijkstra's Algorithm and the A* algorithm implementations, there is a dramatic difference between the two cases: in both implementations the single-thread program performed much faster than the multi-thread program using one thread. This implies that the additional steps used to manage multi-threading capabilities such as properly validating, acquiring and releasing locks, and so on come with a hefty cost to program execution time. However, this is apparently contradicted by the other two implementations: BFS minorly improved in the 1000 and 10000 node cases using multi-threading with 1 active thread compared to the single-threaded implementation while DFS greatly improved.

The purpose of this research was an effort to determine if there were any significant or obvious benefits to multi threading these algorithms. With that goal in mind, we cannot objectively say that there is an easy rule of thumb. What we can say, however, is that there are some obvious patterns that emerged when were testing our implementations of these algorithms.

For both A* and Disjktra's, we can clearly see a linear pattern forming for the threaded times. It is important to recognize this because the number of threads being used is increasing exponentially. This implies that the number of threads does not linearly increase performance, which is not an obvious fact. That conclusion does not apply to BFS or DFS, which seems to perform very similarly across a thread pool of any size greater than 1. This could be due to a number of reasons. However, when we take into account the scale of those measurements, they could very well be costs incurred by spawning and terminating threads coupled with the relatively low level of complexity of those algorithms. If that is the case, then the times would have little to no difference as the majority of the time measured would have been spent on the overhead of managing multiple threads.

VI. Discussion

For the loops within the graphs, most of the complicated ones use a system to keep track of which nodes have been visited. This way it will not get stuck in a loop. For the simple algorithms since it performs a search through every possible pathway it will not consider the same path over and over again.

Most of the problems encountered had to do with a way to parallelize the movement through the graphs, as the algorithms are trying to find the shortest path as they transverse through them, there has to be a way to assign this method to multiple threads. As such, we need them to be synchronized which leads to needing to wait for threads to finish so they can communicate data before taking a decision. Because of this some algorithms do not take full advantage of a parallel approach, but it is very effective with the algorithms that brute force their way to the shortest path, as they can just split all the possible combinations amongst the threads. Some of the algorithms should be changed for a better approach, the only downside is that these implementations are not straight forward as making use of different threads in an algorithm that needs previous information to take decisions is challenging.

There is still much research to be done in this field. When running our final tests they were all run on a single system. This system ran Windows 10, had 16 GB of RAM, and an i5-10600k processor. However, systems that have different OSes and different processors in particular may see different results.

These could lead to better performance for single-threaded or for multi-threaded algorithms. As such, programmers and businesses who use drastically different hardware or different operating systems would benefit from similar trials examining differences across different computer systems.

In addition to hardware and operating system differences, future researchers would do well to examine performances using different program implementations. More precisely, similar implementations in Java, Rust, and other programming languages popular in the industry would do well in expanding the scope of who would find this topic useful. Similarly, the compiler used could play a role in how the machine code of the instructions is interpreted.

Future studies might also find that different procedures would prove more enlightening. Using graphs with higher node counts or graphs guaranteed to be connected graphs are both good options. The first would prove relevant to extremely large databases such as those employed by Microsoft, Amazon, and Google. The latter could help remove some variation from the results even if validating their connectivity would take some significant time. Finally, including more intermediary graph sizes could help better illustrate the trends in increasing the thread count.

VII. Conclusion

In conclusion, our team analyzed four different algorithms for finding the least-cost path in a graph: A*, Breadth-First Search, Depth-First Search, and Dijkstra's Algorithm. The algorithms were implemented both in single-threaded and multi-threaded versions, and were tested on different graph sizes to determine the optimal number of threads for each algorithm. The results of the tests were analyzed using Python's Pandas, Numpy, and MatPlotLib packages.

In the end, we did not come to any broad conclusion from our testing, but some useful ideas can still be learned. When using multiple threads with relatively short execution time per job, the cost of multi threading will likely play a large role in limiting the lower bound of performance at scale. For multiple threads with higher execution time per job, the relationship between performance gain per added core provides diminishing returns, which is in accordance with Amdahl's law.

We do not believe that this will have any impact on the current state of multi threaded programming, but it does reinforce what is already known in regard to threading and the costs and benefits. Our only recommendation is that more time and effort should be spent on taking classic algorithms and attempting to rewrite them to take further advantage of modern multi core CPU architectures and to learn more about the lower bounds of threaded applications of those algorithms.

Appendix A
GitHub Project Page

You can find our project and code publically hosted at *https://github.com/Parra-Parallel-Project/parallel-research-project*. A brief explanation of the files is as follows:

- main.cpp - The runner file
- WeightedGraphh.cpp - Handles graph creation functions
- times.csv - Intermediary file that stores trial results
- analysis_script.ipynb - Reads data from times.csv, finds time averages, and plots bar charts
- BFS.cpp - BFS implementations
- DFS.cpp - DFS implementations
- Dijkstra.cpp - Dijkstra's Algorithm implementations
- Astar.cpp - A* implementations

References

[1] J. Ma and X. Guan, "An Optimum Search Algorithm of Simulating Hydrokinetics," 2015 8th International Conference on Intelligent Computation Technology and Automation (ICICTA), Nanchang, China, 2015, pp. 157-161, doi: 10.1109/ICICTA.2015.48.

[2] S. Belhaous, S. Baroud, S. Chokri, Z. Hidila, A. Naji and M. Mestari, "Parallel implementation of A* search algorithm for road network," 2019 Third International Conference on Intelligent Computing in Data Sciences (ICDS), Marrakech, Morocco, 2019, pp. 1-7, doi: 10.1109/ICDS47004.2019.8942279.